LNCS 7590

Robert B. France
Jürgen Kazmeier
Ruth Breu
Colin Atkinson (Eds.)

Model Driven Engineering Languages and Systems

15th International Conference, MODELS 2012
Innsbruck, Austria, September/October 2012
Proceedings

Springer

Robert B. France   Jürgen Kazmeier
Ruth Breu   Colin Atkinson (Eds.)

Model Driven Engineering Languages and Systems

15th International Conference, MODELS 2012
Innsbruck, Austria, September 30–October 5, 2012
Proceedings

Springer

Volume Editors

Robert B. France
Colorado State University
Department of Computer Science
Fort Collins, CO 80523, USA
E-mail: france@cs.colostate.edu

Jürgen Kazmeier
Siemens AG
Healthcare Services International, Managed Services
Hartmannstr. 16, 91052 Erlangen, Germany
E-mail: juergen.kazmeier@siemens.com

Ruth Breu
University of Innsbruck
Institute of Computer Science
Technikerstr. 21a, 6020 Innsbruck, Austria
E-mail: ruth.breu@uibk.ac.at

Colin Atkinson
University of Mannheim
Software Engineering Group
A5, 6, 68131 Mannheim, Germany
E-mail: atkinson@informatik.uni-mannheim.de

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-33665-2 e-ISBN 978-3-642-33666-9
DOI 10.1007/978-3-642-33666-9
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012947524

CR Subject Classification (1998): D.3.4, D.2.1-3, D.2.5, D.2.9, D.2.11, D.3.2, F.3.2-3, K.6.3, I.6.3-5, C.4

LNCS Sublibrary: SL 2 – Programming and Software Engineering

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

The MODELS conference serves as the premier venue for disseminating and discussing high-quality work in the area of Model-Driven Development (MDD). The conference series covers all aspects of MDD, including software and system modeling languages, methods, tools, and their applications.

Authors were invited to submit papers to be reviewed for the MODELS 2012 foundations and applications tracks. Foundations-track papers make significant contributions to the MDD body of knowledge in the form of new ideas and results that advance the state of the art in MDD. Three categories of foundations-track papers are included in these proceedings. *Technical papers* describe original scientifically rigorous solutions to challenging model-driven development problems. *Empirical papers* present evaluations of MDD practices, or scientific validations of proposed solutions through, for example, empirical studies, experiments, case studies, simulations, formal analyses, and mathematical proofs. *Exploratory papers* describe new, non-conventional MDD research positions or approaches that challenge the status quo and describe solutions that are based on new ways of looking at MDD problems.

The MODELS 2012 applications track aimed to provide a realistic and verifiable picture of the current state of practice in MDD and to provide a forum for analyzing experience related to the industrial adoption of model-driven techniques. Applications-track papers describe and analyze the application of MDD techniques to industrial case studies, or describe innovative solutions and concepts arising from practical deployment of tools and techniques.

The MODELS 2012 review process was designed to support the role of MODELS as a MDD community development agent and as a dissemination vehicle for high-quality MDD-related knowledge and experience. MODELS responsibilities with respect to community development include providing useful feedback to all authors of submitted papers through written reviews, and ensuring that only the best papers are disseminated through the conference proceedings.

MODELS 2012 introduced a Program Board (PB) to help ensure, to the extent humanly possible, that the reviews received by submitters provided good feedback, and that the selection process was as rigorous and fair as possible. The PB assisted the Program Committee (PC) chairs with the monitoring of reviews submitted by PC members and with initiating and monitoring discussions in the online PC meeting.

In the course of the 2012 review process each paper was reviewed by at least three members of the Program Committee, and the reviews were monitored by a PB member assigned to the paper. Each paper was extensively discussed at the online PC meeting, and due consideration was given to author responses. The PB then met to finalize the selection of papers by making acceptance decisions on those papers for which online PC discussions did not converge on a clear decision. The PB did not have the authority to override any final decisions made during the online PC discussions. In this sense, the PB served primarily to assist the PC chairs in their decision-making role.

This year, out of 151 foundations-track papers submitted, the PC and the PB accepted 35 papers, that is, a 23% acceptance rate. Of the 30 applications-track papers submitted, the PC and the PB accepted 15 papers, resulting in a 50% acceptance rate.

The PC chairs also conducted a submitter survey to obtain feedback on the quality of reviews. Authors of 90 papers (out of a total of 181 papers) rated each review they received. Authors were asked to indicate whether the review was very useful, useful, not useful, or not useful at all. Just over 75% of the respondents indicated that their reviews were either useful or very useful (approximately 27% rated their reviews as very useful and approximated 48% rated their reviews as useful). Feedback like this helps us determine the effectiveness of the MODELS review process and we greatly appreciate the effort of the authors who submitted completed survey forms.

In closing, we thank the PC, the PB, and the additional reviewers who contributed to the MODELS 2012 review process. Their outstanding contributions help ensure that MODELS continues to play a vital role in the MDD community. Thanks to all the authors who submitted papers to MODELS, and we congratulate those authors whose papers appear in these proceedings. These papers reflect the quality of the current state of the art in MDD research and practice. We also thank Richard van de Stadt for his CyberChairPRO support. Lastly, we thank members of the steering and organizing committees for the support they gave us during the planning and execution of the MODELS 2012 review process.

October 2012

Robert B. France
Jürgen Kazmeier
Ruth Breu
Colin Atkinson

Organization

Conference Chairs

Ruth Breu	University of Innsbruck, Austria
Colin Atkinson	University of Mannheim, Germany

Foundations Track Program Chair

Robert B. France	Colorado State University, USA

Applications Track Program Chair

Jürgen Kazmeier	Siemens AG, Germany

Workshop Chairs

Jeff Gray	University of Alabama, USA
Joanna Chimiak-Opoka	University of Innsbruck, Austria

Tutorial Chair

Antonio Vallecillo	University of Malaga, Spain

Exhibition and Demo Chair

Dániel Varró	Budapest University of Technology and Economics, Hungary

Social Media Chair

Richard Paige	University of York, UK

Doctoral Symposium Chair

Yvan Labiche	Carleton University, Canada

Educators' Symposium Chairs

Dan Chiorean	University of Babes-Bolyai, Romania
Benoit Combemale	University of Rennes, France

Panel Chair

Publicity Chair (North America)

Publicity Chair (Europe)

Local Arrangements Chair

Web Chair

Program Board

Program Committee: Foundations Track

Hans Vangheluwe	University of Antwerp, Belgium and McGill University, Canada
Dániel Varró	Budapest University of Technology and Economics, Hungary
Michael Whalen	University of Minnesota, USA
Jon Whittle	Lancaster University, UK
Andrea Zisman	City University, UK
Steffen Zschaler	King's College London, UK

Program Committee: Applications Track

Alfred Aue	Cap Gemini, Germany
Robert Baillargeon	Sodius, USA
Balbir Barn	Middlesex University, UK
Klaus Beetz	EIT ICT Labs, Germany
Nelly Bencomo	INRIA, France
Brian Berenbach	Siemens Corporate Research, USA
Bezhad Bordbar	University of Birmingham, UK
Rudolf Haggenmüller	ITEA, Germany
Øystein Haugen	SINTEF, Norway
Pavel Hruby	CSC, Denmark
Cornel Klein	Siemens AG, Germany
Georg Kreuch	ATOS, Austria
Vinay Kulkarni	Tata Consultancy Services, India
Thomas Mayerdorfer	Siemens AG, Austria
Stephen J. Mellor	Independent Consultant, UK
Dan Paulish	SCR, USA
Isabelle Perseil	INSERM, France
Rob Pettit	The Aerospace Corp., USA
Bernhard Schätz	Technical University of Munich, Germany
Wolfram Schulte	Microsoft, USA
Sebastin Thoma	ATOS, Germany
Juha-Pekka Tolvanen	MetaCase, Finland
Laurence Tratt	King's College, UK
Michael von der Beeck	BMW Group, Germany
Frank Weil	UniqueSoft, USA

Steering Committee

Geri Georg (Chair)
Gregor Engels (Vice Chair)
Jean Bézivin
Ruth Breu
Lionel Briand
Jean-Michel Bruel
Krzysztof Czarnecki
Matthew Dwyer
Øystein Haugen
Heinrich Hussmann
Thomas Kühne
Pierre-Alain Muller

Oscar Nierstrasz
Dorina Petriu
Rob Pettit
Gianna Reggio
Doug Schmidt
Andy Schürr
Steve Seidman
Jon Whittle

Conference Sponsors

Corporate Donors Gold Level

Bronze Level

Supporters

Additional Reviewers

El Arbi Aboussoror
Mathieu Acher
Vander Alves
Anthony Anjorin
Davide Arcelli
Thorsten Arendt
Nesa Asoudeh
Colin Atkinson
Marco Autili
Souvik Barat
Amel Belaggoun
Luca Berardinelli
Gábor Bergmann
Thomas Beyhl
Jan Olaf Blech
Arnaud Blouin
Petra Brosch
Frank Burton
Fabian Büttner
Denis Bytschkow
Alarico Campetelli
Javier Canovas
Franck Chauvel
Hyun Cho
Fabian Christ
Antonio Cicchetti
Harald Cichos
Philippe Collet

Benoit Combemale
Ricardo Contreras
Frederik Deckwerth
Andreas Demuth
Davide Di Ruscio
Hubert Dubois
Keith Duddy
Sophie Ebersold
Jonas Eckhardt
Sebastian Eder
Ramin Etemaadi
Alain Faivre
Jean-Remy Falleri
Claudiu Farcas
Emilia Farcas
Kleinner Farias
Massimo Felici
Henning Femmer
Daniel Mendez Fernandez
Frédéric Fondement
Germain Forestier
Sebastian Gabmeyer
Silke Geisen
Rohit Gheyi
László Gönczy
Juan Gonzalez-Calleros
Baris Güldali
Samir Hameg
Regina Hebig
Ábel Hegedüs
Stephan Hildebrandt
Florian Hölzl
Ákos Horváth
John Hutchinson
Ludovico Iovino
Ferosh Jacob
Dimitris Kolovos
Vasileios Koutsoumpas
Mirco Kuhlmann
Ivan Kurtev
Angelika Kusel
Pieter Kwantes
Fadoi Lakhal
Leen Lambers
Philip Langer
Agnes Lanusse
Marius Lauder
Hervé Leblanc
Maurizio Leotta
Christian Leuxner
Qichao Liu
Malte Lochau
Markus Luckey
Khaled Mahbub
Salvador Martinez
Nikos Matragkas
Tanja Mayerhofer
Bart Meyers
Jakob Mund
Faiza Munir
Benjamin Nagel
Stefan Neumann
Guilherme M. Nogueira
Florian Noyrit
Alexander Nöhrer
Iulian Ober
Lars Patzina
Patrizio Pelliccione
Birgit Penzenstadler
Gilles Perrouin
Holger Pfeifer
Jan Philipps
Suresh Pillay
Ernesto Posse
Alex Potanin
Alek Radjenovic
István Ráth
Sanjai Rayadurgam
Alexander Reder
Filippo Ricca
Louis Rose
Alessandro Rossini
Suman Roychoudhury
Karsten Saller
Patrizia Scandurra
Johannes Schönböck
Martina Seidl
Gehan Selim
Sagar Sen
Filippo Seracini

Christian Soltenborn	Martin Wieber
Hui Song	James R. Williams
Maria Spichkova	Manuel Wimmer
Daniel Strüber	Ernest Wozniak
Sagar Sunkle	Sebastian Wätzoldt
Gerson Sunyé	Nataliya Yakymets
Laurent Thiry	Yan Yan
Massimo Tisi	Paraskevi Zerva
Alessandro Tiso	Wei Zhang
Philipp Torka	Xiang Zhang
Damiano Cosimo Torre	Celal Ziftci
Catia Trubiani	Karolina Zurowska
Thomas Vogel	

Table of Contents

Quantitative Reactive Models*

Thomas A. Henzinger

IST Austria

Formal verification aims to improve the quality of hardware and software by detecting errors before they do harm. At the basis of formal verification lies the logical notion of correctness, which purports to capture whether or not a circuit or program behaves as desired. We suggest that the boolean partition into correct and incorrect systems falls short of the practical need to assess the behavior of hardware and software in a more nuanced fashion against multiple criteria.

We advocate quantitative fitness measures for systems, specifically for measuring various aspects of function, performance, and robustness. Among all systems that are correct with respect to a given set of requirements, usually some are preferred to others; and among all systems that are, in a strict boolean sense, incorrect, surely some are more acceptable than others. A quantitative theory of reactive models provides such preference metrics and acceptability measures for reactive systems, that is, systems —such as concurrent and embedded processes— which interact with their environment in a sequence of steps over time.

An appealing quantitative theory should support quantitative generalizations of the paradigms that have been success stories in qualitative reactive modeling, including

– executability and the distinction between safety and liveness considerations;
– compositional design and analysis of complex systems;
– property-preserving abstractions and abstraction refinement;
– specification by temporal logic and verification by model checking;
– automatic synthesis by solving games played on state spaces.

A quantitative theory may be probabilistic, and it may refer to the consumption of resources such as time, space, and power, but we focus on quantitative theories that replace the classical boolean notion of correctness by numerical distances between systems and requirements. For example, if a system does not satisfy a requirement, then various correctness distances measure how close the system comes to satisfying the requirement. Dually, if a system satisfies a requirement, then various robustness distances measure how much the system can be perturbed without violating the requirement [1]. Synthesis, which in a qualitative setting is a constraint-satisfaction problem, becomes —in a quantitative framework— an optimization problem, which asks for the construction, from a set of possibly inconsistent requirements, of the preferred system [2,3].

* This work was supported in part by the ERC Advanced Grant QUAREM and by the Austrian Science Fund NFN on Rigorous Systems Engineering (RiSE).

References

1. Cerný, P., Henzinger, T.A., Radhakrishna, A.: Simulation Distances. Theoretical Computer Science 413, 21–35 (2012)
2. Cerný, P., Chatterjee, K., Henzinger, T.A., Radhakrishna, A., Singh, R.: Quantitative Synthesis for Concurrent Programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 243–259. Springer, Heidelberg (2011)
3. Cerný, P., Gopi, S., Henzinger, T.A., Totla, N., Radhakrishna, A.: Synthesis from Incompatible Specifications. In: Proceedings of EMSOFT. ACM (2012)

Bottom-Up Meta-Modelling: An Interactive Approach

Jesús Sánchez-Cuadrado, Juan de Lara, and Esther Guerra

Universidad Autónoma de Madrid (Spain)
{Jesus.Sanchez.Cuadrado,Juan.deLara,Esther.Guerra}@uam.es

Abstract. The intensive use of models in Model-Driven Engineering (MDE) raises the need to develop meta-models with different aims, like the construction of textual and visual modelling languages and the specification of source and target ends of model-to-model transformations. While domain experts have the knowledge about the concepts of the domain, they usually lack the skills to build meta-models. These should be tailored according to their future usage and specific implementation platform, which demands knowledge available only to engineers with great expertise in MDE platforms. These issues hinder a wider adoption of MDE both by domain experts and software engineers.

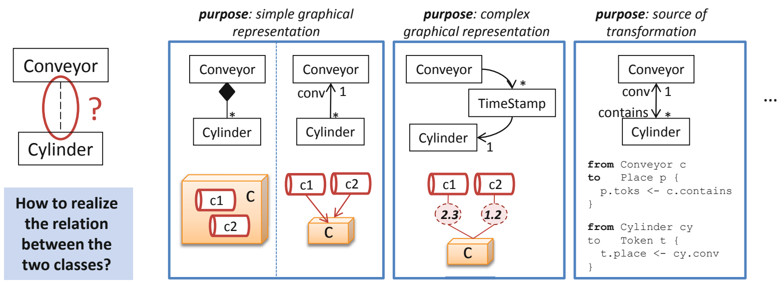
In order to alleviate this situation we propose an interactive, iterative approach to meta-model construction enabling the specification of model fragments by domain experts, with the possibility of using informal drawing tools like *Dia*. These fragments can be annotated with hints about the *intention* or *needs* for certain elements. A meta-model is automatically induced, which can be refactored in an interactive way, and then compiled into an *implementation* meta-model using profiles and patterns for different platforms and purposes.

Keywords: Meta-Modelling, Domain-Specific Modelling Languages, Interactive Meta-Modelling, Meta-Model Design Exploration.

1 Introduction

Model-Driven Engineering (MDE) makes heavy use of models during the development process. Models are usually defined using Domain-Specific Modelling Languages (DSMLs) which are themselves specified through a meta-model. A DSML should contain useful, appropriate primitives and abstractions for a particular application domain. Hence, the input from domain experts is essential to obtain effective, useful meta-models [13].

The usual process of meta-model construction requires first building (a part of) the meta-model which only then can be used to build models. Even though software engineers are used to this process, it may be counter-intuitive to non-meta-modelling experts, which may prefer drafting example models first, and then abstract those into classes and relations in a meta-model. As Oscar Nierstrasz put it, "*... in the real world, there are only objects. Classes exist only in*


Conveyor
?
Cylinder
How to realize the relation between the two classes?
purpose: simple graphical representation
Conveyor
Cylinder
*
Conveyor
conv
1
*
Cylinder
c1
c2
C
purpose: complex graphical representation
Conveyor
*
TimeStamp
Cylinder
1
c1
c2
2.3
1.2
C
purpose: source of transformation
Conveyor
conv
1
contains
*
Cylinder
...

```
from Conveyor c
to   Place p {
  p.toks <- c.contains
}

from Cylinder cy
to   Token t {
  t.place <- cy.conv
}
```

Fig. 1. Different meta-model realizations depending on its future usage

our minds" [19]. In this way, domain experts and final users of MDE tools are used to working with models, but not with meta-models. Asking them to build a meta-model *before* drafting example models is often too demanding. In general, an early exploratory phase of model construction, to understand the main concepts of the language, is recommended for DSML engineering [4,13].

Another issue that makes meta-model construction cumbersome is the fact that meta-models frequently need to be fine-tuned depending on their intended use: designing a textual modelling language (e.g., with xText[1]), a graphical language (e.g., with GMF or Eugenia [14]), or the source or target of a transformation. As illustrated in Fig. 1, the particular meta-model usage may impact on its design, for instance to decide whether a connection should be implemented as a reference (e.g., for simple graphical visualization), as an intermediate class (e.g., for a more complex visualization, or to enable iterating on all connection instances), as a bidirectional association (e.g., to allow back navigation if it is used in a transformation), or as an intermediate class with composition (e.g., to enable scoping). The use of a specific technological platform, like EMF [24], has also an impact on how meta-models are actually implemented, e.g., regarding the use of composition, the need to have available a root class, and the use of references. As a consequence, the *implementation* meta-model for a particular platform may differ from the *conceptual* one as elicited by domain experts. Specialized technical knowledge is required for this implementation task, hardly ever found in domain experts, which additionally has a steep learning curve.

In order to alleviate this situation, this paper presents a novel way to define meta-models and modelling environments. Its ultimate goal is to facilitate the creation of DSMLs by domain experts without proficiency in meta-modelling and MDE platforms and technologies. For this purpose, we propose an iterative process for *meta-model induction* in which model fragments are given either sketched by domain experts using drawing tools like *Dia*[2], or using a compact textual notation suitable for engineers which allows annotating the *intention* of the different modelling elements. From these fragments, a meta-model is

[1] http://www.eclipse.org/Xtext/
[2] http://projects.gnome.org/dia/

automatically induced, which can be refactored if needed. Finally, the resulting meta-model is compiled into a given technology (e.g., EMF or MetaDepth [8]), optimized for a particular *purpose* (visual or textual language, transformation) and a particular tool (e.g., xText or GMF).

Paper Organization. Section 2 overviews the working scheme of our proposal. Its main steps are detailed in the following sections: specification of fragments (Section 3), meta-model induction and refactoring (Section 4), and compilation of the induced meta-model for different purposes and platforms (Section 5). Next, Section 6 presents tool support. Finally, Section 7 compares with related research and Section 8 ends with the conclusions.

2 Bottom-Up Meta-Modelling

Interactive development [21] promotes rapid feedback from the programming environment to the developer. Typically, a programming language provides a *shell* to write pieces of code, and the running system is updated accordingly. This permits observing the effects of the code as it is developed, and to explore different design options easily. This approach has also been regarded as a way to allow non-experts to perform simple programming tasks or to be introduced to programming, since a program is created by defining and testing small pieces of functionality that will be composed bottom-up instead of devising a design from the beginning.

Inspired by interactive programming, we propose a meta-modelling framework to facilitate the integration of end-users into the meta-modelling process, as well as permitting engineers with no meta-modelling expertise to build meta-models. The design of our framework is driven by the following requirements:

– *Bottom-up.* Whereas meta-modelling requires abstraction capabilities, the design of DSMLs demands, in addition, expert knowledge about the domain in two dimensions: horizontal and vertical [1]. The former refers to technical knowledge applicable to a range of applications (e.g., the domain of Android mobile development) and experts are developers proficient in specific implementation technologies. The vertical dimension corresponds to a particular application domain or industry (e.g., insurances) where experts are usually non-technical people. Our proposal is to let these two kinds of experts build the meta-models of DSMLs incrementally and automatically starting from example models. Afterwards, the induced meta-model can be reviewed by a meta-modelling expert who can refactor some parts if needed.
– *Interactive.* Creating a meta-model is an iterative process in which an initial meta-model is created, then it is tested by trying to instantiate it to create some models of interest, and whenever this is not possible, the meta-model is changed to accommodate these models [13]. The performed changes may require the detection of broken test models and their manual update. In our proposal, if a new version of the meta-model breaks the conformance with existing models, the problem is reported together with possible fixes.

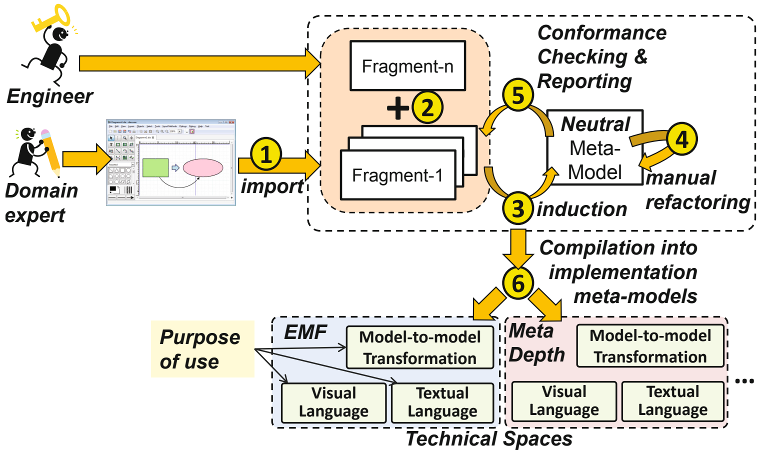


Fig. 2. Working scheme of bottom-up meta-modelling

– *Exploratory.* The design of a meta-model is refined during its construction, and several choices are typically available for each refinement. To support the exploration of design options, we should let the developer annotate his example models with hints about his intention, which are then translated into some meta-model design decision. If two models contain conflicting annotations, this is reported to the developer who can decide among the different design options. We also consider the possibility of rolling back a decision.
– *Implementation-agnostic.* The platform used to implement a meta-model may enforce certain meta-modelling decisions (e.g., the use of compositions vs references). This knowledge is sometimes not available to meta-modelling experts, but only to experts of the platform. For this reason, we postpone any decision about the target platform to a last stage. The meta-models built interactively are neutral or implementation-agnostic, and only when the meta-model design is complete, it is compiled for a specific platform.

Starting from the previous requirements, we have devised a process to build meta-models that is summarised in Fig. 2. First, a domain expert creates one or more example models using some tool with sketching facilities, such as Visio, PowerPoint or Dia. These examples are transformed into untyped model fragments made of elements and relations (step 1). An engineer can manipulate these fragments and define new ones, and also annotate his particular insight of certain elements in the fragments (step 2). A meta-model is induced from the fragments and their annotations (step 3), and it can be visualized to gather feedback about the effect of the fragments. At this point, there are two ways to evolve the meta-model: by adding new model fragments and updating the meta-model accordingly, or by performing some refactorings from a catalogue on the induced meta-model (step 4). In both cases, a checking procedure detects possible conformance issues between the new meta-model and the existing fragments, reporting potential problems and updating the fragments if possible (step 5). Finally, in step 6, the user selects a platform and purpose of use, and the neutral meta-model (and the fragments) is compiled into an implementation one, following the specific idioms of the target technical space. The compilation rules are customizable, and new compilations can be defined.

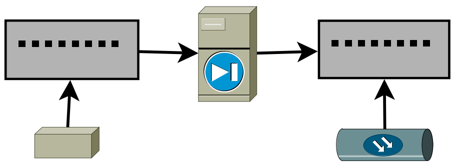
Realizing this approach poses several challenges. First of all, both engineers and non-technical experts need to develop model fragments. Engineers must be provided with a comprehensive set of annotations to specify design intentions. For non-technical experts, fragments are defined by sketches that have to be interpreted, for instance taking advantage of spatial relationships (e.g., containment). Secondly, the induction process is not a batch operation, but it is an interactive process that must take into account both the current version of the meta-model and the previous and new model fragments, detecting conflicts if they arise. Thirdly, a mechanism to let the users supervise the decisions of the induction algorithm has to be defined, as well as a set of meta-model refactorings to enable the resolution of conflicts. Finally, we compile meta-models for specific platforms and uses, which requires studying the requirements of the considered platforms. These issues are discussed in Sections 3, 4 and 5.

3 Definition of Model Fragments

In our approach, users provide model fragments –examples of concrete situations– from which a meta-model is induced. Model fragments can be specified by a domain expert, typically using a drawing tool, or by an engineer, using a more concise syntax that can include annotations to guide the induction process.

As a running example, suppose we need to design a DSML to model simple factories and assist domain experts in modelling networks of machines. The machines receive and produce parts to conveyors, which can themselves be interconnected. Factories can include generators of two kinds of parts: dowels and cylinders. The left part of Fig. 3 shows a model fragment with a particular network as it would be sketched by a domain expert. It includes a machine connected to input and output conveyors, each transporting a different type of part. The right part of the figure shows the same fragment using the textual syntax that the engineer would use. Actually, we have an importer from *Dia* drawings that translates the sketches into textual fragments.

The textual syntax allows the engineer to enrich the fragments with *domain* and *design* annotations to guide the meta-modelling induction process. Domain annotations assign a meaning or feature to certain aspects of the fragment elements. For instance, the annotation *@container* attached to `Conveyor` indicates that, conceptually, conveyors are containers of items while these are being transported (see line 2 in Fig. 3). It is not necessary to repeat the same annotation for all objects of the same kind, but it is enough to annotate one of them. Another example of domain annotation is *@global*, regarding the shareability of elements between different models. For example, a global clock may be used to synchronise all simulation models of a system. If an element is not tagged as *@global*,



```
 1 fragment "conveyorSequence" {
 2   @container
 3   cin: Conveyor {
 4     rel asm = a;
 5   }
 6   cout: Conveyor {}
 7   a: Assembler {
 8     rel outs = cout;
 9   }
10   p1: Dowel {
11     rel conv = cin;
12   }
13   p2: Cylinder {
14     @general
15     rel conv = cout;
16   }
17 }
```

Fig. 3. Model fragment definition: Graphical concrete syntax used by the domain expert (left), and compact textual syntax used by the engineer (right)

it is assumed to be local, i.e., accessible in the scope of the current model only. These annotations may be translated later into meta-modelling design decisions or used to guide the meta-model compilation process.

On their side, *design* annotations refer to meta-modelling decisions that should be reflected in the meta-model generated from the fragments. These decisions can also be given later by refactoring the induced meta-model, but the engineer is given the possibility to define them in advance using annotations. For instance, the *@general* annotation specifies that a certain reference or attribute should be kept as general as possible, i.e., it should be placed as high as possible in the inheritance hierarchy. This may cause the creation of an abstract class in the meta-model, as a parent of all classes owning the reference or attribute. For example, the annotation in Fig. 3 (line 14) will cause the creation of a parent class for dowels and cylinders, defining the common reference `conv`. Other examples of design annotations are *@external*, to indicate cross-references, or *@partial*, to indicate that a class is only partially defined and should be completed with others through merging or inheritance.

Altogether, annotations are a means to record an insight of the engineer at a given point in the running session, and will be used at some point in the future to guide the meta-model induction process.

4 Bottom-Up Meta-Model Construction

Whenever the user enters a new fragment, the meta-model is updated accordingly to consider the new information. The annotations in the fragment are transferred to the meta-model, and this may trigger meta-model refactorings. Any conflicting information within and across fragments, like the assignment of non-compatible types for the same field, is reported to the user and automatically fixed whenever possible. Next, we describe our meta-model induction

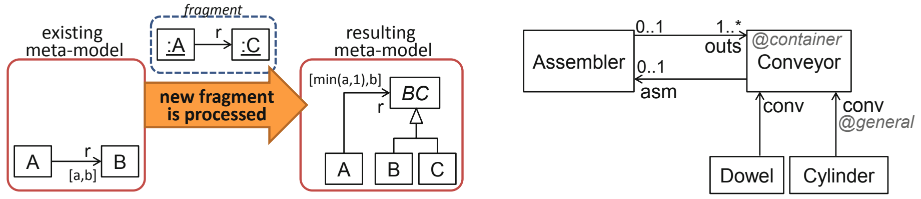


Fig. 4. Processing a reference with different target type in the meta-model and a fragment (left). Meta-model induced from the fragment in Fig. 3 (right).

algorithm, how meta-model refactorings are applied, and the strategy for conflict resolution.

4.1 The Meta-Model Induction Algorithm

Given a fragment, our algorithm proceeds by creating a new meta-class in the meta-model for each object with distinct type. If a meta-class already exists in the meta-model due to the processing of previous fragments, then the meta-class is not newly added. Then, for each slot in any object, a new attribute is created in the object's meta-class, if it does not exist yet. Similarly, for each relation stemming from an object, a relation type is created in its meta-class, if it does not exist. The minimum cardinality of relations is set to 0 by default, or to 1 if all objects of the same kind define the reference. The maximum cardinality can be set to 1 or unbounded. We take the convention of mapping plural reference names to multivalued references, and singular to monovalued ones.

If two relations with the same name point to objects of different type, our algorithm creates an abstract superclass as target of the relation type, with a subclass for the type of each target object. This situation is illustrated to the left of Fig. 4, where the new abstract class `BC` is created as parent of both `B` and `C`. In this example, `BC` would not be created if the `B` meta-class is abstract and the `C` object defines features that are compatible with those in `B`. The minimum cardinality of the relation type `r` is set to `min(a, 1)` because it should accept at least one element (the one provided in the fragment), but the previous minimum cardinality (value `a`) may be zero. The maximum cardinality `b` of the relation is kept. As we will explain in Section 4.3, any automatic design decision made by the algorithm is reported to the user, so that he can change it.

As an example, Fig. 4 shows to the right the meta-model induced from the fragment in Fig. 3. According to the heuristic, the `conv` and `asm` relations (*singular*) were assigned upper bound 1, while `outs` (*plural*) received upper bound *. For the lower bound, it is 0 in `asm` because the fragment contains conveyors not connected to assemblers, but it is 1 in `outs` because all assemblers are connected to some conveyor in the fragment. Additionally, the *@general* and *@container* annotations were copied from the fragment to the meta-model.

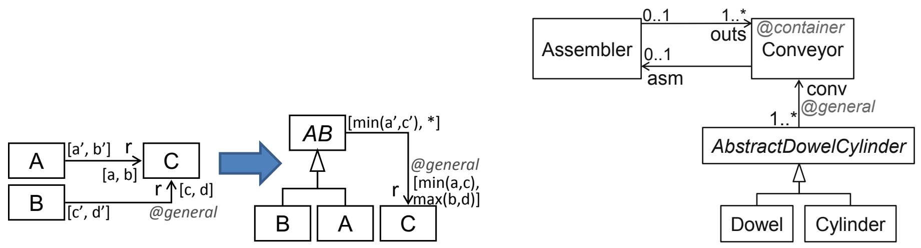


Fig. 5. Scheme of the refactoring triggered by *@general* (left). Result of the refactoring on the meta-model to the right of Fig. 4 (right).

4.2 Refactoring of Meta-Models

The annotations transferred from the fragments to the meta-model may trigger refactorings in it to reflect the annotated intentions. For example, the left of Fig. 5 shows the refactoring triggered by the *@general* annotation, which is similar to the *pull-up* refactoring [10]. It pulls up the annotated attribute or relation as general as possible in an inheritance hierarchy. If the annotated attribute or relation is shared by two classes that are not related through inheritance, then an abstract, parent class is created for them so that the attribute or reference can be pulled up (i.e., Fowler's *extract superclass* refactoring [10] is applied). The target of the pulled relation receives as lower bound the minimum of the lower bounds, and as upper bound the maximum of the upper bounds. The situation is similar for the source, but in this case the upper bound is generalised to * as the pulled relation must merge those from `A` and `B` (i.e., we take * as the upper bound instead of the sum of the upper bounds `b'+d'`).

The right of Fig. 5 shows the result of executing this refactoring to the meta-model in Fig. 4, due to the *@general* annotation in reference `conv`. A new abstract class `AbstractDowelCylinder` is created as parent of both `Dowel` and `Cylinder`, acting as target of the reference.

4.3 Interactivity and Exploration by Supervising Decisions

Our induction process and the triggered refactorings are automated mechanisms. If there are several design alternatives available, then our algorithm takes a decision; therefore, some supervision on behalf of the user may be needed. Our aim is that the environment assists the user in refining the meta-model interactively as it is being built. To this end, our induction algorithm records the decisions taken, and presents possible alternatives to the user in the form of "open issues".

Each open issue presents one or more alternatives, each one of them associated to a refactoring. Whenever an alternative is selected, the corresponding refactoring is applied to the meta-model. This interactive approach enables non-expert users to refine a meta-model by observing the effects of their actions and following suggestions from the environment. The user may apply refactorings manually, through a catalogue of refactorings provided as part of the environment, if his level of expertise allows him to decide how to change the meta-model.

On the other hand, our induction algorithm is conservative as it does not break the conformance of previous fragments when the meta-model needs to be changed to accommodate new fragments; if the algorithm finds a disagreement, then it raises a conflict. However, the resolution of an open issue by means of a refactoring may break the conformance. According to [5], changes in meta-models can be classified into *non-breaking*, *breaking and resolvable*, and *breaking and unresolvable*. Our refactorings automatically update the fragments if a change is non-breaking or resolvable. For unresolvable ones, the user is asked to provide additional information or to discard the no longer conformant fragment.

We have defined three kinds of open issues: *conflict*, *automatic* and *suggestion*, which are briefly explained next.

Conflict. The definition of new fragments may imply the update of the meta-model. For example, the cardinality of existing relations may need to be changed, or new classes may need to be created. If a fragment contains contradictory information, e.g., if the same attribute is assigned incompatible types in different objects, then a conflict arises. For instance, there is a conflict if a conveyor defines an attribute `attr id=‘‘c1’’`, and another conveyor defines `attr id=2`. In this case, our algorithm chooses one of the types (e.g., String) and notifies the conflict and the alternative to the user (e.g., choosing Integer). This open issue must be resolved at some point by the designer. Changing the type of an attribute from Integer to String is an example of breaking and resolvable change (e.g., the conveyor with `id=2` would be automatically changed to `id=‘‘2’’`), while a change from String to Integer is breaking and unresolvable, and requires the intervention of the user. Our algorithm chooses by default an alternative that is non-breaking or, at least, resolvable.

Automatic. These are decisions automatically taken by the induction algorithm when several alternatives exist. For instance, the name of the superclass automatically introduced for `Dowel` and `Cylinder` is built by concatenating the subclasses' names prefixed by "Abstract" (i.e., `AbstractDowelCylinder`). The user is notified about this design decision, and is offered the possibility of changing the superclass' name.

Suggestion. Some meta-model improvements may be possible, like the application of guidelines or meta-model design patterns. These are provided as suggestions which, if accepted, will trigger a certain meta-model refactoring. As an example, if a reference is multivalued but its name is singular, our engine will suggest the user to give it a plural name. So far, we support simple suggestions, but our aim is to define and implement a catalogue of meta-modelling good practices that help non-expert users improve the quality of their meta-models.

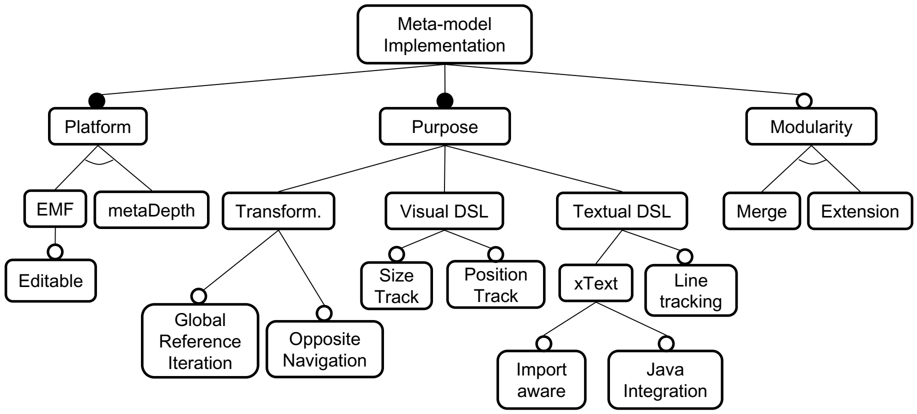


Fig. 6. Feature model for meta-model compilation (excerpt)

5 Meta-Model Compilation for Specific Platforms

The bottom-up meta-modelling process results in a conceptual meta-model that still needs to be *implemented* in a particular platform (e.g., EMF, MetaDepth), and tweaked for a particular purpose. For example, in EMF, an extra *root* class is frequently added (e.g., if the models will be edited with the default tree editor), making heavy use of composition associations. If we aim at creating a model-to-model transformation, then we often implement references as bidirectional associations to ease the definition of navigation expressions. Therefore, we propose to define a number of transformations from such a neutral, conceptual meta-model into implementation ones for specific platforms and purposes.

Fig. 6 shows a feature model that gathers some compilation variants from our neutral meta-model. We currently support two platforms: EMF and MetaDepth. For each one of them, one can select different profiles or purposes: transformation, visual language and textual language definition. Each platform and profile has different options, which can help to fine-tune the compilation. The implementation of the modularity mechanism for meta-models is also subject of two variants: package merge or cross-references between meta-models.

Next, we enumerate the different compilations that we support up to now.

– **EMF platform**. This compilation produces an *ecore* meta-model. Optionally, by setting the *Editable* flag equal to *true*, the compilation generates a *root* class and composition associations to allow any class to be reachable from the root class via composition associations. Each reference from a class annotated with *@container* to a class annotated as *@containee* is converted to a composition. Finally, EMF uses references instead of full-fledged associations, and references can only have cardinalities at the target end. For this reason, this compilation generates two opposite references for those relations in the neutral meta-model with cardinality different from * in the source.

– **MetaDepth platform**. This compilation produces a MetaDepth meta-model, which takes advantage of some special features of metaDepth, like *Edges* to model bidirectional associations and associative classes. In contrast to EMF, MetaDepth does not support composition, therefore the compilation generates OCL constraints for those references between *@container* and *@containee* objects.
– **Transformation profile**. In this profile, we can configure two aspects to optimize navigation expressions. By selecting *Opposite Navigation* all relations become bidirectional, so that writing navigation expressions will be easier in languages making use of query expressions (like QVT). The *Global Reference Iteration* option should be selected when we foresee having to iterate over references in a global scope. In this case, an intermediate class is generated to permit the iteration. If MetaDepth is selected as target platform, this option generates an *Edge* instead.
– **Textual language profile**. In xText, there is the convention of using a feature called "name" to allow cross-references to objects. Thus, any class that is target of a non-containment reference must include an attribute "name", otherwise it is added by the compilation. Additionally, xText offers the possibility to automatically provide import facilities for textual files as well as to integrate a DSML with Java types. This requires adding certain classes and attributes to the meta-model, which is automatically done by the compiler if the variants *Import Aware* and *Java Integration* are selected. Finally, some DSMLs may require associating the line/column information to the elements (this is even required in tools like TCS), which is implemented making all classes inherit from a common *LocatedElement* class.
– **Visual language profile**. In this case, we can select whether to include in classes attributes to store the size and position of elements in the canvas.

As an example, Fig. 7 shows to the left the neutral meta-model obtained by the induction process. This meta-model was obtained from the fragment in Fig. 3 and two additional fragments: one specifying a `Generator` connected to a `Conveyor`,

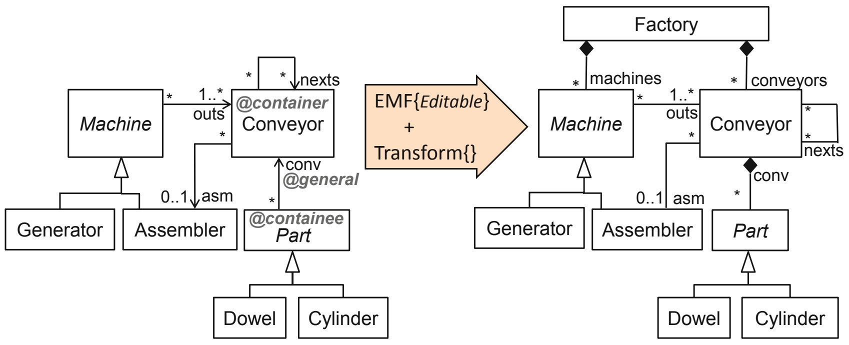
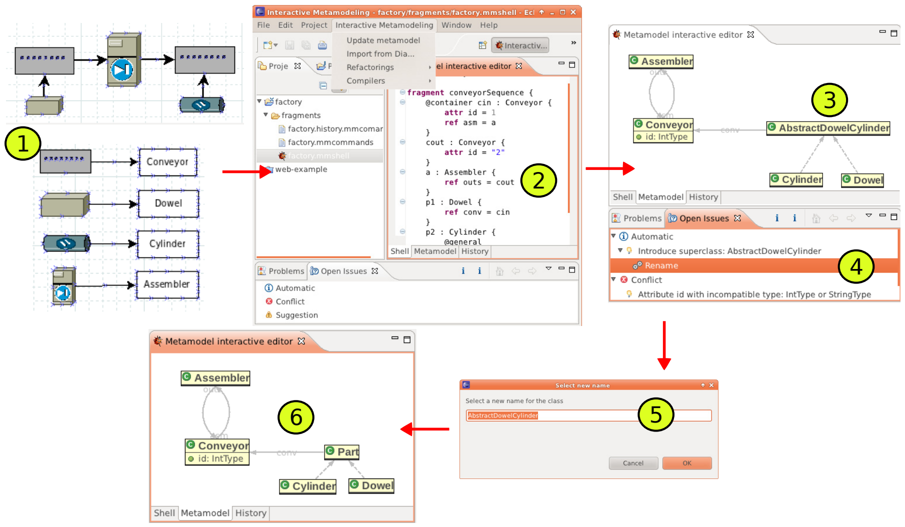


Fig. 7. Compiling to EMF for transformation



```
fragment conveyorSequence {
    @container cin : Conveyor {
        attr id = 1
        ref asm = a
    }
    cout : Conveyor {
        attr id = "2"
    }
    a : Assembler {
        ref outs = cout
    }
    p1 : Dowel {
        ref conv = cin
    }
    p2 : Cylinder {
        @general
```

Automatic
Introduce superclass: AbstractDowelCylinder
Rename
Conflict
Attribute id with incompatible type: IntType or StringType

Fig. 8. Example of interaction with the tool

which produced the `Machine` abstract class, and another one connecting two `Conveyors`, which generated the `nexts` relation. The `AbstractDowelCylinder` class was renamed to `Part` through a *renaming* refactoring. Since none of the new fragments contained `Parts`, the lower bound of the source of `conv` was set to zero by the algorithm. Finally, an additional *@containee* annotation was manually added to `Part`. The figure also shows the compilation of this meta-model into EMF using the transformation profile. For the EMF platform, we chose the generation of a root class, and the reference between the *@container* and *@containee* was compiled into a composition. Additionally, the *transformation* profile makes each reference bidirectional. A wizard asks the user any information needed to complete the compilation, like the name of the root class and association ends. Moreover, all model fragments are compiled using the same options, so that a set of testing models becomes available for free.

6 Tool Support

Realizing our approach requires specialized, integrated tool support that has to go beyond the dominant style of meta-modelling nowadays: top-down and based on a batch processing style. To this end, we have implemented a tool for Eclipse that gives interactivity to our approach[3]. Next, we describe the elements of our tool by going through an interaction example that is shown in Fig. 8.

1. Sketching fragments. If a domain expert is involved in the meta-modelling process, he may first sketch fragments that represent scenarios in the domain.

[3] Available at `http://sanchezcuadrado.es/projects/interactive-metamodeling`

We have implemented an import facility that takes a diagram sketched with the Dia tool and generates a model fragment ready to be evaluated. This facility implements simple heuristics to determine sensible names for objects and references, and takes advantage of the visual containment relationships between elements to generate equivalent annotations.

A drawing tool such as Dia (and others like Visio) offers a wide variety of symbols, organized in categories, that can be used to sketch fragments. However, we do not expect that a non-technical user respects the semantic meaning of the symbols (in the figure, the symbol to depict a conveyor is actually a representation of a network hub). Instead, a symbol is used if the pictogram resembles what the user wants to convey. Nonetheless, it is important that each meta-model element is assigned a meaningful name within the domain. For this purpose, symbols can be attached a legend with their name.

2. Editing fragments. Engineers can create fragments by using the textual notation introduced in Section 3. Actually, the sketched fragments are translated into this second textual notation for their subsequent edition and manipulation, e.g., to add annotations or to refine the name of the types. In the example, the engineer has added the `@container` and `@general` annotations, as well as a new attribute `id` to `Conveyors`. Textual fragments are edited by using a text editor (built with xText) with syntax highlighting, error reporting, templates and autocompletion. A key binding and the "Update metamodel" menu option permits processing the current fragment.

3. Visualizing the meta-model. Processing a fragment induces a new version of the meta-model. However, there is no editor to modify the meta-model. Instead, the meta-model is visualized so that the user can check its state and evolve it in three ways: processing new fragments, applying manual refactorings or addressing open issues. Implementation-wise, the current version of our tool uses the Zest framework to render and layout meta-models.

4. Addressing open issues. The interactive and exploratory nature of our approach is realised through the *open issues view*. This view gives the user information about conflicts as well as suggestions of possible refactorings. Selecting an issue will show possible fixes that in turn will launch a refactoring. In the figure, two issues are reported: (1) the introduction of a new superclass, and (2) a conflict related to incompatible types for the `id` attribute. For the first issue, our system proposes a rename action (if needed). Additionally, every command and updated fragment is recorded and can be queried in the history tab. Sessions are persistent, that is, the state of the session is stored after each evaluation so that it can be interrupted and resumed later.

5. Applying a refactoring. Selecting a proposal from the open issues view will raise a refactoring. The refactoring may require the user to provide some information, as is the case of the figure, where the name of the class is asked (in this case `Part` is given). Afterwards, the visualization of the meta-model is automatically updated (step 6).

These steps are performed iteratively until all concepts of the domain are represented in the meta-model. At this point, the meta-model can be compiled for some particular purpose, as explained in Section 5. The user then selects the purpose (for instance, creating a textual DSML with xText), and the environment automatically selects dependent features (e.g., xText implies EMF) and shows a *wizard* that asks the user for optional features. Finally, the meta-model is generated in the selected format, for instance Ecore.

7 Related Work

There are some works dealing with the inference of meta-models from models. The MARS system [12] enables the recovery of meta-models from repositories of models using grammar inference. The objective is being able to use a set of models after migrating or losing their meta-model. Actually, the induction process can be seen as a form of structural learning [15]. In contrast, our purpose is enabling the *interactive* construction of meta-models, also by domain experts.

There are a few works using test-driven development (TDD) to build meta-models iteratively. For instance, in [7], the authors attach test cases to the meta-classes in a meta-model. Test cases are executable models written in PHP, and perform some kind of transformation like code generation. If a test case shows that a meta-model is inadequate, this must be manually modified. Similarly, in [20], the authors combine specifications and tests to guide the construction of Eiffel meta-models. Specifications are given as Eiffel contracts, whereas tests are written using the acceptance test framework for Eiffel. Another example is [22], which supports the specification of positive and negative example models from which test models for meta-model testing are generated. In our case, the meta-model is automatically induced from model fragments, and there is a greater level of interactivity. Moreover, meta-models are updated through refactorings, which simplifies their evolution and the propagation of changes to model fragments (i.e., side effects). A catalogue of meta-model refactorings, although not directly related to TDD of meta-models, is available in [18]. We provide support for many of them. Finally, the idea of testing meta-models by creating test cases is orthogonal to our approach, and could be integrated in our environment.

Techniques to build MDE artefacts "by example" have emerged in the last years, but it is still novel for meta-models. In the position paper [4], the authors identify some challenges to define DSMLs by demonstration. They discuss the usefulness to bridge informal drawing tools with modelling environments, as the former are the working tools of domain experts. They also recognise the difficulty for experts to manually build meta-models, and suggest an iterative process. Recently, the authors have realised their ideas in a framework where domain experts can provide model examples using a concrete syntax, from which a meta-model describing their abstract syntax is inferred [3]. While their proposal is similar to ours, we also stress that meta-models may be different depending on the target platform and usage. Hence, we support the automated induction of a *neutral* meta-model, its refactoring and different compilations into *implementation* meta-models, guided through annotations and selection of configurations.

In our approach, newly introduced fragments may raise conflicts if the fragments contain contradictory information. Some application domains where the resolution of conflicts has been extensively studied are model merging [17], change propagation in software systems [9] and distributed development [6]. It is up to future work to identify how the conflicts that may arise when evolving a meta-model relate to these previous works.

Another line of related work concerns the expressiveness of model fragments. While one could simply use object diagrams, in [16], the authors extend object diagrams with modalities to declare positive and negative model fragments and invariants (i.e., fragments that should occur in every valid diagram). Their goal is to check the consistency of a set of object diagrams, and for that purpose they use Alloy. In our case, the goal is different as we use fragments to automatically induce a meta-model. While we consider negative fragments, they are not yet taken into account by the induction algorithm.

A way to simplify and make the development of meta-models systematic is through design patterns. In [2], some design patterns for meta-models are proposed, while in [23], the requirements for meta-models are represented as use case diagrams and the meta-models are evolved by applying patterns. We plan to integrate patterns in our approach to guide the induction phase and refactor meta-models towards patterns. Integrating end-users in the meta-model construction process has also been regarded as a means to improve the quality of the resulting meta-model. In [11], the authors propose a collaborative approach to meta-model construction which involves both domain and technical experts. The approach is supported by a DSL to represent the collaborations among stakeholders (change proposals, solution proposals or comments) while the meta-model is being developed.

Finally, our meta-model refactorings and subsequent propagation to the model fragments can be seen as a simplified scenario of meta-model/model evolution [5].

8 Conclusions and Future Work

In this paper, we have presented a novel approach to the development of meta-models to make MDE more accessible to non-experts. For this purpose, we have proposed a bottom-up approach where a meta-model is induced from model fragments, which may be specified using informal sketching tools like Dia. A specialized textual notation is also provided for advanced users, who can annotate the fragments to guide the automatic induction of the meta-model. The process is iterative, as fragments are added incrementally, causing updates in the meta-model, which can be refactored in the process. Finally, the meta-model can be compiled for specific platforms and usage purposes.

Even though we allow the specification of negative fragments, these are not currently used to induce the meta-model, which is left for future work. We would like to perform an empirical evaluation of the approach with our industrial partners. We also plan to improve the tool support. One direction is to enhance collaboration by building a web application where domain experts can sketch fragments that are automatically integrated in the environment for their refinement by an engineer. Another goal is to automatically build a visual modelling environment out of the sketched fragments. The integration of different implementation meta-models compiled from the same neutral meta-model, e.g., to support different syntaxes for a DSML, is also future work.

Acknowledgements. This work was funded by the Spanish Ministry of Economy and Competitivity (project "Go Lite" TIN2011-24139) and the R&D programme of the Madrid Region (project "e-Madrid" S2009/TIC-1650).

References

1. Baldwin, C.Y., Clark, K.B.: Design Rules: The Power of Modularity, vol. 1. The MIT Press (2000)
2. Cho, H., Gray, J.: Design patterns for metamodels. In: DSM 2011 (2011)
3. Cho, H., Gray, J., Syriani, E.: Creating visual domain-specific modeling languages from end-user demonstration. In: MiSE 2012 (2012)
4. Cho, H., Sun, Y., Gray, J., White, J.: Key challenges for modeling language creation by demonstration. In: ICSE 2011 Workshop on Flexible Modeling Tools (2011)
5. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: EDOC 2008, pp. 222–231 (2008)
6. Cicchetti, A., Ruscio, D.D., Pierantonio, A.: Managing Model Conflicts in Distributed Development. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 311–325. Springer, Heidelberg (2008)
7. Cicchetti, A., Ruscio, D.D., Pierantonio, A., Kolovos, D.: A test-driven approach for metamodel development. In: Emerging Technologies for the Evolution and Maintenance of Software Models, pp. 319–342. IGI Global (2012)
8. de Lara, J., Guerra, E.: Deep Meta-modelling with MetaDepth. In: Vitek, J. (ed.) TOOLS 2010. LNCS, vol. 6141, pp. 1–20. Springer, Heidelberg (2010)
9. Egyed, A.: Automatically detecting and tracking inconsistencies in software design models. IEEE TSE 37(2), 188–204 (2011)
10. Fowler, M.: Refactoring. Improving the Design of Existing Code. Addison-Wesley (1999)
11. Izquierdo, J.L.C., Cabot, J.: Community-driven language development. In: MiSE 2012 (2012)
12. Javed, F., Mernik, M., Gray, J., Bryant, B.R.: MARS: A metamodel recovery system using grammar inference. Inf. & Sof. Technology 50(9-10), 948–968 (2008)
13. Karsai, G., Krahn, H., Pinkernell, C., Rumpe, B., Schneider, M., Völkel, S.: Design guidelines for domain specific languages. In: DSM 2009, pp. 7–13 (2009)
14. Kolovos, D.S., Rose, L.M., Abid, S.B., Paige, R.F., Polack, F.A.C., Botterweck, G.: Taming EMF and GMF Using Model Transformation. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 211–225. Springer, Heidelberg (2010)
15. Liquiere, M., Sallantin, J.: Structural machine learning with galois lattice and graphs. In: ICML 1998, pp. 305–313. Morgan Kaufmann (1998)

16. Maoz, S., Ringert, J.O., Rumpe, B.: Modal Object Diagrams. In: Mezini, M. (ed.) ECOOP 2011. LNCS, vol. 6813, pp. 281–305. Springer, Heidelberg (2011)
17. Mens, T.: A state-of-the-art survey on software merging. IEEE TSE 28(5), 449–462 (2002)
18. Metamodel refactorings, http://www.metamodelrefactoring.org/
19. Nierstrasz, O.: Ten things I hate about object-oriented programming. Journal of Object Technology 9(5) (2010)
20. Paige, R.F., Brooke, P.J., Ostroff, J.S.: Specification-driven development of an executable metamodel in Eiffel. In: WISME 2004 (2004)
21. Perera, R.: First-Order Interactive Programming. In: Carro, M., Peña, R. (eds.) PADL 2010. LNCS, vol. 5937, pp. 186–200. Springer, Heidelberg (2010)
22. Sadilek, D.A., Weißleder, S.: Towards automated testing of abstract syntax specifications of domain-specific modeling languages. In: CEUR Workshop Proceedings, CEUR-WS.org, vol. 324, pp. 21–29 (2008)
23. Schäfer, C., Kuhn, T., Trapp, M.: A pattern-based approach to DSL development. In: DSM 2011, pp. 39–46 (2011)
24. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley Professional (2008)

FacadeMetamodel: Masking UML

Florian Noyrit[1], Sébastien Gérard[1], and Bran Selic[2]

[1] CEA, LIST, Laboratory of Model Driven Engineering for Embedded Systems, Point Courrier 174, Gif-sur-Yvette, 91191, France
{florian.noyrit,sebastien.gerard}@cea.fr
[2] Malina Software Corp. Nepean, Ontario, Canada
selic@acm.org

Abstract. UML profiling is pragmatic choice that lets language designers define a Domain-Specific Modeling Language (DSML) by tuning UML to meet specific domain. An alternative approach is to define a pure-DSML. Each approach has its own benefits and drawbacks. We propose an approach and a tool that helps get the best from both approaches; maximizing reuse while retaining a focused and adapted DSML. We guide the language designer in the definition of a metamodel based on one or more UML profiles. Language designers then recast UML so that only what they need will appear in this metamodel. From that, the tool automatically generates the pure-DSML and the transformations between it and UML. However, the new pure-DSML is only a facade; models can be manipulated using the pure-DSML abstract syntax but they are actually stored in fully-compliant UML abstract syntax and therefore remain compatible with UML tools.

1 Introduction

At the present time, there seem to be two main approaches to defining a Domain-Specific Modeling Language (DSML): either by defining a UML profile [1] or by defining a brand new DSML. The UML profile mechanism was motivated by the desire to *reuse* the effort and language design expertise invested in the development of UML as well as to exploit the important infrastructure support built around it, such as tooling, training, and documentation. Indeed, UML provides a rich set of constructs with their notation and with the recent introduction of the fUML specification [2], well-defined semantics. This provides a solid base for a broad spectrum of different DSMLs.

An alternative is to design a DSML from scratch. In this paper we shall call such a language a *pure-DSML*. This has the obvious benefit of freeing language designers from constrains imposed by UML allowing the most direct and concise expression of domain concepts and phenomena.

Each approach has benefits and drawbacks so that there is no general rule that universally places one above the other. Relying on standards can lead to major cost savings. On the other hand, custom solutions can lead to higher productivity and better quality.

In this paper we propose an approach and a tool that combine the advantages of both alternatives. It was motivated from the observation that, *in our experience with both approaches to language design, pure-DSML metamodels are often just subsets of the UML metamodel with just a few additional metaclasses and some terminological differences*. Our approach generates what appears as a pure-DSML to the modeler, but which is actually just kind of "facade" behind which sits standard UML, thereby allowing full reuse of the UML infrastructure.

We present the benefits and drawbacks of both approaches to design DSMLs in section 2. This helps us to highlight the limitations that we need to overcome to get the best features of each alternative. Then, in section 3, we introduce our approach and show how these limitations are addressed. In section 4 we briefly give details of an implementation of the approach. In section 5 we demonstrate how a facade language masks out the UML in practice and thereby lets users think in terms of domain concepts instead of UML. Section 6 presents related work. Finally, we summarize the paper together with some discussions in section 7.

2 Approaches to the Design of DSMLs

Working directly with problem domain concepts rather than computer technology concepts is one of the foundations of model-based engineering (MBE). This is the premise behind DSMLs. This coincides nicely with the vision specified in the ISO/IEC/IEEE 42010 standard [3] which suggests that each stakeholder needs dedicated viewpoints to address his or her concerns. Such viewpoints should use syntactical forms that express those concerns in the most direct and intuitive form. It is the responsibility of language designers and methodologists to provide DSMLs that support these viewpoints. And, it is the responsibility of tool designers to provide appropriate user interfaces to corresponding language authoring tools.

Clearly, a key objective is to align the DSML constructs with the domain concepts. The most common approach to this is to first design a *domain model*, which is a concrete artifact intended to capture the ontology of the target domain. Such a model should be completely free of any language implementation details. Each concept in the domain model is then described, most often using informal natural language. This model then serves as a guide for implementing a concrete DSML.

2.1 The Pure-DSML Approach

In this approach, a common way to implement a DSML is to design a *metamodel* based on the domain model but supplemented with constraints and usually refactored somewhat due to various implementation concerns. This defines the *abstract syntax* and *terminology* of the modeling language. The *semantics* of the DSML constructs can be described formally, although it is much more common to describe them informally with a textual description in some natural language. In practice, language semantics specifications are often inadequate leaving too much room for varied interpretations. In addition, a concrete syntax, also called *notation*, is provided for the human user to author models with the DSML.

Last but not least, the tooling infrastructure for the DSML must also be defined and implemented. Even though important research and development has been done in recent years to provide tools that help in the implementation of DSMLs, it remains a time-consuming task that requires highly specialized expertise. This is compounded if the chosen notation is graphical.

If done with care, this approach leads to a relatively compact DSML and tools infrastructure that are nicely aligned with the domain. Both problems and solutions can be expressed directly and succinctly using such a language. However, over the long-term, maintaining pure-DSMLs and custom tooling can become very burdensome and expensive, drawing both time and resources away from core business concerns. This includes the cost of supporting and evolving tools as well as developing and maintaining training materials and providing language courses. In addition, developing a system using multiple viewpoint languages based on independently-designed DSMLs can lead to complex design and tool integration problems.

2.2 Approaches Based on Reusing Existing Metamodels

An alternative to design pure-DSMLs from scratch is to reuse the time, effort, and expertise that went into an existing language such as UML. This typically involves a number of different refactorings of the base language metamodel to accommodate domain-specific concerns. The following types of refactorings may be needed:

- *Extension* is the addition of new metaclasses for capturing domain-specific concepts not present in the base language.
- *Pruning* is used to eliminate those parts of the base language that are not used in the DSML. We adopt a broad definition of pruning: anything that reduces the expressivity. It usually involves removing unnecessary concepts, adding constraints, or merging concepts.
- *Terminology adaptation* is used to alias base language concepts, their properties and operations to suit the domain.

This type of approach has the benefits of reuse but, if done in an *ad hoc* manner, the full benefits may not be gained. The profiling mechanism of UML offers a structured method for metamodel refactoring.

2.3 The UML Profile Mechanism and Its Limitations

Being a general-purpose modeling language, UML provides a rich source of general modeling concepts and a well-vetted metamodel that reflects years of experience in modeling language design.

The profile mechanism provides both extension and pruning capabilities. However, the profiling mechanism limits these capabilities since profiles must be compatible with standard UML. This ensures that profile-based models can be manipulated correctly by UML-compliant tools. One primary constraint is that the standard UML abstract syntax cannot be reconfigured and that standard UML semantics cannot be contradicted. Profiles can only be more constraining and, above all, must be consistent with UML. In addition, the profiling mechanism does not provide an aliasing feature that would allow renaming of standard UML constructs to fit the domain terminology.

Extension through Stereotypes. In a UML profile, extension is done using *stereotypes*. A stereotype is defined as an *extension* of a UML *base* metaclass (the metaclass that it specializes) or a refinement of an existing stereotype. Although the notation for the Extension concept in UML is reminiscent of standard MOF Generalization (suggesting conceptual similarity), its semantics are quite different. Specifically, a stereotype instance cannot be instantiated independently of an instance of its base metaclass. These unique semantics mean that stereotypes are really just metadata attachments that serve to annotate metaclass instances.

A stereotype can be defined as being mandatory (i.e., the "*isRequired*" property of the extension set to true). This means that all instances of the base metaclasses in the model must have their own attached stereotype of this type. For example, in case of a model based on a Java profile, all instances of UML Class in the model must be stereotyped with a JavaClass stereotype, which adds constraints to the base concept consistent with the Java language definition (e.g., no multiple inheritances).

However, it is possible to interpret the semantics of stereotype extension in a number of subtly different ways. One may see it:

- As a simple association (which is how it is implemented),
- As a kind of generalization, or
- As merely a way of adding properties to the base metaclass.

Some profiles even create empty stereotypes, using extension merely to achieve aliasing. Consequently, within a single profile different stereotypes may rely on different interpretations. For instance, in SysML [4], the Block stereotype is intended to be used differently than the RequirementRelated stereotype. The former is an "*in-place*" stereotype since the Block concept fully displaces the Class concept in SysML, whereas the latter is an "*annotating*" stereotype "used to add properties to those elements that are related to requirements via the various dependencies described" (16.3.2.4 of [4]). For Block, the extension is a kind of generalization while for RequirementRelated the extension is a simple association.

Consequently, to ensure accurate representation of the language designer's intentions, the interpretation to give to each extension must be defined explicitly.

Pruning the UML Metamodel. As an alternative to a pure pruning feature, UML provides a metaclass filtering function known as "*isStrict*", which applies at the time that a profile is applied to a UML model. The filtering rules allow selecting which metaclasses are visible in the profiled model (details of the filtering rules are defined in clauses 18.3.7 and 18.3.8 of [1]). Unfortunately, filtering can only be applied at the metaclass level so that individual metaclasses properties and operations cannot be filtered out.

This filtering feature is rarely used in practice to define a profile or to define a corresponding DSML interface for tools. Consequently, instead of interpreting the profiles definitions, UML developers and tool vendors often create manually specific "filtered" user interfaces for each profile to hide those parts of UML that are not needed.

Furthermore, this type of filtering still leaves the full UML metamodel unchanged merely hiding unnecessary metaclasses from view of the modeler: filtering applies at the M1 level not at M2. The full complexity of the UML is still seen by tools, such as: model-to-model (M2M) tools (e.g., QVTo [5][6] or ATL [7]), model-to-text (M2T) tools (e.g., Acceleo [8]), or various model analysis tools. These tools and their users must cope with the full UML abstract syntax as well as the extensions defined in the profile. This is further complicated by the fact that navigating from a stereotype to its base metaclass instance and vice versa is not straightforward.

Additional constraints, such as OCL [9] constraints, can be used to further constrain elements of the UML metamodel. For instance, one could define an OCL constraint that all instances of Class must have at least one Operation (regardless of whether the Class is extended by a stereotype or not). However, these rules are not provided to hide the complexity; they are only used to validate (most of the time a posteriori) that a model conforms to a more constraining syntax.

Implicit Rules for Stereotype Applications. Multiple stereotypes can be applied on the same UML element. However, most of the time, this is not something profile designers expect. For instance both Requirement and Block stereotypes in SysML extend the UML Class concept. However nothing in the profile forbids applying both stereotypes to the same instance of Class, even though this does not make sense. We refer to this as the issue of *stereotype application compatibilities*.

Another potential issue arises whenever a stereotype extends a non-leaf metaclass, since this creates an opening for inappropriate stereotyping. This possibility is often overlooked by language designers. For instance, the Requirement stereotype, which extends the Class metaclass, can also be applied to instances of a StateMachine, which happens to be a kind of Class in UML. Of course, stereotyping a StateMachine as a Requirement is not meaningful and should not be allowed. We call the *scope of applicability* of a stereotype the set of metaclasses to which it can be applied.

Generate Specific Viewpoints. A UML profile in addition to defining the abstract syntax, concrete syntax and semantic adaptations, may also implicitly define the viewpoints of a language. For instance, SysML provides a requirement modeling viewpoint, a system modeling viewpoint, a parametrics viewpoint, etc. Such viewpoints are often defined in the form of dedicated sub-profiles. However, this does not state explicitly which UML concepts are intended to be used in which viewpoint – leaving it up to tool designers to figure out.

One way to make those viewpoints more explicit would be to use the metaclass filtering feature for each sub-profile. To the best of our knowledge, however, no standard profile used that design pattern. Also, as stated before, filtering does not provide the necessary granularity nor does it hide complexity from other tools.

In addition, specific user interfaces are designed most of the time for a single profile. In cases where a user needs a viewpoint that uses multiple profiles, there is usually no adequate tool support.

2.4 Providing Pure-DSMLs through UML Profiles

Despite these limitations of profiles, we feel that it is a practical choice because of all its reuse and other benefits cited earlier. Therefore, the key idea behind our FacadeMetamodel approach—a kind of "recasting" of UML—is that users can be "doing UML without being aware of it", retaining thus the benefits of UML without suffering the drawbacks of inflexibility and complexity.

However, it is not always possible to tell what the language designer's intentions were, by simply inspecting a profile definition. At best, the textual specifications identify how to use the different stereotypes and how they should be interpreted. While humans can sometimes infer the most likely intent even if it is underspecified or understand natural language specifications, computers cannot. Because profile definitions lack many explicit specifications, specific tool support for each viewpoint of profile has to be manually coded. It is, therefore, necessary to make the intentions of profile designers explicit and, more importantly, computer interpretable. Specifically, to obtain the advantages of pure-DSMLs, language designers must explicitly capture their intent in terms of: how stereotype extensions are to be interpreted, the scope of applicability of stereotypes, their application compatibilities, aliasing, etc. In the following section, we explain how our approach provides such capabilities.

3 The FacadeMetamodel Approach

This approach is based on defining a *facade definition model* for each viewpoint in the profile(s). These definition models are then interpreted to generate corresponding facade metamodel and UML custom diagrams for tools. To describe this approach, we will make use of two sample UML profiles (Fig. 1), which were artificially designed to cover most of the common patterns used in defining UML profiles. The full process for creating a facade metamodel is a multi-step procedure described below.

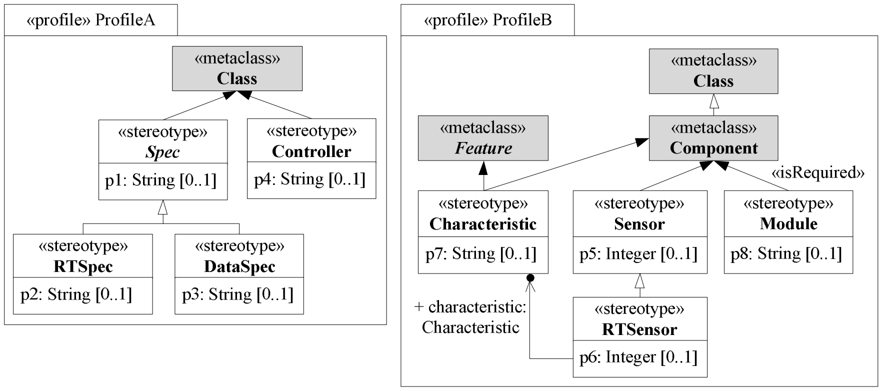


Fig. 1. Definition of sample UML profiles ProfileA and ProfileB

3.1 Step 1: Qualify the Intent of the Extensions

As noted earlier, the UML Extension concept, which relates a stereotype to its base metaclass, can have multiple interpretations serving different ends. Therefore, in our approach it is first necessary to clarify the intent of this relationship. This has two objectives: (a) to disambiguate the relationship between a stereotype and its base metaclass and (b) to clarify the scope of applicability of the stereotype. Metaclasses in the facade metamodel are called *facade metaclasses*, properties (either attributes or references) are called *facade properties* and operations are called *facade operations*.

Our approach recognizes two different interpretations of extensions:

- *Tag*. In this interpretation, the extension acts as an ordinary association; that is, it attaches additional information to instances of the base class (left hand side of Fig. 2). It matches "annotating" stereotypes. In the facade metamodel, each desired stereotype is represented by a new facade metaclass. A bidirectional association is added to represent the relation between the facade metaclass that represents the base metaclass and the facade metaclass that represents the stereotype. It is defined as composite association with the facade metaclass that represents the base metaclass as the owner. This new association facilitates navigation between the base metaclass and the stereotype in either direction. In this interpretation the scope of applicability of the stereotype is necessarily the base metaclass and all its subclasses.
- *Sub-Metaclass*. In this interpretation the Extension acts as a generalization (right hand side of Fig. 2). It matches "in-place" stereotypes. The language designer can explicitly specify the scope of applicability. For example, if the Extension has Class as the base metaclass, it normally means that the stereotype is automatically applicable to all subclasses of Class (including StateMachine, Node, etc.). But, this may not be desirable, so we require the applicability of the stereotype to be explicitly defined for each subclass of the base class. A new facade metaclass and a new generalization are added to the facade metamodel only for applicable subclasses.

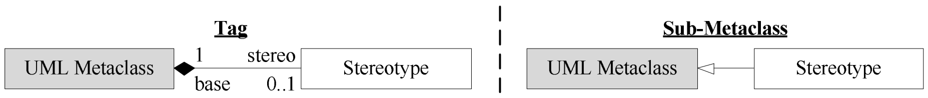


Fig. 2. Pattern of the Tag (left) and Sub-metaclass (right) interpretations of extensions

These qualifications apply transitively to all sub-stereotypes of the stereotype rooted at an Extension. For instance, defining the Extension between Spec and Class in ProfileA (Fig. 1) as a *Tag* means that RTSpec and DataSpec will also be interpreted as *Tags*. Also, different Extensions from a stereotype can have different interpretations. For instance, the two Extensions of Characteristic in ProfileB (Fig. 1) could have different interpretations. If they are all defined to have a *Tag* interpretation, only one facade metaclass is created. Otherwise, separate facade metaclasses will be created to represent the stereotype.

If the Extension is defined as *isRequired* and the interpretation is set to *Sub-Metaclass,* the facade metaclass that represents the base metaclass in the facade metamodel is forced to be abstract. If the interpretation is *Tag,* however, multiplicities for the association ends are set according to the multiplicities of the extensions' member ends. This results in a 1..1-1..1 association, which forces the need for a third interpretation:

– *Merge*. In this case, the language designer probably simply intends to merge the properties of the stereotype with those of the base metaclass. We haven't implemented this interpretation in our tool yet.

On our case study, the facade definition model could contains the definitions shown in Table 1 (abstract stereotypes are in italics, stereotypes with required extension are in bold and ↳ is used to denote sub-stereotypes).

Table 1. Clarification of the extensions in the example in Fig. 1

Stereotype	Interpretation	Scope of applicability
Spec, ↳RTSpec, ↳DataSpec	Tag	-
Controller	Sub-Metaclass	Activity, AssociationClass, Class, Component, Device, Interaction, Node, ExecutionEnvironment, StateMachine, FunctionBehavior, OpaqueBehavior, ProtocolStateMachine, Stereotype
Sensor, ↳RTSensor	Sub-Metaclass	Component
Characteristic (Component)	Sub-Metaclass	Component
Characteristic (Feature)	Sub-Metaclass	Connector, ExtensionEnd, Operation Port, Property, Reception
Module	Sub-Metaclass	Component

Note that Controller, Sensor, RTSensor and Characteristic (Component) do have Component in their scope. However they cannot be applied alone because they must be applied in combination with Module, which is required.

3.2 Step 2: Define Stereotype Application Compatibilities

In cases when the interpretation of an extension is *Sub-Metaclass*, compatibilities between stereotype applications can be defined. For instance the application of both Characteristic and Sensor to a Component must be explicitly declared as compatible if the language designer wants a new facade metaclass to represent this combination. By default, stereotypes interpreted as *Sub-Metaclass* are incompatible, that is, they cannot be applied to the same metaclass instance. For each compatible combination, a new facade metaclass and corresponding new generalizations are added to the facade metamodel.

For example, on our case study, we might specify the compatibilities presented in Table 2 in the facade definition model.

Table 2. Clarification of stereotype application compatibilities in the example in Fig. 1

Scope	Stereotype application compatibilities
Component	**Module**, (Sensor, ↳RTSensor)
	Module, Characteristic
	Module, Controller

Note that stereotype application compatibilities conform to the stereotype hierarchies. For instance, defining compatibility with Sensor implies compatibility with RTSensor.

3.3 Step 3: Generate a Preliminary Facade Metamodel

Once Extension qualifications, scope of applicability, and stereotype compatibilities have been specified, a tool can automatically generate a preliminary facade metamodel. We present here the different patterns used to generate it. Fig. 3 shows a fragment of the generated metamodel for our example.

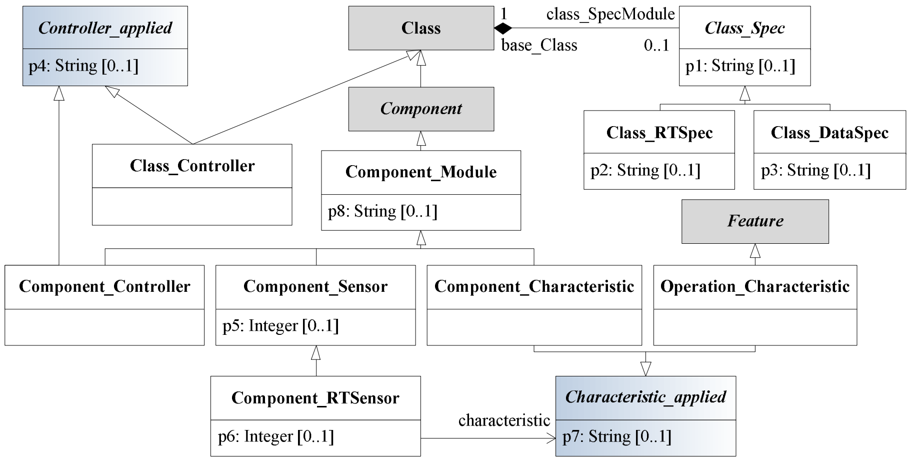


Fig. 3. Representation of an extract from the generated preliminary facade metamodel

The preliminary facade metamodel consists of facade metaclasses corresponding to the UML metaclasses supplemented with facade metaclasses that represent stereotype applications to UML metaclasses. In this preliminary facade metamodel, the name of facade metaclasses that represent stereotype applications are prefixed with the name of the UML metaclass to which they apply.

- *Tags*. The Tag stereotypes (Class_Spec, Class_RTSpec and Class_DataSpec) are related to the base facade metaclass (Class) with a composite bidirectional association to facilitate navigation.
- *Sub-Metaclass*. Metaclasses that represent a stereotype application specialize directly the UML metaclass unless there is a required stereotype to apply on this UML metaclass. For example, because Module is required, Component becomes abstract and all stereotypes that were defined as compatible with Module become subclasses of Component_Module. In our prototype tool, we allow only one required stereotype to be defined with *Sub-Metaclass* interpretation, because the alternative would result in needlessly complex generation patterns. This restriction will be removed when we implement the *Merge* interpretation.
- If a stereotype is used by multiple facade metaclasses in the facade metamodel, to avoid the creation of multiple facade properties in each facade metaclass that represents a combination of stereotype applications, we add a new abstract facade metaclass in the facade metamodel. This abstract facade metaclass contains common facade properties of the stereotype, while combinations that represent its application specialize from it. For instance, Class_Controller and Component_Controller both specialize Controller_applied. This simple design pattern guarantees consistency of facade properties among facade metaclasses that represent stereotype combinations.

3.4 Step 4: Customize the Preliminary Facade Metamodel

In the next step the language designer customizes the preliminary facade metamodel to better match the domain. Possible customizations include:

- Aliasing (i.e., renaming) facade properties, facade operation and facade metaclasses (NB: the names must be unique).
- Refinement of multiplicities of facade properties. To ensure compatibility with UML, the refined multiplicity ranges must be narrower than the original ones (i.e., higher lower value and/or lower upper value).
- A facade metaclass can be made abstract but with the obvious constraint that there must exist at least one concrete facade metaclass among its subclasses.
- Facade metaclass, facade operations, and facade properties can be pruned. For this, we adopt the solution proposed in [10].

On our example we might decide to apply the following aliasing: Component_Module → Module, Class_Controller → Controller, Component_Controller → ControllerModule.

3.5 Step 5: Generate the Facade Metamodel

Generation of the facade metamodel consists in generating an ordinary metamodel from the customized preliminary facade metamodel and storing, in the facade definition model, the mapping that exists between the actual facade metamodel and, the UML metamodel and the profiles. Part of the metamodel of the facade definition is given in Fig. 4.

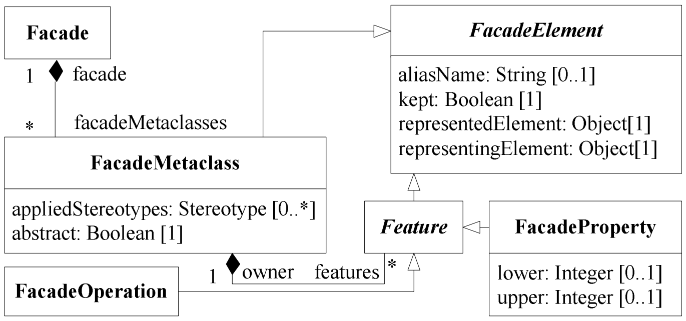


Fig. 4. Fragment of the metamodel of the facade definition

The UML or profile elements are referenced through *representedElement* while facade metamodel elements are referenced via *representingElement*. A facade metaclass can be the representation of a combination of stereotype applications. This information is stored in *appliedStereotypes*. The a*bstract* property is used to make a facade metaclass abstract. *AliasName* stores the alias to apply and *kept* is used for the pruning feature. The *lower* and *upper* properties are used to override multiplicities.

The facade definition model contains now all the information to transform models that conform to the facade metamodel to UML models and vice versa. These transformations could be used to override the parsing and serializing functions that are used to load and save models. However, instead of generating specific transformations for each facade language we generate a generic override that interprets the facade definition model to transform elements. In addition, the facade definition model can also be used to generate custom diagrams for UML authoring tools. Although some manual coding is still required to complete the custom diagram, this greatly reduces the development effort required to develop dedicated user interfaces for the different viewpoints provided by one or more profiles.

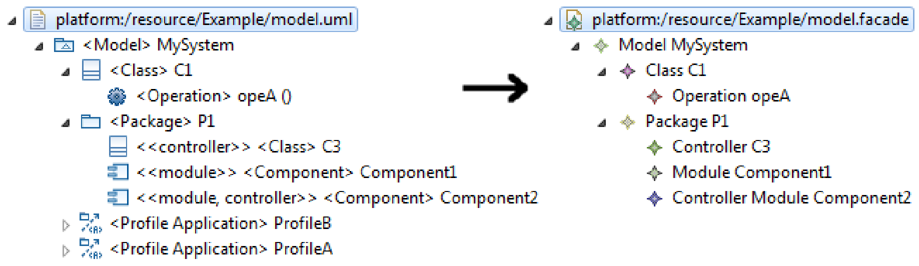
4 Implementation of the FacadeMetamodel Approach

We implemented our approach in Eclipse. The facade metamodel is generated as an Ecore [11] metamodel. The facade language can be created as usual in Eclipse with EMF. It consists in creating a GenModel from witch model, edit, and editor code is generated. The only difference is that we want this language to be only a facade, so that the actual data stored is pure UML. Concretely, it consists in providing a specific implementation of the EMF Resource. This override interprets the facade definition model at runtime to transform, on loading, the UML elements into elements of the actual facade metamodel. During editing, the model is stored in memory where it conforms to the facade metamodel. Upon saving the opposite is done: the UML elements that correspond to the elements of the facade model are generated.

The facade definition can also be used to generate a custom user interface. In Papyrus [12], it consists mainly in generating a custom palette that creates the stereotyped elements, the specific properties view focused on wanted and aliased UML properties, and a specific model explorer.

5 Using the Facade Language

If we consider the example UML model on the left hand side of Fig. 5, the corresponding facade model appears as shown on the right hand side of Fig. 5.



```
platform:/resource/Example/model.uml
  <Model> MySystem
    <Class> C1
      <Operation> opeA ()
    <Package> P1
      <<controller>> <Class> C3
      <<module>> <Component> Component1
      <<module, controller>> <Component> Component2
    <Profile Application> ProfileB
    <Profile Application> ProfileA
```

```
platform:/resource/Example/model.facade
  Model MySystem
    Class C1
      Operation opeA
    Package P1
      Controller C3
      Module Component1
      Controller Module Component2
```

Fig. 5. The UML model (left side) is rendered using the facade language (right side)

However, as shown in Fig. 6. the facade model does not contain anything except a reference to the corresponding UML model.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<source uri="platform:/resource/Example/model.uml"/>
```

Fig. 6. Content of the facade model

The facade language appears to the modeler as a regular pure-DSML. Therefore, all existing tools (such as Acceleo, QVTo, ATL) can use the facade language transparently, even though the model is ultimately a UML model. In other words, the facade is a pure-DSML that has its own tool infrastructure but it remains compatible with the UML tool infrastructure. To illustrate this we give an Acceleo M2T transformation for UML models in Fig. 7 and the same M2T transformation for facade models in Fig. 8. They show that instead of thinking in terms of stereotype attached to Class instances (the "*getAppliedStereotype('ProfileA::Controller')*" and "*getAppliedStereotype('ProfileB::Module')*"), the facade language enables expressing transformations in terms of domain concepts (the "*oclIsTypeOf(ControllerModule)*").

```
[module
generate('http://www.eclipse.org/uml2/3.0.0/UML')]
[template public generateElement(aClass : Class)]
    [if
    ((aClass.getAppliedStereotype('ProfileA::Controller')
    <> null) and
    (aClass.getAppliedStereotype('ProfileB::Module') <>
    null))]
      [file (aClass.name+'.txt', false)]
        ControllerModule =
        [aClass.getValue(aClass.getAppliedStereotype(
        'ProfileA::Controller'), 'p4')/] -
        [aClass.getValue(aClass.getAppliedStereotype(
        'ProfileB::Module'), 'p8')/]
      [/file]
    [/if]
[/template]
```

Fig. 7. Acceleo M2T transformation for UML models

```
[module generate('facade')]
[template public generateElement(aClass : Class)]
    [if (aClass.oclIsTypeOf(ControllerModule))]
      [file (aClass.name+'.txt', false)]
        ControllerModule =
        [aClass.oclAsType(ControllerModule).p4/] -
        [aClass.oclAsType(ControllerModule).p8/]
      [/file]
    [/if]
[/template]
```

Fig. 8. Acceleo M2T transformation for facade models

Note that the UML model must be handled only through the facade language or the custom diagram type. If the UML model is edited directly with a standard UML editor, it can be corrupted such that it no longer conforms to the facade language.

6 Related Work

The following work inspired our approach or sought at similar goal.

The problem of bridging pure-DSMLs and UML profiles has been addressed in [13], which proposes generating transformations between the pure-DSML and the UML profile from a weaving model. However, as the authors note in their paper, the mapping structures they defined are still insufficient to describe the relationship between profiles and pure-DSMLs. The main reason is that they do not clarify the semantics of the Extension relation between a stereotype and its base metaclasses. Also, they assume that the metamodel of the pure-DSML already exists. That assumption makes the definition of the mapping much more complex and, in some cases, the mapping cannot even be defined.

The authors of [14] recognized that profile definition lacks explicit support for UML pruning, aliasing etc. Therefore, instead of directly implementing specific user interfaces, they propose to refine the UML profile definition with “customizations”, which allow defining aliases, pruning some properties and associations, and allowing or forbidding some owned elements and owner elements. This customization is then interpreted by a UML editor to provide the adapted user interface. However, this approach does not provide a pure-DSML metamodel, so that the full UML is hidden only in UML editors but not in other tools.

The OMG is currently working on defining a Metamodel Extension Facility, a new mechanism that would offer more capabilities while aiming at being retro-compatible with existing profiles. It notably provides more filtering capabilities and it allows declaring the scope of applicability of extensions. It also provides a means to define incompatibilities between stereotype applications. This is a first step toward making more explicit extensions choices. However, it does not provide an aliasing feature and the refinement of the metamodel is possible only by adding new constraints. Also, it maps the existing concept of Extension to the concept of Generalization (defined in SMOF [15]). This has the benefit of clarifying the interpretation of the Extension concept but it does not meet all our requirements, such as annotation profiles used in various model analyses, which often consider extensions as if they are ordinary associations.

We also refer to [10], whose authors propose a solution to prune big metamodels (such as UML) to reveal only the “effective-metamodels” i.e. a focused metamodel. This applies directly to our pruning requirement, but, it does not let language designers customize the pruned metamodel nor does it handle profiles.

Creating a virtual schema is not a totally new idea. We refer to Virtual EMF [16], which provides virtual views on heterogeneous models. EMF Facet [17] also provides a way to add virtual information to a metamodel, but it is limited to adding derived properties. In addition, the definition of a facet requires manual definition of queries, which is time demanding. Even though these tools take a similar approach, they do not address the same problem nor do they aim at virtualizing the entire metamodel.

7 Discussions and Summary

Our objective was to enable users to do UML modeling but without being aware of it. Indeed, we observed that the trade-off between designing a pure-DSML and designing a UML profile is not easy choice. We also observed that many pure-DSMLs are often just specialized subsets of UML with slightly different terminology and a small number of additional metaclasses or properties. Consequently, we advocate reusing a standard highly evolved modeling language with well-defined semantics as a pragmatic choice on which to base a DSML.

However, the current profile mechanism does not offer enough flexibility for adapting UML. We therefore propose an approach and a tool that let language designer generate what we refer to as a facade. A facade is an aliasing, a pruning and an extension of the UML that appears like an ordinary pure-DSML but its models are stored internally as fully-compliant UML models. Using our approach, many pure-DSMLs will not have to be created from scratch anymore, because both the semantics and syntax of UML can be reused. Despite being based on UML, modeling languages generated in this manner can be compact and highly expressive, fully capable of concise and precise expression of domain-specific concepts. More important, our solution reuses today available standards.

One could create manually a metamodel that meets its specific needs by reusing the UML metamodel supplemented by one or more profiles. He/she could then create the transformations that translate models from this new pure-DSML to UML and vice versa. Finally he could create dedicated user interface to manipulate the models. However, this approach would be not only time-consuming but also error-prone. Our approach does it automatically and prevents language designers from creating a pure-DSML that is not completely compatible with UML.

In future work we aim at providing more customizations of the preliminary facade metamodel to let, for instance, language designer remove necessary properties with the requirement to implement a getter and a setter.

Acknowledgments. We would like to thank François TERRIER (from CEA LIST) for his valuable comments and useful discussions.

References

1. Object Management Group: Unified Modeling Language (UML), Superstructure - Version 2.4.1 - formal/2011-08-06 (2011), `http://www.omg.org/spec/UML/2.4.1`
2. Object Management Group: Semantics of a Foundational Subset for Executable UML Models (FUML) - Version 1.0 - formal/11-02-01 (2011), `http://www.omg.org/spec/FUML/1.0`
3. ISO/IEC/IEEE: ISO/IEC/IEEE 42010 - Systems and software engineering - Architecture description (2011)
4. Object Management Group: Systems Modeling Language (SysML) - Version 1.2 - formal/2010-06-01 (2010), `http://www.omg.org/spec/SysML/1.2`
5. M2M - Operational QVT Language (QVTO), `http://wiki.eclipse.org/M2M/Operational_QVT_Language_QVTO`
6. Object Management Group: Query/View/Transformation Specification (QVT) - Version 1.1 - formal/2011-01-01 (2011), `http://www.omg.org/spec/QVT/1.0`
7. ATLAS Transformation Language (ATL), `http://www.eclipse.org/m2m/atl/`
8. Acceleo, `http://www.eclipse.org/acceleo`
9. Object Management Group: Object Constraint Language (OCL) - Version 2.2 - formal/2010-02-01 (2010), `http://www.omg.org/spec/OCL/2.2`
10. Sen, S., Moha, N., Baudry, B., Jézéquel, J.M.: Meta-model Pruning. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 32–46. Springer, Heidelberg (2009)
11. Eclipse Modeling Framework (EMF), `http://www.eclipse.org/modeling/emf`

12. Papyrus, `http://www.eclipse.org/modeling/mdt/papyrus/`
13. Abouzahra, A., Bézivin, J., Del Fabro, M.D., Jouault, F.: A practical approach to bridging domain specific languages with UML profiles. In: Proceedings of the Best Practices for Model Driven Software Development at OOPSLA (2005)
14. Silingas, D., Vitiutinas, R., Armonas, A., Nemuraite, L.: Domain-Specific Modeling Environment Based on UML Profiles. In: Proceedings of Information Technologies 2009, pp. 167–177 (2009)
15. Object Management Group: MOF Support for Semantic Structures (SMOF) - 1.0 Beta 2 - ptc/2011-08-21 (2011), `http://www.omg.org/spec/SMOF/1.0/Beta2`
16. Clasen, C., Jouault, F., Cabot, J.: VirtualEMF: A Model Virtualization Tool. In: De Troyer, O., Bauzer Medeiros, C., Billen, R., Hallot, P., Simitsis, A., Van Mingroot, H. (eds.) ER Workshops 2011. LNCS, vol. 6999, pp. 332–335. Springer, Heidelberg (2011)
17. EMF Facet, `http://www.eclipse.org/modeling/emft/facet`

$T_{\square}$: A Domain Specific Language for Rapid Workflow Development

Fazle Rabbi and Wendy MacCaull

Centre for Logic and Information
St. Francis Xavier University
Nova Scotia, Canada
{rfazle,wmaccaul}@stfx.ca

Abstract. In MDE, software systems are always synchronized with their models since changes are made first to the model whenever there are changes in the requirement specifications. While MDE has a lot of potential, it requires maturity and tool support. In this research we present a framework for a workflow management system based on the MDE approach. We propose a domain specific language, $T_{\square}$ (T-Square) for rapidly specifying details of (workflow) tasks and their associated user interfaces which may be used with the NOVA Workflow, an executable workflow management system. $T_{\square}$ includes syntax for writing procedural statements, for querying an ontology, for declaring user interfaces, for applying access control policy, and for scheduling tasks, using Xtext to write the grammar. We apply transformation methods, based on Xtend, to generate executable software from the abstract task specifications. A running example from health services delivery illustrates the usefulness of this approach.

Keywords: Workflow Management System, Model Driven Engineering, Ontology, Domain Specific Language.

1 Introduction

Today software systems are more complex than ever before providing features such as transactions, discovery, fault tolerance, event notification, security, and distributed resource management. Higher level languages have been invented to alleviate the complexity of modern software systems. Advances in languages and platforms during the past two decades have minimized the need to reinvent common and middleware services by providing libraries, APIs, etc. but researchers and developers still need to focus on such technical issues, a major impediment to rapid software development. Software researchers and developers require abstractions of their system to help them program in terms of their design intent rather than in terms of the underlying computing environment [23]. Model Driven Engineering (MDE) raises the level of abstraction in program specification and aims to increase automation in software development. MDE offers a promising approach to further alleviate the complexity of platforms and permits the user an effective way to express domain concepts.

A model is specified by modeling notations or modeling languages. Modeling languages are often tailored to certain domains, and such a language is often called Domain Specific Languages (DSL). A DSL can be visual (e.g., Business Process Modeling Notation (BPMN) [1], Yet Another Workflow Language (YAWL) [26]) or textual (e.g., CSS, regular expressions, ant, SQL). DSLs help developers focus on problem domains rather than on technical details.

Workflow models are used to describe the behaviour of complex processes (workflows) which may have characteristics like concurrency, resource sharing and synchronization. Workflows often require massive data and knowledge management and sometimes need to be modified at short notice. With these aspects in view, we present a DSL called $T_\Box$ (T-Square) for writing workflow task specification. It incorporates the following features: a) a simple syntax for i) writing procedural statements, ii) querying and manipulating ontologies, iii) designing a rich user interface (UI), iv) code generation, v) specifying access control policy, vi) dynamically scheduling tasks; b) abstraction of communication details; and c) ease of customization by dramatically reducing the number of lines of code.

The user-friendly syntax for querying and manipulating ontologies in $T_\Box$ allows the developer to use domain concepts directly in his/her programming. Ontologies are used in $T_\Box$ for data and knowledge persistence. An ontology is a knowledge representation technique which can represent complex business rules declaratively. In contrast to traditional knowledge-based approaches, ontologies seem to be well suited for an evolutionary approach for the capture of domain knowledge and for the specification of requirements [27]. Software modeling languages and methodologies can benefit from the integration with ontology languages in various other ways, e.g., by reducing language ambiguity, and by enabling validation and automated consistency checking of facts [25]. Moreover intelligent applications for example, ontology driven workflows, may be built based on ontology reasoning. $T_\Box$ has been incorporated in the NOVA Workflow workbench [17]. A workflow may be designed graphically in NOVA Workflow and workflow task specification is written using $T_\Box$. The control flow along with the task specification are automatically transformed to executable server side (J2EE) components and client side applications (e.g., mobile applications). The automated transformation of workflow models and task descriptions to executable software greatly reduces the effort required for adapting requirement changes to software applications.

Our work is especially motivated by problems from health services delivery [16]. Health care frequently requires that workflows conform to national or provincial guidelines, which contain tasks and procedures that must be customized to a local setting (e.g., hospital, clinic or doctor's office). These workflows not only need to be modified frequently, their business logic is often partialy or wholly dependent on domain knowledge that is coded by an ontology. Ontologies are common in health care to standardize terminology across districts. For example medication ontologies may be used by pharmacists and physicians to organize and standardize the knowledge and concepts involved in modern medicine. $T_\Box$ allows us to use this knowledge during workflow enactment to guide the control flow. Ensuring the correct transformation of $T_\Box$ code to a software system gives us the confidence that the generated software components will produce correct workflows, a very essential feature for safety critical systems such as health care.

The rest of the paper is organized as follows: in section 2 we propose the model driven approach for workflow development; in section 3 we give details of the proposed language $T_\Box$ together with a running example; in section 4 we present some related works and in section 5 we conclude the paper.

2 Model Driven Workflow Development with $T_\Box$

$T_\Box$ is a procedural language with declarative features. It has been designed to develop certain aspects of workflows which can be used in many workflow systems. Workflow systems are often designed with graphical workflow modeling languages such as BPMN, YAWL, the Compensable Workflow Modeling Language (CWML) [21], etc. These languages use abstract notation for tasks and control flow to visualize workflow but to describe the detailed specification of a task, some workflow systems use XML-based languages, while others use a general-purpose programming language (GPL) such as C++, Java, etc. A control flow consists of two artefacts: i) the flow relation, and ii) branching conditions. When a workflow executes, it follows the flow relation as described in the graphical model and executes the tasks (i.e., specifications written in XML or GPL). During execution, the branching conditions are evaluated to guide the control flow.

We propose $T_\Box$ for describing tasks and branching conditions, but for defining the flow relation and visualizing a workflow, existing workflow languages may be used. $T_\Box$ has been incorporated in NOVA Workflow [17] which uses the graphical workflow modeling language CWML and has an executable workflow engine. CWML is a block structured workflow modeling language [21] which has compensable components along with common workflow components (e.g., atomic tasks, control flow operators, such as XOR, OR, AND, etc.). Each atomic task in CWML is associated with a task description file (written in $T_\Box$) containing procedures. XOR, OR, and Internal Choice control flow operators require branching conditions to route the flow; these branching conditions are specified in $T_\Box$ as procedures which may consult an associated ontology.

Fig. 1 shows the overall architecture of the proposed approach using $T_\Box$ for model driven workflow development. We developed an editor for $T_\Box$ using Xtext [6] and integrated it with the NOVA Editor (for modeling workflows). Specifications written in $T_\Box$ are automatically transformed to executable Java programs using Xtend [2]. Each task specification may consist of code for user interface, business logic, access control, etc. One writes the user interface related code in a procedure named `view`, the business logics (e.g., action statements) in a procedure named `action`, the access control related code in a procedure named `accessPolicy`, etc. In $T_\Box$, if a task description file contains a procedure named `view`, the `view` procedure is transformed to a client side Java application. The automatic transformation technique for $T_\Box$ generates code for Android [3] application. All other procedures are transformed to executable Java server side

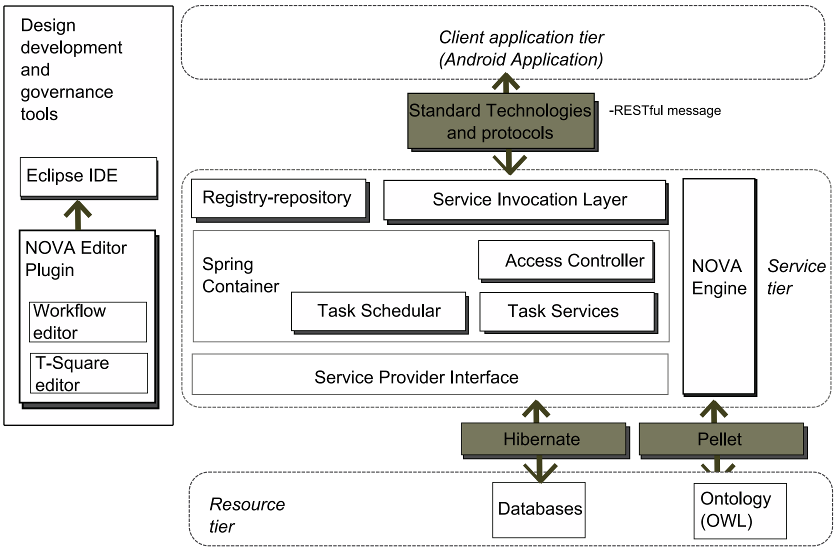


Fig. 1. Overall architecture

programs; in NOVA Workflow we used J2EE server side components. A `view` method may invoke other procedures; this will initiate an asynchronous data communication to the server by Web Service (RESTful message). If a task executes, meaning the execution of a procedure named `action` at the server side, the NOVA Workflow engine updates the control flow of the workflow according to the graphical workflow model. For querying and manipulating ontologies we used Pellet [24] which is a sound and complete OWL-DL reasoner with extensive support for reasoning about individuals. In $T_\square$, ontology queries are written in the SQWRL (Semantic Query-enhanced Web Rule Language) [19] format but are translated to the SPARQL-DL query format since the Pellet reasoner supports SPARQL-DL. SQWRL is a query language for OWL [11][5] built on SWRL [4], a rule language which includes a high-level abstract syntax for Horn-like rules. The basics of description logic, OWL and SPARQL-DL may be found in [7] [5] [20]. We chose the SQWRL query syntax for $T_\square$ because of its simplicity.

3 The $T_\square$ Language

A workflow management system often deals with many users and resources. Typical requirements of a workflow management system are presented here by a real life problem description. The 'Guidelines for the management of cancer-related pain in adults' [9] is a guideline for pain management of cancer related pain. The guidelines suggests the use of Opioids for cancer patients. Opioids are very useful in cancer care to alleviate the severe, chronic, disabling pain but there are common side effects in patients taking opioids which include: nausea and vomiting, drowsiness, itching, dry mouth, miosis, and constipation. The proper use of opioid dosage is important.

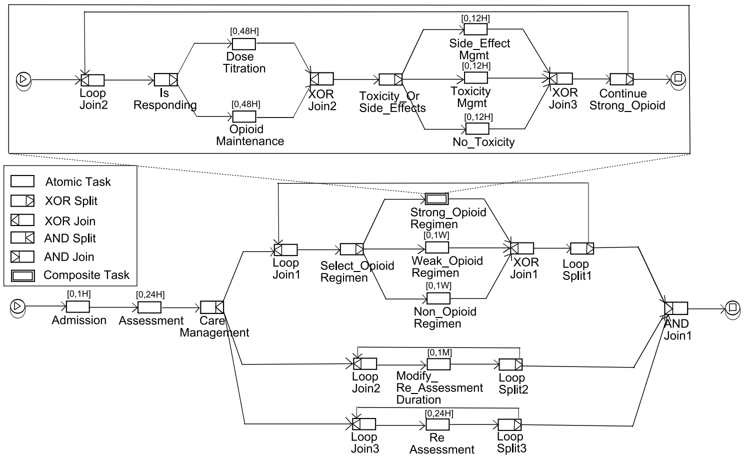


Fig. 2. Pain management workflow model defined in CWML

We designed a workflow model from the guideline. Fig. 2 shows the sequence of tasks. A patient is admitted to the system and then pain is assessed. The 'Assessment' task assesses all causes of pain, determines pain location, pain intensity, and documents all previous analgesics. The 'Care Management' is an AND split which activates all of its outgoing branches. During the workflow execution, tasks in these branches can execute concurrently. The 'AND Join1' is an AND join which synchronizes the control flow of its incoming branches. The 'Select_Opioid Regimen' is an XOR split which activates only one of its outgoing branches. The patient's opioid regimen is determined by her pain intensity and medication information. The 'Strong_Opioid Regimen' is a composite task which is unfolded to a subnet workflow. The guideline also suggests different re-assessment times for patients depending on the patient's pain intensity and opioid regimen. We used loops to model the fact that re-assessment tasks are done repeatedly.

The following requirements are clear from the given problem description: i) use of domain concepts; ii) user interaction; iii) data persistence; iv) task and control flow; v) task scheduling; vi) access control policy. To represent the domain concepts we built an ontology from the guideline. Fig. 3 shows the class hierarchy of the ontology. We defined three subclasses of *Medication* named *StrongOpioid*, *WeakOpioid*, and *NonOpioid* according to the guideline. A patient can be either in the 'Non Opioid', 'Weak Opioid' or 'Strong Opioid' regimen depending on his current medication. If a patient is taking any *StrongOpioid* medicine he is under 'Strong Opioid' regimen. Some of the class definitions are given below:

StrongOpioid ≡ Medication and (((hasFrequency value Q4h) and (hasMeasures value units) and (isMadeOf value Acetaminophen) and (hasDose some double[≥ 1.0])) or ((hasFrequency value Q4h) and (hasMeasures value units) and (isMadeOf value Oxycodone) and (hasDose some double[≥ 1.0])) or ...)

PatientUnderStrongOpioid ≡ Patient and (hasMedication some StrongOpioid)

For example, we see that *StrongOpioid* is a *Medication* which has certain properties: Acetaminophen 1 tablet (units) or more every 4 hours (i.e., Q4h), or Oxycodone (eg. Percocet) 1 tablet (units) or more every 4 hours (i.e., Q4h), etc. In the following subsections we provide the syntax of $T_\Box$ and see how it relates to the above mentioned requirements.

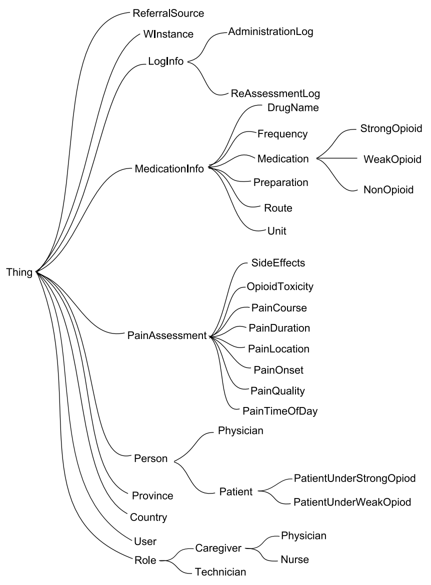


Fig. 3. Simplified class hierarchy of ‘Pain Ontology’

3.1 Writing Procedural Statements

Variables in $T_\Box$ are inferred variables; variable types are determined from their use. Variables in $T_\Box$ may be indexed as array indexes but a declaration of the size is not required. The size is adjusted dynamically at execution time. If no index is used, it refers to the 0^{th} index of a variable. In $T_\Box$, procedures may be invoked by ‘call by value’ or ‘call by reference’. The ‘call by reference’ of a variable is indicated by a leading ‘&’. In $T_\Box$ syntax for the Assignment operation, If-Else statements, For-loops, etc., are similar to that for the C family of languages.

Note that in $T_\Box$ every procedure returns a value, and return types are not required for procedures. In $T_\Box$ some utility procedures such as `size`, `today`, `currentTime`, `date`, `month`, `year`, `time`, and `tokenize` have been incorporated to deal with strings, arrays, dates and times.

3.2 Querying and Manipulating Ontologies

Ontologies allow data and rules to be organized efficiently so as to permit the calculation (i.e., inference) of implicit knowledge from explicit information. Using ontologies to drive workflows allows for a more compact representation of the workflow and changes made in an ontology (which are often simple to implement) can avoid the need to change the workflow (which can be more complicated). An important aspect in the design of $T_\Box$ was the facility to query and manipulate ontologies. It provides four different tags to perform Create (C), Read (R), Update (U), and Delete (D) operations (CRUD operations) in an ontology. $T_\Box$ allows us to write queries in the easy-to-use SQWRL format. One can perform queries combining concepts and facts from the Tbox and/or Abox. The Tbox describes conceptualizations and contains assertions about concepts such as subsumption ($Man \sqsubseteq Person$) [7]. The Abox contains role assertions between individuals ($hasChild(John, Mary)$) and membership assertions ($John : Man$). Similar to the ‘select’ operator of SQWRL, the ‘select’ operator in $T_\Box$ takes one or more arguments, which are typically variables used in the pattern specification of the query. A particular value may be passed as a query criterion; if a variable is used in an ontology query without a leading question mark (?) then the value is read by the query engine. For example, the following query, written in the SQWRL format, retrieves all persons in an ontology with a pain intensity that is greater than 5, together with their pain intensities:

```
var  p, pain, v = 5;
{R$ Patient(?p), hasPain(?p, ?pain), greaterThan(?pain, v) →
     select(?p, ?pain) $R}
```

The query engine will populate the variables passed as arguments of the 'select' operator. Selected results may be sorted in ascending (descending) order by the 'orderBy' ('orderByDescending') operator. To create a new instance/individual or relation in the ontology Abox, the OntAssertion statements may be used directly from $T_{\square}$. The following OntAssertion statements create a new 'Patient' individual and inserts a data property for the relation 'hasPain'.

```
var  p;
{C$ p := Patient("Alex") $C}
{C$ hasPain(p, 6) $C}
```

Note that a reference of the newly created Patient individual is assigned to the variable 'p'. An individual may be created with an auto-incremented identity (id) if in the ontology there exists a data property named 'hasId', where the domain of 'hasId' is 'Thing' and range is 'Long' data type. The following code shows how to create a new Patient individual with an auto-incremented id.

```
{C$ p := Patient(newid) $C}
```

OntDel statements may be used to delete an individual or a relation from an ontology Abox. The following code shows how to delete a Patient with id=1010.

```
var p, pid = 1010;
{R$ Patient(?p), hasId(?p, pid) → select(?p) $R}
{D$ Patient(p) $D}
```

In this code fragment, a search operation is performed on an ontology for a Patient individual with id=1010 and a reference is retrieved; the Patient individual's reference is then passed as an argument to the delete operation. To update a data property or object property of an individual, OntUpdate statements may be used. Following code fragment shows the use of an update operation.

```
var p, P, bDate, Age, age, newAge, cDate = today();
{R$ Patient(?P), hasBirthDate(?P, ?bDate), isEqual(?bDate, cDate),
    hasAge(?P, ?Age) → select(?P, ?Age) $R}
foreach( p in P, age in Age){
    newAge = age + 1;
    {U$ hasAge(p, age => p, newAge) $U}
}
```

This code fragment updates the ages of all patients whose birthday is today.

3.3 Designing User Interfaces

Applications such as health services delivery require that data be gathered from many sources. Many "forms", often specific to a local setting, are required. For example in a palliative care program the care team for any particular patient can be comprised of 10–15 professionals (nurses, social workers, specialists of various kinds) and in the course of treatment, 40 or more forms must be filled in, some repeatedly. Data gathered must be available to a variety of people in a variety of settings. Health services delivery (and other paper-based services such as the insurance industry) are catching up to the electronic age. Health professionals and clients alike are demanding electronic health records so electronic forms are a necessity. $T_\square$ was designed to meet these needs.

To print a text or a number in a UI, the `getLabel` procedure may be used. The `getLabel` procedure produces a 'Label' view component in the UI. One can pass either a string literal or a variable as argument of the `getLabel` procedure. If a variable is passed, the variable is bound to a 'Label' view component. Whenever this variable is updated, the change is reflected in the 'Label'.

```
var wid = 112;
getLabel("WorkflowInstance : ");
getLabel(wid);
```

This code fragment produces two labels; during execution, the first label will display the text "Workflow Instance:" and the second label will display the number '112'.

The `getText` procedure produces a 'Text Field' view component. A 'Text Field' is a common UI component to take user input. The `getText` procedure can take one or two arguments: i) a string to produce a label, and ii) a variable (optional) to display the initial text in a 'Text Field'. A destination variable name after the symbol '>>' is required for a `getText`. The user input is captured by the destination variable. Optionally, some statements (also known as action statements) may be written inside curly braces after the destination variable name of a `getText` procedure. These action statements will be executed when a user finishes her entry into the 'Text Field'. The following code fragment uses the `getLabel` and `getText` procedures:

```
var hospitalName, displayText = "NoInput";
getText("EnterHospitalName : ") >> hospitalName{
    displayText = "Hospital : " + hospitalName; };
getLabel("EnteredText : ", displayText);
```

This produces a 'Text Field' where the user will enter text input; the entered text will be stored in a variable named 'hospitalName'. As soon as the user finishes entering text into the 'Text Field' the action statement (enclosed with a curly bracket) will execute and assigns a value entered by the user to the variable named 'displayText'. Since the variable 'displayText' is bound to a 'Label', when its value changes, the 'Label' view component will be updated and will display the hospital name entered in the 'Text Field'.

The `getInteger` procedure is similar to the `getText` procedure; this also produces a ‘Text Field’ to take input from the user, but the difference is that only numbers are allowed here. The following code fragment gives an example:

```
var basicPay = 14, hourlyPay, totalHr = 0, totalSalary = 0;
getInteger(“Hourlypayment : $”, basicPay) >> hourlyPay {
    totalSalary = hourlyPay * totalHr; };
getInteger(“TotalHourWorked : ”) >> totalHr {
    totalSalary = hourlyPay * totalHr; };
getLabel(“TotalSalary : $”, totalSalary);
```

The first ‘Text Field’ will display the value of the ‘basicPay’ variable which is ‘14’. The user may change it by inserting a different number; the entered number will be stored in the variable named ‘hourlyPay’. The user can also enter the total hours worked in the second ‘Text Field’. Whenever the user finishes entering numbers in either of the ‘Text Fields’ the total salary is calculated and displayed in the UI by a ‘Label’.

The `getDouble` and `getDate` procedures are similar to the `getInteger`; here the user can enter a floating point number and date respectively. The `getTextMultiple` procedure is similar to the `getText` procedure but it produces a ‘Text Area’ (for multiline text input) instead of a ‘Text Field’. The `getBoolean` procedure takes one argument as input to display a title for a ‘Check box’ (a view component to select or de-select an item). The user may select or de-select the ‘Check box’ and a true or false value is assigned to the associated destination variable of a `getBoolean` procedure. If action statements are written for a `getBoolean` procedure, they will be executed after the user selects or de-selects a check box item.

```
var painCrisis;
getBoolean(“PainCrisis”) >> painCrisis;
```

The above code fragment will display a ‘Check box’ in the UI. The destination variable of the ‘Check box’ is ‘painCrisis’ which will be assigned with a ‘true’ or ‘false’ value depending on the selection of the ‘Check box’. Since the ‘painCrisis’ variable is bound to a ‘Check box’ view component, if the value is changed from another portion of the procedure, it will be reflected in the UI. This feature may be useful to display a form to update existing information. For example if we want to display a patient’s existing Pain Crisis information and allow the user to modify it, we can use the following code:

```
var painCrisis, id = 1010;
getBoolean(“PainCrisis”) >> painCrisis;
{R$ Patient(?p), hasId(?p, id), hasPainCrisis(?p, ?painCrisis) →
    select(?painCrisis) $R}
```

The `getOne` procedure is used to select one item from a list of items. This procedure will either display a ‘Drop Down’ view component (if one source variable is provided as argument) or a ‘Table’ with ‘Radio buttons’ (if more than one source variable is provided). The user cannot select more than one item from the displayed list. A destination variable name is required for a getOne procedure where the selected item (user input) will be stored. Optionally another destination variable name may be mentioned to store the position of the item selected from the source variable. If action statements are given for a getOne procedure, they will be executed as soon as the user selects an item. In the following example a list of countries is retrieved from an ontology by performing the read operation. A getOne procedure is used to display the list of country in a 'Drop Down'. Another getOne procedure is used to display a list of provinces in another 'Drop Down'. Since the provinces are different from one country to another, on the selection of a country, a further query is performed on the ontology to retrieve related province information; this is done in the action statements. The 'province' variable is bound as the source variable with the second getOne procedure; as a result, if provinces' information were updated they will be reflected in the 'Drop Down'.

```
var c, country, province, selectedProvince;
{R$ Country(?c) → select(?c) $R}
getOne("Country : ", c) >> country {
    {R$ Province(?province), hasCountry(?province, country) →
      select(?province) $R}
};
getOne("Province : ", province) >> selectedProvince;
```

Note that the 'source' variable fills a 'Drop Down' view component but if we want to display one item from the items available in the 'Drop Down' we may use the 'destination' variable. For instance, in a patient's admission record update form, we want to display the patient's existing province in the 'Drop Down'; this can be achieved by assigning the name of the province to the destination variable.

The getMultiple procedure is similar to the getOne procedure but here the user may select more than one item from the source variable(s). The values of the source variable(s) are either displayed in a list of Check Boxes or in a 'Table' with 'Check Boxes' (for more than one source variable).

The getButton procedure produces a button in the UI. When a button is pressed, the statements associated with it are executed. $T_\square$ provides two procedures to arrange the view components in the UI; namely openLayout and closeLayout. The openLayout procedure takes an integer parameter which indicates the number of columns. All view components mentioned after a openLayout procedure will follow this arrangement. A closeLayout procedure stops putting view components in the order started by an openLayout procedure. An openLayout procedure can be nested with another openLayout procedure; in this way a complex table layout structure may be achieved.

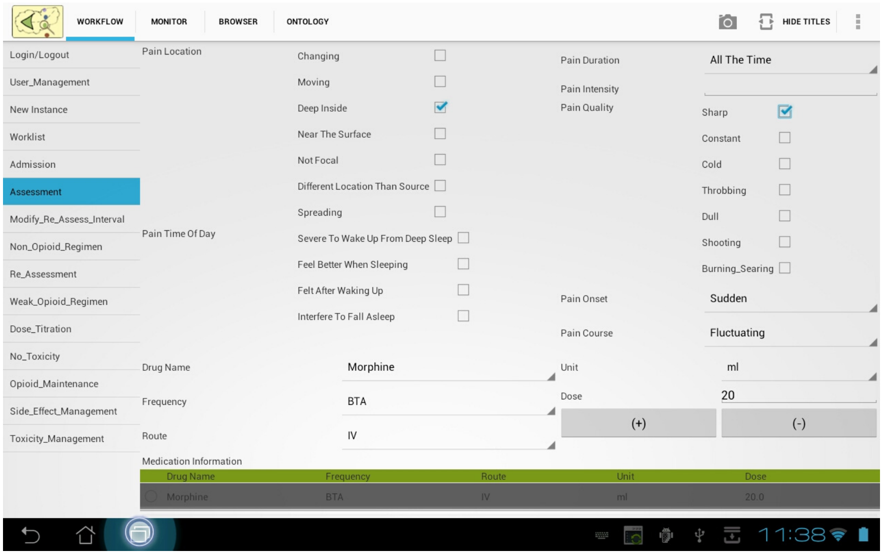
3.4 Automatic Code Generation

Task specifications written in $T_\square$ are automatically translated to executable Java code by Xtend. This is one of the main advantage of using MDE for building software systems. We use $T_\square$ to design a user interface for a task from Fig. 2. The code below shows the view procedure of the 'Assessment' task.

```
func view(){
 var wInstance, pc;
 . . .//Other variable declaration
 wInstance  =  getCurrentInstance(); // Get current workflow instance id
 {R$ PainCourse(?pc), hasName(?pc, ?pcName)  →  select(?pcName)$R}
 . . .// Read other pain assessment information from ontology
 {R$ Drug(?drug)  →  select(?drug)$R}
 . . .// Read Frequency, Route, Unit information from ontology
 openLayout(2); openLayout(2); // Nested Table Layout
 getMultiple("PainLocation", pLocation)  >>  painLocation;
 getMultiple("PainTimeOfDay", pTime)  >>  painTime;
 closeLayout(); openLayout(2);
 getOne("PainDuration", pDuration)  >>  painDuration;
 . . .// Code to display other view components
 getButton("(+)") {
 drugList[size(drugList)]  =  selDrug; // Adding a new drug into drugList
 . . .// Add frequency, route, dose information into list
 };
 getButton("(−)"){
 clear(drugList, medPos); // Removing selected item from list (Table)
 . . .// Remove frequency, route, dose information from list
 };
 closeLayout(); openLayout(1);
 getOne("MedicationInformation", drugList "DrugName", freqList
  "Frequency", routeList "Route", unitList "Unit", doseList "Dose")
  >> destDrug, medPos; // Showing medications in a table
 closeLayout();
 // make a submit button to send information to server
 submit(wInstance, painLocation, painTime, painDuration, . . .);}
```

The transformation method automatically produced 1160 lines of Java code, and a few xml configuration files to manage the android UI including network operations. The value of the MDE approach, incorporating a simple-to-use DSL and automatic code generation is abundantly clear.

The output of this procedure is shown in Fig. 4, which is a screenshot from a Tablet device operating on 'Android' operating system. The 'Pain Location', 'Pain Duration', 'Drug Name' etc., information comes from the ontology and is displayed in the UI. The clinician selects a drug name, frequency, route, and unit from 'Drop Down' view components and inserts dose in a 'Text Field' and adds them into the 'Medication Information' table by clicking on the (+) button. When each piece of medication information is sent to the server, it is stored in the ontology Abox which will support further reasoning on this data. We performed such reasoning to select a patient's opioid regimen after the execution of the 'Assessment' task. We used the ontology reasoning to classify medications into opioids. The rules for different opioids were incorporated into the ontology. In this way, complex rules of domain knowledge can be effectively handled by using the reasoning power of an ontology. During execution only one outgoing branch of the XOR task 'Select_Opioid Regimen' is activated. A patient goes



Pain Location
- Changing ☐
- Moving ☐
- Deep Inside ✓
- Near The Surface ☐
- Not Focal ☐
- Different Location Than Source ☐
- Spreading ☐

Pain Time Of Day
- Severe To Wake Up From Deep Sleep ☐
- Feel Better When Sleeping ☐
- Felt After Waking Up ☐
- Interfere To Fall Asleep ☐

Pain Duration: All The Time

Pain Intensity:

Pain Quality
- Sharp ✓
- Constant ☐
- Cold ☐
- Throbbing ☐
- Dull ☐
- Shooting ☐
- Burning_Searing ☐

Pain Onset: Sudden

Pain Course: Fluctuating

Drug Name: Morphine

Frequency: BTA

Route: IV

Unit: ml

Dose: 20

Medication Information

Drug Name	Frequency	Route	Unit	Dose
Morphine	BTA	IV	ml	20.0

Fig. 4. Assessment form: Output of the `view` procedure of the 'Assessment' task

into strong opioid regimen if he is currently on a strong opioid or he is on a weak opioid with moderate severe pain or unstable pain. A patient goes into the weak opioid regimen if his pain intensity is mild with unstable pain, otherwise he goes into the non opioid regimen. The code is provided below; the procedure `getBranchCondition` takes two parameters: workflow instance id and branch number and returns true for that branch that should be activated for a particular workflow instance.

```
xorsplittask Select_Opioid_Regimen;
func getBranchCondition(wInstanceId, brNo){
 var p, pid, pIntensity, pUnderStrong, pUnderWeak, pc, painCourse;
 {R$ Patient(?p), hasWfInstance(?p, wInstanceId), hasId(?p, ?pid),
  hasPainIntensity(?p, ?pIntensity) → select(?p, ?pid, ?pIntensity) $R}
 {R$ PatientUnderStrongOpioid(?pUnderStrong), hasId(pid)
     → select(?pUnderStrong) $R}
 {R$ PatientUnderWeakOpioid(?pUnderWeak), hasId(pid)
     → select(?pUnderWeak) $R}
 {R$ Patient(p), hasPainCourse(p, ?pc), hasName(?pc, ?painCourse)
     → select(?painCourse) $R}
 if(brNo = 1){
  if(pUnderStrong ≠ null || (pUnderWeak ≠ null && pIntensity ≥ 4) ||
   (pUnderWeak ≠ null && painCourse = "GettingWorse") ||
   (pIntensity ≥ 4 &&
      (painCourse = "Fluctuating" ||painCourse = "Getting Worse")))
   return true;}
 else if(brNo = 2){
  if( (pIntensity ≥ 2 && (painCourse = "Fluctuating" ||
     painCourse = "GettingWorse")) || (painCourse = "Getting Worse"))
  return true;}
 else if(brNo = 3)
  return true;
return false;}
```

A patient is administered with his prescribed medicine in the 'Strong_Opioid Regimen', 'Weak_Opioid Regimen', and 'Non_Opioid Regimen' tasks; the 'Re Assessment' task executes concurrently with these tasks. The 'Strong_Opioid Regimen' is a composite task which is unfolded to a subnet workflow. This subnet workflow deals with any opioid toxicity or side effects found during the treatment procedure.

3.5 Dynamic Task Scheduling

Time plays an important role for task scheduling. Several explicit time constraints have been identified for time management [15]. The parameters *delay* and *duration* suffice to capture most time constraints. *Delay* is the time duration between two subsequent tasks. This time constraint indicates that tasks can start only when its predecessor tasks are finished. *Duration* is the time span required to finish a task. In [18] we incorporated time in Nova Workflow and presented the formal semantics of *Timed Compensable WorkFlow Nets*. The workflow in Fig. 2 is a Timed Workflow net. Each atomic task in this workflow is associated with a pair of time parameters, $[delay, duration]$. *Delay* time '0' indicates that the task becomes enabled as soon as its predecessor tasks have finished their execution. *Duration* time '24H' for task 'Re Assessment' indicates that the task should finish its execution within 24 hours after it becomes enabled. But in real life, the requirements for *delay* and *duration* may not be static. For example, in our workflow, if the patient's symptoms are not under control, daily assessments need to be done in person by an attending health professional. If symptoms are under control the assessment can be completed weekly. The nurse may need to modify the *duration* time for the 'Re Assessment' task for a particular patient at various points during care. Using $T_{\Box}$ one can specify dynamic time constraints for tasks by providing two procedures, `getDelay` and `getDuration`, for each atomic task. The following code fragment shows the `getDuration` procedure of the 'Re Assessment' task.

```
func getDuration(){
   var wInstance = getCurrentInstance(), p, aInt;
   {R$ Patient(?p), hasWfInstance(?p, wInstance),
     hasInterval(?p, ?aInt) → select(?p, ?aInt) $R}
   return aInt;
 }
```

The `getDuration` procedure queries the ontology and depending on the patient's pain level and other symptoms the ontology reasoner returns the assessment interval (variable *aInt*) attribute which was updated from the task 'Modify Re_Assessment Duration' for this patient. So the duration of the task 'Re Assessment' is now dynamic which satisfies the dynamic scheduling requirement for a task. The `getDelay` procedure may be specified similarly.

3.6 Specifying Task-Based Access Control Policy

Privacy of information is an essential requirement of modern day information systems. The potential misuse of sensitive information has grown considerably with electronic storage of information. In health services many providers need to have ready access to some parts of a patient's data and access is largely dependent on the role of the health professional and the current task [14]. Ontologies are used to organize the facts and rules pertaining to patients, data and the role of the caregivers.

Different access control policies may be specified for different tasks by writing a procedure named `accessPolicy` for each task. A utility procedure named `getCurrentUser` was provided in $T_{\Box}$ to determine the current signed in user. An ontology based access control policy may be easily specified by $T_{\Box}$. For example, in our ontology we have defined a hierarchy of roles (see Fig. 3). *Caregiver* is a superclass of *Physician* and *Nurse*. A *User* individual has the *hasRole* relationship with *Role*. The following code fragment specifies an access policy for the task 'Re Assessment' which gives access to the users who have the *Caregiver* role. This procedure will be consulted before any other procedures execute at the server for the task 'Re Assessment'.

```
func accessPolicy(){
   var uid = getCurrentUser(), u, role;
   {R$ User(?u), hasId(?u, uid), hasRole(?u, ?role),
     Caregiver(?role) → select(?u) $R}
   if(u ≠ null) return true;
   else return false; }
```

4 Related Work

Much research has been done in last two decades to handle similar problems. An adaptive process management system, ADEPT2, was presented by Reichert et. al., in [22] which supports dynamic change of process schema and definition. In ADEPT2 a block structured workflow modeling language similar to CWML has been used, although CWML has more features such as compensation. The main difference between ADEPT2 and NOVA Workflow is their underlying persistent technology and data structure; ADEPT2 does not support ontologies and the activities in ADEPT2 are written in a GPL. In ADEPT2 web forms are automatically generated from the workflow model although ADEPT2 does not allow action statements for UI operations. ADEPT2 performs a dynamic validation of process schema change which makes the workflow system consistent. In NOVA Workflow a consistency check is performed whenever any record is inserted into or updated from an ontology Abox.

In [12] the authors presented an evolutionary approach for the model-driven construction of Web service based Web applications on the basis of workflow models founded on DSLs and a supporting technical framework. The Workflow DSL is an executable specification language for workflow based Web applications which allows the use of various graphical notations taken from the workflow modeling field, e.g., BPMN, Petri Nets, UML activity diagrams etc., as well as custom notations. This model driven design approach makes development faster by reusing components but it is not ontology based. Workflow development with $T_\Box$ can benefit from reusing an ontology.

In [13] the author worked on ontology oriented programming and proposed a compiler which produces a traditional object-oriented class library that captures the declarative norms of an ontology. The developer is required to use a GPL and the approach is not model driven.

In [10] the authors introduced the knowledge representation features of a multi-paradigm programming language called Go! that integrates logic, functional, object oriented and imperative programming styles. In that paper the authors described the Go! language and its use for ontology oriented programming, comparing its expressiveness with OWL. This is related to our work since the authors also proposed a language for building executable ontologies. However the syntax proposed for $T_\Box$ is simple and abstract and $T_\Box$ provides syntax for control flow and UI design.

In [8] Baker et. al., surveyed a large number of existing workflow systems and listed their features considering different problem aspects. But none of them is following ontology based model driven approach as in $T_\Box$. Although T-Square has been implemented in the NOVA Workflow and integrated with CWML, it is not limited to CWML, it may be integrated with other workflow modeling languages, because it provides a nice way to separate business logic from control flow. What is missing in existing approaches, such as BPMN based commercial tools, is the latter use general purpose programming languages (GPL) or XML based languages which cannot provide proper abstraction allowing the developer to concentrate on the domain model.

5 Conclusion

In this paper we provided the syntax and semantics for the $T_\Box$ language. A developer may learn the simple syntax of $T_\Box$ and start developing applications without needing detailed knowledge of the complex API's for Ontology, Web Service, Android, etc. Code is generated automatically, allowing the developer to fully concentrate on the domain model and system analysis. If there is a change in user requirements, the developer can make the change in $T_\Box$ and the NOVA workflow system will automatically update the software accordingly. The output of NOVA Workflow is currently an Android application which runs on mobile devices but different transformations may be applied to generate other applications, for iPad, the Web, etc. End users interact with the client application.

Since the reasoning process over an ontology is time consuming, in future we will work on a bigger case study and deal with the problems of scalable ontology reasoning. One approach is to use a relational database and materialize an ontology into a database; research to speed up the materialization to permit frequent updates is required. We are investigating an incremental ontology materialization approach.

References

1. BPMN: Business Process Model and Notation (BPMN), http://www.omg.org/spec/BPMN (last accessed, January 2012)
2. Eclipse xtend, http://www.eclipse.org/xtext/xtend (last accessed, January 2012)
3. Google android, http://www.android.com (last accessed, January 2012)
4. SWRL, http://www.w3.org/submission/swrl (last accessed, January 2012)
5. Web Ontology Language (OWL), http://www.w3.org/2004/owl (last accessed, January 2012)
6. Xtext, http://www.eclipse.org/xtext (last accessed, January 2012)
7. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2003)
8. Barker, A., van Hemert, J.: Scientific Workflow: A Survey and Research Directions. In: Wyrzykowski, R., Dongarra, J., Karczewski, K., Wasniewski, J. (eds.) PPAM 2007. LNCS, vol. 4967, pp. 746–753. Springer, Heidelberg (2008)
9. Broadfield, L., Banerjee, S., Jewers, H., Pollett, A., Simpson, J.: Guidelines for the Management of Cancer-Related Pain in Adults. In: Supportive Care Cancer Site Team, Cancer Care Nova Scotia (2005)
10. Clark, K.L., McCabe, F.G.: Ontology oriented programming in Go! Appl. Intell. 24(3), 189–204 (2006)
11. Dean, M., Schreiber, G.: OWL web ontology language reference. W3C recommendation, W3C (February 2004)
12. Freudenstein, P., Nussbaumer, M., Allerding, F., Gaedke, M.: A domain-specific language for the model-driven construction of advanced web-based dialogs. In: Proceedings of the 17th International Conference on World Wide Web, WWW 2008, Beijing, China, April 21-25, pp. 1069–1070. ACM (2008)
13. Goldman, N.M.: Ontology-Oriented Programming: Static Typing for the Inconsistent Programmer. In: Fensel, D., Sycara, K., Mylopoulos, J. (eds.) ISWC 2003. LNCS, vol. 2870, pp. 850–865. Springer, Heidelberg (2003)
14. Leyla, N., MacCaull, W.: A Personalized Access Control Framework for Workflow-Based Health Care Information. In: Daniel, F., Barkaoui, K., Dustdar, S. (eds.) BPM Workshops 2011, Part II. LNBIP, vol. 100, pp. 273–284. Springer, Heidelberg (2012)
15. Li, W., Fan, Y.: A time management method in workflow management system. In: Workshops at the Grid and Pervasive Computing Conference, pp. 3–10 (2009)
16. MacCaull, W., Jewers, H., Latzel, M.: Using an interdisciplinary approach to develop a knowledge-driven careflow management system for collaborative patient-centred palliative care. In: ACM International Health Informatics Symposium, IHI 2010, Arlington, VA, USA, pp. 507–511. ACM (2010)
17. MacCaull, W., Rabbi, F.: NOVA Workflow: A Workflow Management Tool Targeting Health Services Delivery. In: Liu, Z., Wassyng, A. (eds.) FHIES 2011. LNCS, vol. 7151, pp. 75–92. Springer, Heidelberg (2012)
18. Mashiyat, A.S., Rabbi, F., MacCaull, W.: Modeling and Verifying Timed Compensable Workflows and an Application to Health Care. In: Salaün, G., Schätz, B. (eds.) FMICS 2011. LNCS, vol. 6959, pp. 244–259. Springer, Heidelberg (2011)
19. O'Connor, M.J., Das, A.K.: SQWRL: A query language for OWL. In: Proceedings of the 5th International Workshop on OWL: Experiences and Directions (OWLED 2009), vol. 529 (2009)

20. Prud'hommeaux, E., Seaborne, A.: SPARQL Query Language for RDF (Working Draft). Technical report, W3C (March 2007)
21. Rabbi, F., Wang, H., MacCaull, W.: Compensable WorkFlow Nets. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 122–137. Springer, Heidelberg (2010)
22. Reichert, M., Rinderle, S., Kreher, U., Acker, H., Lauer, M., Dadam, P.: ADEPT2 - next generation process management technology. In: Proceedings Fourth Heidelberg Innovation Forum, Aachen, D.punkt Verlag (April 2007)
23. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. IEEE Computer 39(2), 25–31 (2006)
24. Sirin, E., Parsia, B., Grau, B., Kalyanpur, A., Katz, Y.: Pellet: A practical OWL-DL reasoner. Web Semantics: Science, Services and Agents on the World Wide Web 5(2), 51–53 (2007)
25. Tetlow, P., Pan, J.Z., Oberle, D., Wallace, E., Uschold, M., Kendall, E.: Ontology driven architectures and potential uses of the semantic web in systems and software engineering. In: History, W3C, pp. 1–17 (2006)
26. van der Aalst, W.M.P., ter Hofstede, A.H.M.: YAWL: yet another workflow language. Information Systems 30(4), 245–275 (2005)
27. Wouters, B., Deridder, D., Paesschen, E.V.: The use of ontologies as a backbone for use case management. In: European Conference on Object-Oriented Programming (ECOOP 2000). Workshop: Objects and Classifications, a Natural Convergence (2000)

Relaxing Claims: Coping with Uncertainty While Evaluating Assumptions at Run Time

Andres J. Ramirez[1], Betty H.C. Cheng[1], Nelly Bencomo[2], and Pete Sawyer[2,3]

[1] Michigan State University, East Lansing, Michigan 48823
{ramir105,chengb}@cse.msu.edu

[2] INRIA Paris - Rocquencourt, 78153 Le Chesnay, France
nelly@acm.org

[3] Lancaster University, InfoLab 21, Lancaster, UK LA12WA
sawyer@comp.lancs.ac.uk

Abstract. Self-adaptation enables software systems to respond to changing environmental contexts that may not be fully understood at design time. Designing a dynamically adaptive system (DAS) to cope with this uncertainty is challenging, as it is impractical during requirements analysis and design time to anticipate every environmental condition that the DAS may encounter. Previously, the RELAX language was proposed to make requirements more tolerant to environmental uncertainty, and Claims were applied as markers of uncertainty that document how design assumptions affect goals. This paper integrates these two techniques in order to assess the validity of Claims at run time while tolerating minor and unanticipated environmental conditions that can trigger adaptations. We apply the proposed approach to the dynamic reconfiguration of a remote data mirroring network that must diffuse data while minimizing costs and exposure to data loss. Results show RELAXing Claims enables a DAS to reduce adaptation costs.

1 Introduction

Dynamically adaptive systems (DASs) are systems built to continuously monitor their environment and then adapt their behavior according to changing environmental conditions. [4]. The inherent uncertainty associated with the operational environment of a DAS is challenging as it is often impractical to anticipate every environmental condition that a DAS will encounter throughout its lifetime [4,30]. Previously, the RELAX [30] requirements specification language was proposed to make requirements more tolerant to environmental uncertainty, and Claims [28,29] were applied as markers of uncertainty to record the rationale for a decision made with incomplete information in a DAS. This paper integrates both techniques to assess the validity of Claims at run time while tolerating minor and unanticipated environmental conditions that can otherwise trigger adaptations.

Certain properties about the DAS or its execution environment might not be known until run time. This uncertainty forces developers to make assumptions about the design or configuration of the system. A Claim [28,29] can be used at design time to document and analyze assumptions about how a DAS achieves its goals in different operational contexts. For example, a Claim can be used to document that data should be encrypted since a network might not be secure. A Claim can also be monitored at run time to prove or disprove its validity [28], thereby triggering an adaptation to reach more desirable system configurations if necessary. Nevertheless, Claims are also subject to uncertainty, in the form of unanticipated environmental conditions and unreliable monitoring information, that can adversely affect the behavior of the DAS if it spuriously falsifies a Claim.

This paper proposes the RELAXation of a Claim in recognition that environmental uncertainty may prevent a DAS from categorically proving or disproving a Claim at run time. Specifically, our approach uses RELAX operators to introduce a fuzzy logic layer upon the evaluation criteria that establishes a Claim's validity. Ideally, RELAXing a Claim should enable a DAS to tolerate environmental uncertainty that may otherwise mistakenly disprove a Claim. In this manner, RELAXing a Claim may also reduce adaptation costs for a DAS by preventing frequent, and perhaps unnecessary, adaptations and reconfigurations of its goal realization strategies at run time.

In this paper we propose a stepwise process for RELAXing Claims. First, sources of uncertainty that can disprove the validity of a Claim must be identified. Next, a Claim *applicability metric* must be derived to compute the veracity of a Claim at run time. Both ordinal and temporal RELAX operators can be applied to relax the constraints that define this applicability metric. At run time, if the value produced by a RELAXed Claim applicability metric drops below a predetermined threshold, then the value of the corresponding contribution link must be updated and, if necessary, the system may have to reconfigure towards a different goal realization strategy depending on the current set of valid Claims.

We assess the effectiveness of RELAXing Claims by applying the proposed approach to an industry-provided remote data mirroring network [11,12]. Remote mirroring is a technique that improves data protection and availability by replicating and distributing data to all nodes within a network, such as that used for cable television servers. We RELAX Claims that captures sources of uncertainty that can trigger network reconfigurations in response to adverse conditions, such as network link failures and dropped network messages. The remainder of this paper is organized as follows. Section 2 provides background information on remote data mirroring, goal-oriented requirements modeling, RELAX, and Claims. Section 3 presents the proposed approach for RELAXing Claims. Next, Section 4 presents experimental results. Section 5 describes related work. Lastly, Section 6 summarizes results and overviews future directions.

2 Background

This section provides background information on remote data mirroring, goal modeling, the RELAX requirements specification language, and Claims.

2.1 Remote Data Mirroring

Remote data mirroring (RDM) [11,12] is a data protection technique that stores copies of data at physically isolated locations to protect data against loss, unavailability, or corruption. An RDM can be configured in terms of the network's topology, such as a minimum spanning tree, as well as how data is distributed among data servers. For instance, synchronous propagation distributes data whenever it is modified. In contrast, asynchronous propagation batches data modifications, ideally enabling multiple edits to the same data to be coalesced. Each configuration provides different levels of data protection, performance, and cost. That is, while synchronous propagation provides better data protection than asynchronous propagation, it incurs a network performance penalty as every change must be distributed across the network. Asynchronous propagation provides better network performance than synchronous propagation but it also provides a weaker form of data protection because batched data could be lost in the event of a site failure.

2.2 Goal-Oriented Requirements Engineering and Modeling

A goal declaratively specifies the objectives and constraints that a system-to-be and its execution environment must satisfy [14]. A key objective in goal-oriented requirements engineering (GORE) is to systematically elicit, analyze, and refine high-level goals into finer-grained goals. While goals that represent required functional properties can be evaluated in an absolute manner, a special category of goals called soft goals can only be *satisficed* [5] or satisfied to a certain degree. Soft goals typically represent non-functional properties (e.g., performance) that constrain how functionality should be delivered to stakeholders.

A goal-oriented requirements model provides a graphical framework for capturing relationships between goals. Formally, a goal-oriented requirements model is a directed acyclic graph where a node represents a goal and an edge represents a specific type of refinement. For instance, the i* framework [31] provides an agent-oriented approach for modeling strategies of multiple actors within social contexts. In i*, a Strategic Rational (SR) model captures how a system's configuration addresses the interests and concerns of an actor.

The i* SR goal model in Figure 1 captures the following soft goals for the RDM application: "Minimize Operational Expenses", "Maximize Data Reliability", and "Maximize Network Performance". In order to satisfice these soft goals, the RDM must achieve functional goals such as constructing a connected network and distributing data. These functional goals can be achieved through alternative goal realization strategies (modeled in i* as tasks) that include constructing different network topologies, such as a minimum spanning tree or a redundant topology, and changing propagation parameters.

2.3 RELAX Specification Language

RELAX is a specification language that explicitly addresses uncertainty in adaptive systems. In particular, RELAX was developed to identify and declaratively

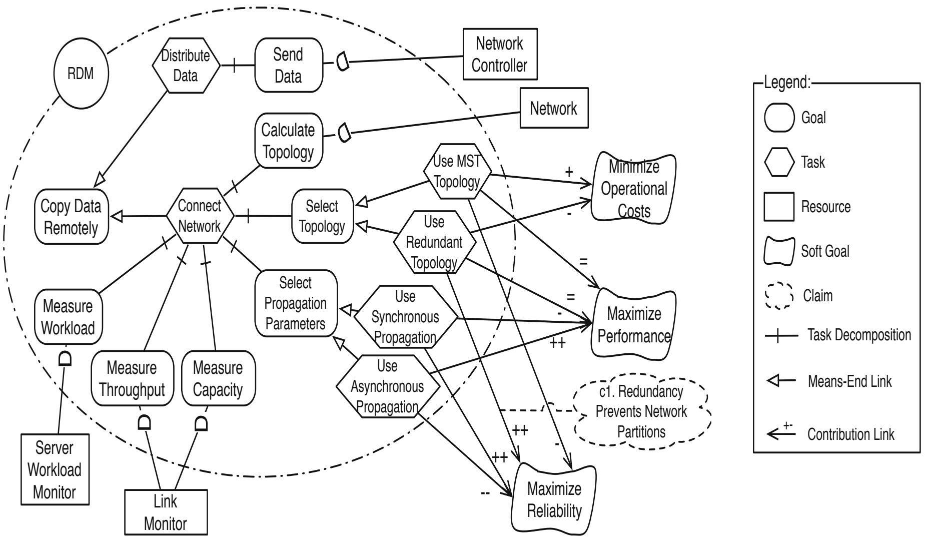


Fig. 1. i* SR goal model for the remote data mirroring application

specify sources of uncertainty that occur at the shared boundary between the system-to-be and its environment [10] . Sources of uncertainty are specified using ENV, MON, and REL elements. ENV specifies properties about the operating context of the DAS, MON lists sensors in the monitoring infrastructure of the DAS that can directly observe and contribute information towards determining the values of ENV properties, and REL defines relationships for computing ENV properties from MON elements.

The RELAX language uses fuzzy logic to express uncertainty in requirements and, by definition, enable developers to systematically design systems that are more flexible and amenable to adaptation. To this end, RELAX provides both ordinal and temporal operators to add flexibility in how and when a functionality may be delivered, respectively. For example, a key requirement in the RDM application states that "The system *shall* distribute new data throughout the network". Environmental uncertainty, in the form of unpredictable link failures and dropped network messages may temporarily hinder the satisfaction of this requirement. RELAXing this requirement to specify that "The system *shall* distribute new data throughout the network AS EARLY AS POSSIBLE" provides temporal flexibility to account for unanticipated events while distributing data. Other RELAX operators include AS LATE AS POSSIBLE, AS CLOSE AS POSSIBLE TO, AS MANY AS POSSIBLE, and AS FEW AS POSSIBLE.

2.4 Claims

Claims were introduced by the NFR framework [5] where they were used to record decision rationale. This role was extended in REAssuRE (REcording of

Assumptions in Requirements Engineering) [29] where Claims are used as *markers of uncertainty* to record the rationale for a decision made with incomplete information in a DAS. REAssuRE showed that Claims are useful where dynamic adaptation is used to maximize the satisficement of a system's soft goals by dynamically selecting between alternative goal operationalizations. The extent to which alternative operationalizations satisfice the system's soft goals under every context cannot always be predicted by the requirements engineer, and Claims serve to record this uncertainty. At run time, Claims can be monitored to test their veracity. If a Claim holds, then the predicted impact of the operationalization on the soft goal is assumed to hold too. If the Claim is falsified by evidence collected by run-time monitoring then the predicted impact of the operationalization on the soft goal is considered unsound and the optimal configuration of operationalizations must be re-evaluated.

In i*, Claims are attached to contribution links, whose values represent the predicted degree of satisficement provided by a task at one end of the link and a soft goal on the other end. In Figure 1, Claim *c1* asserts that the operationalization strategy "Use Redundant Topology" has a strongly positive (++) effect on satisficement of the soft goal "Maximize Reliability". If a Claim proves to be false, such as if RDM nodes were vulnerable to common-cause failures, then the value of the contribution link to which it is attached should be revised to reflect the observed reality. When this happens, the goal model needs to be updated and dynamically re-evaluated. A reconfiguration is triggered if re-evaluation determines that an alternative goal operationalization exists that has a greater predicted net impact on the soft goals.

This automatic determination of the best operationalization is as follows:

Let the function *satisfices* represent the contribution value for a task, soft goal pair:

$$\textit{satisfices:}\;\; T \times SG \rightarrow C$$

where T is the set of tasks, SG is the set of soft goals, and C is the set of possible contribution values {*- -, -, =, +, ++*}. These as are interpreted as corresponding to the range of integer values {*-2, -1, 0, 1, 2*}, respectively. Moreover, i is an index in the set of tasks that represent alternative operationalization of goal g, and t_{ig} is thus one of these tasks.

The task selected as the operationalization strategy for goal g is the one with the net greatest value of contribution link values for all of the soft goals it influences. This is given by the following weighted sum formula in which w represents the relative priority of each softgoal, given as a numeric weight. Note that for simplicity in the remote data monitoring example, we treat each softgoal as being of equal priority, so $w = 1$ in all cases.

$$max_i \sum_{sg \in SG} w_{sg} satisfices(t_{ig}, sg) \qquad (1)$$

In the RDM example, if falsification of *c1* meant that the contribution link value linking "Use Redundant Topology" and "Maximize Reliability" reduced to neutral (=), then taking the aggregate of the contribution link values of "Use

Redundant Topology" and its alternative "Use MST Topology" over the three soft goals, the latter would emerge as the better solution because its set of contribution link values (+,=,-) has a higher net contribution value than that of "Use Redundant Topology" (-,=,=). This evaluation would trigger an adaptation to replace the network configuration that implements "Use Redundant Topology" with "Use MST Topology".

3 Approach for RELAXing Claims

Next we describe how to RELAX a Claim and illustrates with an example.

3.1 Motivation for RELAXing Claims

Figure 2 shows a Claim refinement model that captures assumptions for why a redundant network topology contributes positively to the "Maximize Reliability" soft goal in Figure 1. Although the underlying assumptions might seem reasonable, they are subject to system and environmental uncertainty. In particular, Claims *c4* and *c5* state that link faults do not partition the network nor do they occur coincidentally, respectively. Thus, if two or more network links fail simultaneously then top-level Claim *c1* becomes automatically falsified.

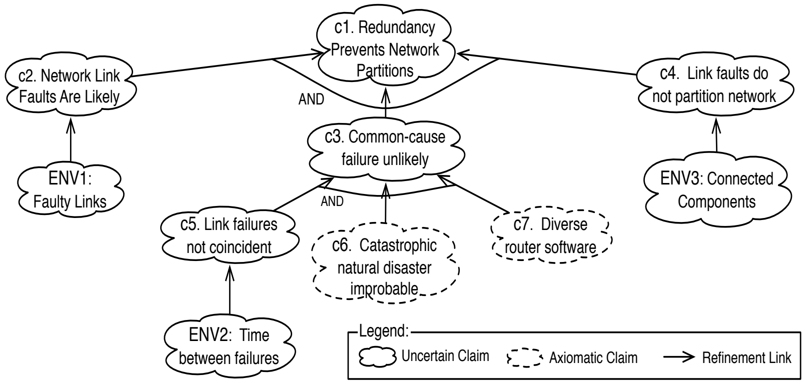

c1. Redundancy Prevents Network Partitions
c2. Network Link Faults Are Likely
AND
c4. Link faults do not partition network
c3. Common-cause failure unlikely
ENV1: Faulty Links
ENV3: Connected Components
AND
c5. Link failures not coincident
c6. Catastrophic natural disaster improbable
c7. Diverse router software
ENV2: Time between failures
Legend:
Uncertain Claim
Axiomatic Claim
Refinement Link

Fig. 2. A Claim refinement model describing why redundancy improves reliability in RDM application

In this scenario, it is *possible* for multiple network link failures to occur simultaneously while the network remains connected. Although a redundant network topology prevents the network from becoming disconnected in this scenario, the top-level Claim *c1* would become disproven by the simultaneous failure of two or more network links. The objective of introducing RELAX operators into the specification of a Claim is to lessen the thresholds or bounds that define the veracity of the Claim itself. Ideally, RELAXing a Claim prevents transient and unanticipated environmental conditions from unnecessarily disproving the validity of a Claim at run time and thus triggering consequential adaptations.

3.2 Process for RELAXing Claims

Figure [3] presents a data flow diagram that overviews the Claim RELAXation process. We describe each step in the Claim RELAXation process in detail:

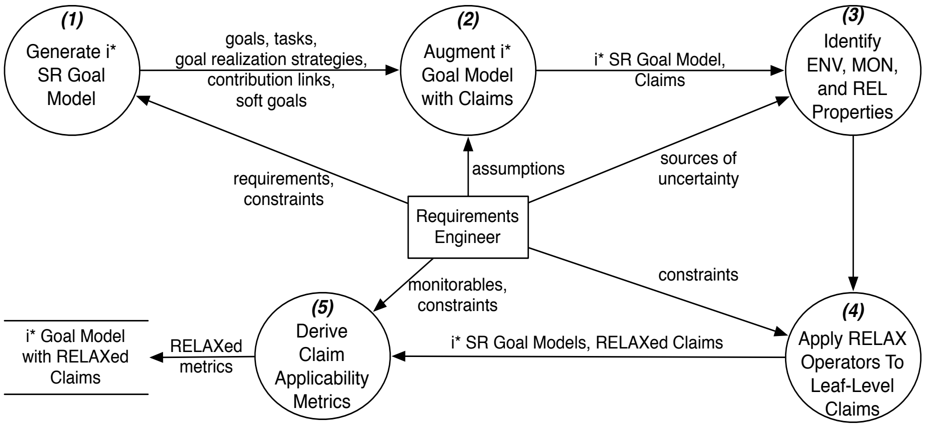

(1)
Generate i* SR Goal Model
goals, tasks, goal realization strategies, contribution links, soft goals
(2)
Augment i* Goal Model with Claims
i* SR Goal Model, Claims
(3)
Identify ENV, MON, and REL Properties
requirements, constraints
assumptions
sources of uncertainty
Requirements Engineer
monitorables, constraints
constraints
(4)
Apply RELAX Operators To Leaf-Level Claims
i* SR Goal Models, RELAXed Claims
(5)
Derive Claim Applicability Metrics
RELAXed metrics
i* Goal Model with RELAXed Claims

Fig. 3. Data flow diagram describing process for RELAXing Claims

(1) Generate i* SR Goal Model. Given a set of requirements and constraints, a requirements engineer must generate an i* SR goal model to capture the goals, tasks, resources, and soft goals of the system-to-be. It must also specify how the alternative goal realization strategies affect the contribution links of soft goals. As an example, the i* SR goal model in Figure [1] captures the various goal realization strategies that the RDM application can use to replicate data across the network, as well as how these strategies affect each soft goal.

(2) Augment i* Goal Model with Claims. The i* SR goal model generated in (1) must be augmented with Claims [29] to document uncertain assumptions about how goal realization strategies affect the contribution links of soft goals. To this end, a Claim refinement model can be used to specify a set of assumptions that collectively support the veracity of a top-level Claim. In a Claim refinement model, certain low-level Claims can be considered axiomatic and need not be monitored at run time. In contrast, evidence for or against the assumption of an uncertain Claim has to be collected at run time [21].[1]

Figure [2] presents a Claim refinement model for the RDM application. The top level Claim in this model, *c1* states that redundancy prevents network partitions, and thus a redundant topology helps the "Maximize Reliability" soft goal. The validity of this top-level Claim is based on the validity of three sub

[1] As an optional step, a requirements engineer can specify how the value of a contribution link should be updated if an attached Claim is falsified, otherwise the value will default to neutral ("=").

Claims. Namely, redundancy prevents network partitions because, while network link faults are likely (*c2*), common cause failures are unlikely (*c3*), and network link faults do not partition the network (*c4*). Similarly, Claim *c3* is refined into additional assumptions that collectively state that common-cause link failures are unlikely because link failures tend to not coincide (*c5*), catastrophic natural disasters are improbable (*c6*), and routers use diverse software systems (*c7*). At run time, Claims *c2*, *c4*, and *c5* must be monitored to prove or disprove their validity. In contrast, Claims *c6* and *c7* are considered axiomatic and need not be monitored.

(3) Identify ENV, MON, and REL Properties. A requirements engineer must identify ENV, MON, and REL properties for each leaf-level non-axiomatic Claim in the Claim refinement model. Furthermore, each ENV property must be mapped to its corresponding MON elements. Specifically, an ENV property that a DAS can directly observe with its sensors can be expressed solely by MON elements. In contrast, an ENV property that a DAS cannot directly observe with its sensors must be indirectly inferred via an REL relationship that might comprise MON elements, as well as constraints and algorithms.

Figure 2 presents three ENV properties associated with leaf-level Claims in the Claim refinement model. Within these leaf-level Claims, *ENV1* refers to the number of faulty links in the RDM network, *ENV2* measures the time between any two link faults, and *ENV3* captures whether the network is connected or partitioned. Since *ENV1* is a property that can be directly observed by LinkMonitor sensors, it can be evaluated as follows: $ENV1 = \sum_{i=1}^{|\text{Links}|} \text{faulty(i)}$, where |Links| is the number of links in the RDM network, and faulty(i) returns 1 if the ith link has failed and 0 otherwise. *ENV3*, on the other hand, cannot be directly observed by sensors and must instead be computed algorithmically by aggregating information from available MON elements. In particular, *ENV3* represents the number of partitions in the RDM network that must be computed with a reachability algorithm [25] that examines which RDM nodes can be reached by traversing active network links.

(4) Apply RELAX Operators to Leaf-Level Claims. A requirements engineer must identify sources of uncertainty that can affect ENV properties associated with each non-axiomatic leaf-level Claim in a Claim refinement model. In addition, a requirements engineer must also determine which of these ENV properties can be safely RELAXed without affecting the satisfaction of invariant goals and requirements. If an ENV property can temporarily deviate from its expected value, then a corresponding RELAX operator must be applied to specify how its value can vary due to environmental uncertainty. Once a RELAX operator is applied, bounds must be established to constrain the extent to which the ENV property can vary without unnecessarily disproving a Claim.

As an example, consider that while Claim *c2* states that network link failures are common, unreliable monitoring information that fails to detect link failures can disprove this Claim. Nevertheless, the validity of Claim *c1* should not be necessarily disproven because no link failures are observed. As such, Claim *c2* can be RELAXed to state that it SHALL hold when UNTIL n units of time have passed without link failures. Likewise, Claim *c4* can be RELAXed to state that it SHALL hold when the number of connected components in the network is AS CLOSE AS POSSIBLE TO one when network link failures occur. Lastly, Claim *c5* can be RELAXed to state that it SHALL hold when network links fail at the same time AS CLOSE AS POSSIBLE TO never.

(5) Derive Claim Applicability Metrics. A requirements engineer must derive a *Claim applicability metric* for each non-axiomatic leaf-level Claim in the Claim refinement model. Specifically, a Claim applicability metric uses fuzzy logic functions (i.e., triangle, trapezoid) to evaluate monitoring information and assess the validity of a Claim at run time. The peak of a Claim applicability metric function represents the ideal case (i.e., when the assessment is solidly true), and the respective tails are still acceptable but not optimal. As input, each Claim applicability metric accepts monitoring information from sensors in MON. If necessary, MON values may need to be mapped to ENV properties through REL relationships. As output, a Claim applicability metric generates a numerical value, between zero and one, that is proportional to the Claim's validity.

Figure 4 shows three RELAXed Claim applicability metrics that can be used to evaluate the validity of Claims *c2*, *c4*, and *c5*. As this figure illustrates, the applicability metric for Claim *c2* returns values close to one as the number of faulty links increases, thereby validating the assumption that network link faults are likely to occur. Similarly, the applicability metric for Claim *c4* returns one when the number of connected components in the RDM network equals the desired number of connected components. Since a connected network has exactly one connected component, then this applicability metric serves to validate the assumption that link failures do not partition the network. Lastly, the Claim applicability metric for Claim *c5* returns values close to one as the time between link failures increases, thereby validating the assumption that network link faults are not coincidental.

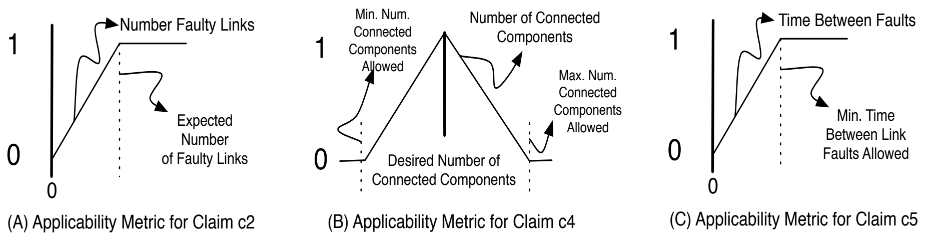


Fig. 4. RELAXed Claim applicability metrics for Claims c2, c4, and c5

Since each Claim applicability metric is associated with a leaf-level Claim, its value must be propagated upwards through the Claim refinement model in order to compute the validity of the top-level Claim. Depending on the type of the Claim refinement link, two methods can be used to compute the applicability value of a parent Claim. While the applicability value of an AND-refined Claim is equal to the minimum value of each sub-Claim, the applicability value of an OR-refined Claim is equal to the maximum value of each sub-Claim. As an example, assume that the applicability values of Claims *c2*, *c3*, and *c4* in Figure 2 are equal to 0.92, 0.89, and 0.98 respectively. In this scenario, the applicability value of top-level Claim *c1* equals 0.89, the minimum value of all AND-refined Claims.

3.3 Adapting in Response to Falsified Claims

A RELAXed Claim is falsified when its Claim applicability value drops below a predetermined threshold. When a Claim is falsified, the value of the contribution link to which it is attached must be updated to indicate that a given goal realization strategy does not necessarily address current environmental conditions. While the value of a contribution link could be inverted (e.g., converting a "+" to a "-"), this approach would correspond to an arbitrary transformation that might not hold either. In the absence of new evidence, a DAS cannot assume anything about the value of the contribution link except that the original value is probably wrong. Thus, the value of a contribution link of a falsified Claim becomes neutral and undefined, regardless of its original polarity or strength. Alternatively, a requirements engineer can override these default values on a per-Claim basis if desired (see footnote in Section 3.2, Step (2)).

Once the value of a contribution link is updated, the DAS must re-evaluate the entire set of goal realization strategies to determine whether it should self-reconfigure towards a more suitable goal realization strategy. In this manner, run-time updates to the goal model drives the self-reconfiguration.

4 Experimental Results

This section describes experimental results for demonstrating the utility of RELAXing Claims.

4.1 Experimental Setup

In the following experiments, each RDM must replicate and distribute new data to all other RDMs. We modeled and implemented an RDM network as a completely connected undirected graph where each node and edge represents an RDM or a network link, respectively. The network itself comprises 25 RDMs and 300 network links that can be activated to distribute data. The operational characteristics of each RDM node and network link, such as workload or capacity, were generated from a random uniform distribution based on RDM operational models previously presented by Keeton *et al.* [11,12]. Throughout the simulation, approximately 25 new data items were randomly inserted at different RDMs from which they had to be efficiently distributed across the network.

The RDM network might self-adapt at run time in response to adverse environmental conditions, such as link failures, repeatedly dropped messages, or unreliable monitoring data. To this end, each RDM implements the dynamic change management (DCM) protocol introduced by Kramer and Magee [13] such that it may reach passive and quiescent states in bounded time. An RDM in a passive state can accept data distributed by other RDMs but may not replicate nor distribute data itself. Similarly, a quiescent RDM can neither distribute nor accept data from other RDMs since both its neighbors and itself are in passive states.

In addition, we implemented a rule-based adaptation engine that leverages Claims constructs as conditionals for adaptation. Specifically, the applicability of each Claim is monitored and evaluated at run time. If a Claim becomes falsified, then the value of its contribution link is updated to reflect this new information. Next, the set of goal realization strategies are re-evaluated to select the one that best addresses soft goals given current system and environmental conditions (see Section 2.4). If an adaptation is required, then a target configuration is selected and the DCM protocol is executed to generate an adaptation path that safely transitions the executing RDM network from its source to its target configuration.

The first experiment explores how a DAS can leverage RELAXed Claims to trigger adaptations at run time, and the second experiment explores how RELAXing Claims can reduce adaptation costs in uncertain environments. For statistical purposes, we conducted 40 trials of each experiment. Where applicable, we plot the mean values with corresponding error bars.

4.2 No Environmental Uncertainty

This experiment evaluates how a falsified RELAXed Claim can trigger the run-time reconfiguration of a DAS. For this scenario, the values of the contribution links in Figure 1 suggest that the best goal realization strategy is to use a redundant network topology with synchronous propagation. As such, Figure 5(A) depicts the initial configuration comprising 25 RDMs using synchronous propagation and 32 active network links, 8 of them redundant. This configuration is based on the validity of Claim *c1* (see Figure 2) that states that a redundant network topology prevents network link failures from partitioning the network. Nevertheless, for this scenario we disabled adverse environmental conditions to purposefully cause the falsification of this Claim and thereby enable us to evaluate how the RDM network self-reconfigures in response.

Since network failures are not possible in this scenario, the applicability of Claim *c2*, which states that network link faults are likely, gradually decreases until it falsifies top-level Claim *c1*. The falsification of Claim *c1* states that, for the given set of system and environmental conditions, the "Use Redundant Topology" goal realization strategy does not necessarily prevent network partitions and therefore does not contribute as positively ("++") to the "Maximize Reliability" soft goal. As a result, the value of the corresponding contribution link in Figure 5(B) changes from "++" to "=" to reflect reduced confidence

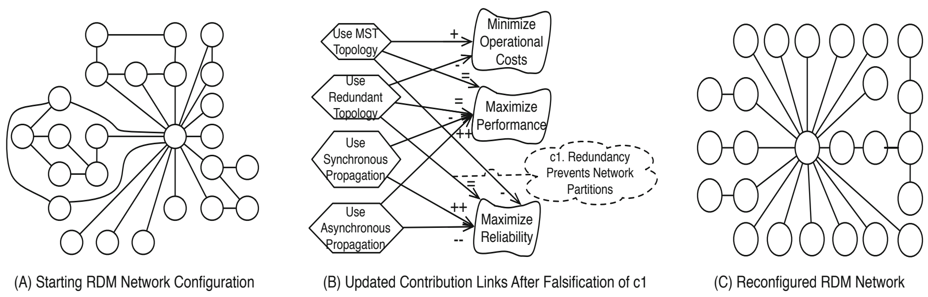

Use MST Topology
Use Redundant Topology
Use Synchronous Propagation
Use Asynchronous Propagation
Minimize Operational Costs
Maximize Performance
Maximize Reliability
c1. Redundancy Prevents Network Partitions
+
-
=
++
--
(A) Starting RDM Network Configuration
(B) Updated Contribution Links After Falsification of c1
(C) Reconfigured RDM Network

Fig. 5. Adaptation progression in response to falsified Claim c1

that a network topology contributes positively to the "Maximize Reliability" soft goal.

Once the value of the contribution link is updated, the DAS re-evaluates its goal operationalizations. As Figures 1 and 5(B) illustrate, once Claim *c1* is falsified, the best goal realization strategy becomes to use a minimum spanning tree (MST) topology with synchronous propagation. The reconfigured RDM network, shown in Figure 5(C) comprises 25 RDMs propagating data synchronously through 24 active network links. Combined, a MST minimizes operational costs while synchronous propagation maximizes network reliability. Note that while asynchronous propagation would also maximize performance, it is not the optimal solution for this scenario since its advantages are offset by contributing equally negatively against the "Maximize Reliability" soft goal.

4.3 Environmental Uncertainty

This experiment compares the benefits, in terms of adaptation costs, that RELAXed Claims can provide over traditional Claims in uncertain environments. For this work, we define adaptation costs based on the adaptation quality property of *settling time et al.* [26]. Specifically, we measure adaptation costs by counting the number of components placed in passive and quiescent states during adaptation. Thus, the null hypothesis, H_0, states that there is no difference in adaptation costs between RELAXed Claims and traditional Claims. Similarly, the alternative hypothesis, H_1, states that RELAXed Claims will incur lower adaptation costs than traditional Claims. The rationale for H_1 is that a RELAXed Claim is more tolerant to uncertain environmental conditions that might otherwise disprove a traditional Claim and thus trigger an unnecessary adaptation.

As in the previous experiment, the RDM is initially configured to use a redundant network topology and synchronous propagation. However, while the previous experiment did not introduce any forms of environmental uncertainty into the RDM, we now randomly kill network links, drop messages, and either delay or introduce noise into monitoring information. These types of environmental conditions are intended to affect the applicability of Claims *c3* and *c4*, as well as the applicability of top-level Claim *c1*.

Figure 6 presents a bar chart with the various adaptation cost metrics tracked throughout each experimental simulation, where error bars denote statistical significance where applicable. Specifically, the first two bars count the mean number of adaptations performed in each trial. The remaining bars count the mean number of components in passive and quiescent mode throughout these adaptations, respectively. As this plot illustrates, RELAXing Claims significantly reduces the number of adaptations triggered by environmental uncertainty, and also significantly reduces the number of components placed in passive mode during adaptations ($p < 0.05$). Although statistically insignificant, this plot also illustrates a decreasing trend in the number of components that are placed in quiescent mode during adaptations, thus minimizing the impact on system functionality even in the face of environmental uncertainty.

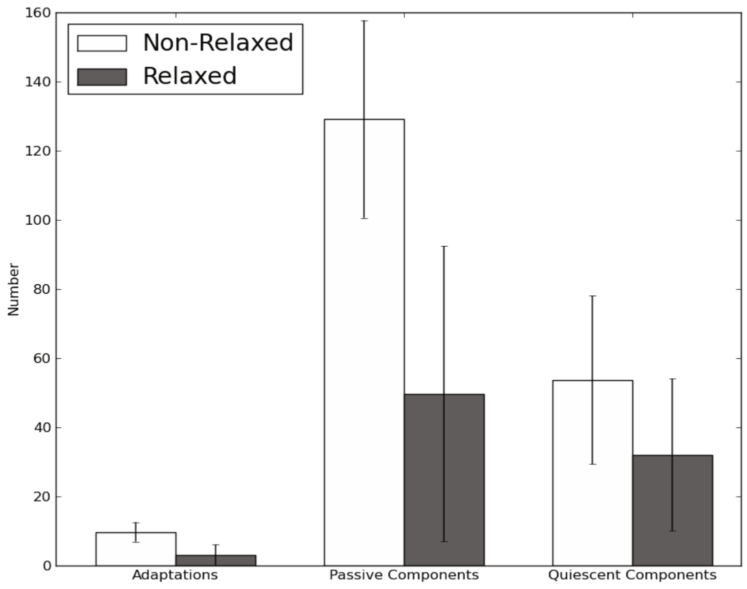


Fig. 6. Comparing adaptation costs between RELAXed Claims and traditional Claims

Given the differences in adaptation costs between RELAXed Claims and traditional Claims we reject the null hypothesis H_0. Likewise, we accept the alternate hypothesis, H_1, as it concerns both the total number of adaptations performed and the number of components placed in passive mode during adaptations. These differences in adaptation costs are caused primarily by the falsification of Claim $c5$ (see Figure 2). While coincident link failures automatically falsify non-RELAXed Claim $c5$, in most cases it did not disprove its RELAXed Claim counterpart as long as the network remained connected. As such, RELAXing Claims enable the RDM network to reduce the number of adaptations it incurred in response to adverse environmental conditions.

5 Related Work

This section presents related work on model-based approaches to the requirements engineering of a DAS. In particular, this section includes related work on documenting and analyzing obstacles and sources of uncertainty, requirements monitoring, and requirements reflection.

Documenting and Analyzing Obstacles and Sources of Uncertainty. A number of requirements-level techniques have been developed to deal with changing environmental conditions faced by a DAS. For instance, Letier and van Lamsweerde [15] introduced a goal modeling approach for identifying and resolving obstacles that can prevent the satisfaction of a goal. Obstacles are identified by negating requirements and then elaborating preconditions for the obstacles to arise. Letier and van Lamsweerde [17] also introduced a probabilistic framework for specifying the partial satisfaction of goals. While these approaches are intended for functional goals, conceptually their techniques could be applied to identify and evaluate conditions that can falsify assumptions.

Fuzzy logic has been used to represent effects of uncertainty on the satisfaction of goals and requirements. RELAX [4,30] and FLAGS [2] are requirements specification languages for making more flexible the satisfaction criteria of functional goals to cope with uncertainty. RELAX can also be used in goal-modeling for identifying and mitigating sources of uncertainty. Serrano *et al.* [23] introduced an approach for RELAXing the contribution links of soft goals, thereby enabling a DAS to evaluate the satisficement [5] of soft goals while coping with uncertainty. None of these approaches, however, focus on how uncertainty can affect the validity of assumptions at run time – the focus of this paper.

Requirements Monitoring. Traditionally, requirements monitoring frameworks [7,8,20] use monitoring information to trace through state-based models that specify allowed states of the system. This information enables a DAS to identify and address conditions that can prevent the satisfaction of requirements. Utility functions have also been applied to monitor requirements [9,19,27]. Our approach is similar to these approaches in that it is intended to detect and respond to run-time conditions that prevent a DAS from satisfying its objectives. Instead of directly monitoring requirements, however, our approach monitors Claims to detect falsified assumptions that can obstruct goals. Both types of techniques can and should be used in conjunction at run time.

Requirements Reflection. Sawyer *et al.* [22] suggested that requirements should be run-time entities about which can be reasoned to determine their level of satisfaction and used to support adaptation decisions. Our approach adopts this view in that it uses Claims to reason about requirements at run time. GMoDS [6] also maintains run-time representations of goals, where a goal can transition between achieved, failed, obviated, or removed states. In contrast to our approach, GMoDS does not perform reasoning about goal satisfaction.

Another approach by Chen *et al.* [3] maintains run-time goal models to reason about tradeoff decisions aimed at achieving *survivability assurance.* As with our approach, their live goal models postpone the necessary quality tradeoff decisions until run time. Unlike our approach, however, their approach deals with functional goals, selectively allowing them to become disconnected from the goal model to spare resources. Awareness Requirements (AwReqs) [16,24] refer to the success or failure of other requirements, providing information on run-time divergence from specified requirements as a feedback loop in an explicitly control-theoretic model of requirements-driven dynamic adaptation. Although AwReqs do not focus on detecting run-time divergence from assumptions, exploring the management of requirements and assumptions at run time would be interesting.

Ali *et al.* [1] present a goal-based framework for modeling and analyzing contextual requirements. Their approach shares similarities with our use of Claims, where context plays a similar role to the domain assumptions that are represented with Claims. While context analysis is similar to Claim refinement, any connection between context and monitoring is implicit.

6 Conclusions

Recent research in RE has applied dynamic adaptation to mitigate uncertainty about the environmental contexts that a system-to-be may encounter at run time [2,3,4,18,28,29,30]. This paper has presented an approach to account for system and environmental uncertainty by RELAXing Claims when there is uncertainty about the evidence for or against a Claim's truth. We argue that failing to recognize both the fallibility of monitoring information, as well as the transient effects of minor environmental disturbances, are likely to cause erratic behavior in a DAS, and may risk unnecessary, and perhaps costly, adaptations. We evaluated our Claim RELAXation technique by applying it to an industry-provided case study of RDM system. Experimental results performed on an RDM simulator show that RELAXing Claims enables a DAS to reduce the number of adaptations when compared to traditional non-RELAXed Claims.

Future work will explore several open issues raised by our investigations. First, we will continue to explore how RELAX can be used to tolerate uncertainty from the environment and the monitoring infrastructure, as RELAX is applied to Claims and functional goals. Next, we are interested in making the propagation of a failed Claim to the goal model more flexible. Currently, a failed Claim results in loss of trust in the predicted degree of soft goal satisficement. Finally, we will explore how the RELAXation process can be more automated.

Acknowledgements. This work has been supported in part by NSF grants CCF-0541131, IIP-0700329, CCF-0750787, CCF-0820220, DBI-0939454, CNS-0854931, Army Research Office grant W911NF-08-1-0495, Ford Motor Company. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation, Army, Ford, or other research sponsors.

This research is also partially supported by the Marie Curie Fellowship Requirements@run.time and the EU CONNECT project.

References

1. Ali, R., Dalpiaz, F., Giorgini, P.: A goal-based framework for contextual requirements modeling and analysis. Requir. Eng. 15, 439–458 (2010)
2. Baresi, L., Pasquale, L., Spoletini, P.: Fuzzy goals for requirements-driven adaptation. In: Proceedings of the 18th IEEE International Requirements Engineering Conference, pp. 125–134. IEEE, Sydney (2010)
3. Chen, B., Peng, X., Yu, Y., Zhao, W.: Are your sites down? requirements-driven self-tuning for the survivability of web systems. In: 19th International Conference on Requirements Engineering (2011)
4. Cheng, B.H.C., Sawyer, P., Bencomo, N., Whittle, J.: A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 468–483. Springer, Heidelberg (2009)
5. Chung, L., Nixon, B., Yu, E., Mylopoulos, J.: Non-Functional Requirements in Software Engineering. Kluwer Academic Publishers (2000)
6. DeLoach, S.A., Miller, M.: A goal model for adaptive complex systems. International Journal of Computational Intelligence: Theory and Practice 5(2) (2010)
7. Feather, M.S., Fickas, S., van Lamsweerde, A., Ponsard, C.: Reconciling system requirements and runtime behavior. In: Proc. of the 8th Int. Workshop on Software Specification and Design, Washington, DC, USA, pp. 50–59 (1998)
8. Fickas, S., Feather, M.S.: Requirements monitoring in dynamic environments. In: Proc. of the Second IEEE Int. Symp. on Requirements Eng., Washington, DC, USA, pp. 140–147 (1995)
9. de Grandis, P., Valetto, G.: Elicitation and utilization of application-level utility functions. In: The Proceedings of the Sixth International Conference on Autonomic Computing (ICAC 2009), pp. 107–116. ACM, Barcelona (2009)
10. Jackson, M., Zave, P.: Deriving specifications from requirements: an example. In: Proceedings of the 17th International Conference on Software Engineering, pp. 15–24. ACM, Seattle (1995)
11. Ji, M., Veitch, A., Wilkes, J.: Seneca: Remote mirroring done write. In: USENIX 2003 Annual Technical Conference, pp. 253–268. USENIX Association, Berkeley (2003)
12. Keeton, K., Santos, C., Beyer, D., Chase, J., Wilkes, J.: Designing for disasters. In: Proceedings of the 3rd USENIX Conference on File and Storage Technologies, pp. 59–62. USENIX Association, Berkeley (2004)
13. Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. IEEE Trans. on Soft. Eng. 16(11), 1293–1306 (1990)
14. van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley (March 2009)
15. van Lamsweerde, A., Letier, E.: Handling obstacles in goal-oriented requirements engineering. IEEE Tran. on Soft. Eng. 26(10), 978–1005 (2000)
16. Lapouchnian, A.: Exploiting Requirements Variability for Software Customization and Adaptation. Ph.D. thesis, University of Toronto (2011)
17. Letier, E., van Lamsweerde, A.: Reasoning about partial goal satisfaction for requirements and design engineering. In: Proc. of the 12th ACM SIGSOFT Int. Symp. on Foundations of Software Eng., pp. 53–62. ACM, Newport Beach (2004)
18. Ramirez, A.J., Cheng, B.H.C.: Cheng: Adaptive monitoring of software requirements. In: Proceedings of the IEEE Workshop on Requirements at Run Time, pp. 41–50. RE@RunTime, Sydney, Australia (September 2010)

19. Ramirez, A.J., Cheng, B.H.C.: Automatically deriving utility functions for monitoring software requirements. In: Proceedings of the 2011 International Conference on Model Driven Engineering Languages and Systems Conference, Wellington, New Zealand, pp. 501–516 (2011)
20. Robinson, W.N.: Monitoring software requirements using instrumented code. In: HICSS 2002: Proceedings of the 35th Annual Hawaii International Conference on System Sciences, pp. 276–285. IEEE Computer Society, Hawaii (2002)
21. Salifu, M., Yu, Y., Nuseibeh, B.: Specifying monitoring and switching problems in context. In: IEEE Int. Requirements Engineering Conference, pp. 211–220 (October 2007)
22. Sawyer, P., Bencomo, N., Letier, E., Finkelstein, A.: Requirements-aware systems: A research agenda for re self-adaptive systems. In: Proceedings of the 18th IEEE International Requirements Engineering Conference, Sydney, Australia, pp. 95–103 (September 2010)
23. Serrano, M., Serrano, M., Sampaio, J.C., Leite, P.: Dealing with softgoals at runtime: A fuzzy logic approach. In: Proceedings of the 2nd International Workshop on Requirements at Run Time, Trento, Italy, pp. 23–31 (August 2011)
24. Silva Souza, V.E., Lapouchnian, A., Robinson, W.N., Mylopoulos, J.: Awareness requirements for adaptive systems. Tech. rep., University of Trento (2010)
25. Tarjan, R.: Depth-first search and linear graph algorithms. In: Proceedings of the 12th Annual Symposium on Switching and Automata Theory, pp. 114–121 (October 1971)
26. Villegas, N.M., Müller, H.A., Tamura, G., Duchien, L., Casallas, R.: A framework for evaluating quality-driven self-adaptive software systems. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, Waikiki, Honolulu, HI, USA, pp. 80–89 (May 2011)
27. Walsh, W.E., Tesauro, G., Kephart, J.O., Das, R.: Utility functions in autonomic systems. In: Proceedings of the First IEEE International Conference on Autonomic Computing, pp. 70–77. IEEE Computer Society, New York (2004)
28. Welsh, K., Sawyer, P., Bencomo, N.: Towards requirements aware systems: Runtime resolution of design-time assumptions. In: Proc. of the 26th IEEE/ACM Int. Conf. on Automated Software Engineering, Lawrence, Kansas, USA, pp. 560–563 (November 2011)
29. Welsh, K., Sawyer, P.: Understanding the Scope of Uncertainty in Dynamically Adaptive Systems. In: Wieringa, R., Persson, A. (eds.) REFSQ 2010. LNCS, vol. 6182, pp. 2–16. Springer, Heidelberg (2010)
30. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C., Bruel, J.M.: RELAX: Incorporating uncertainty into the specification of self-adaptive systems. In: Proc. of the 17th Int. Requirements Eng. Conf (RE 2009), Atlanta, Georgia, USA, pp. 79–88 (September 2009)
31. Yu, E.S.: Towards modeling and reasoning support for early-phase requirements engineering. In: Proc. of the Third IEEE Int. Symposium on Requirements Eng., Annapolis, MD, USA, pp. 226–235 (January 1997)

Dynamic Evolution of Context-Aware Systems with Models at Runtime[⋆]

Germán H. Alférez[1] and Vicente Pelechano[2]

[1] Facultad de Ingeniería y Tecnología, Universidad de Montemorelos, Apartado 16-5, Montemorelos N.L., 67500, Mexico
harveyalferez@um.edu.mx

[2] Centro de Investigación en Métodos de Producción de Software (ProS), Universitat Politécnica de Valéncia, Camí de Vera s/n, E-46022, Spain
pele@dsic.upv.es

Abstract. Model-driven techniques have proven to yield significant benefits for context-aware systems. Specifically, semantically-rich models are used at runtime to monitor the system context and guide necessary changes. Under the closed-world assumption, adaptations are fully known at design time. Nevertheless, it is difficult to foresee all the possible situations that may arise in uncertain and complex contexts. In this paper, we present a model-based framework to support the dynamic evolution of context-aware systems to deal with unexpected context events in the open world. If model adaptations are not enough to solve uncertainty, our model-based evolution planner guides the evolution of the supporting models to preserve high-level requirements. A case study about a context-aware Web service composition, which is executed in a distributed computing infrastructure, illustrates the applicability of our framework. A realization methodology and a prototype system support our approach.

1 Introduction

In nature, organisms adapt themselves to be more suitable to an environment. As organisms live in intricate, changing environments, software is executed in complex and heterogeneous computing infrastructures in which a diversity of events may arise (e.g. security threats and server failures). Thus, it is desirable to translate the ideas of adaptation in the natural world to software in order to solve these situations. *Dynamic adaptation* of software behavior refers to the act of changing the behavior of some part of a software system as it executes, without stopping or restarting it [20]. This type of adaptation is particularly important in critical systems that cannot be stopped to implement the adaptations (e.g. software for electronic commerce and banking).

Adaptations are carried out in response to changing conditions in the supporting computing infrastructure and in the surrounding physical environment. Therefore, a requirement for dynamic adaptation is context awareness. The *context* is any information that can be used to characterize the situation of an entity [13]. The information that is

⋆ This work has been developed with the support of MICINN under the project everyWare TIN2010-18011 and co-financed with ERDF.

collected from the context is used as a basis for automating tasks such as installation, adaptation, or healing [8] (i.e., self-* properties associated with autonomic computing).

Several research works have recently proposed the use of models at runtime as a feasible way to guide dynamic adaptations in domains as diverse as service compositions [23,2], mobile devices [24], and home automation [8]. *Models at runtime* can be defined as causally connected self-representations of the associated system that emphasize the structure, behavior, or goals of the system from a problem-space perspective [6]. In response to changes in the context, the system itself can query these models to determine the necessary modifications in the underlying architecture. Therefore, instead of programming complex scripts to describe adaptation actions to change the behavior of the system during execution, easy-to-understand and technology-independent models can be used to express dynamic adaptations.

Nevertheless, most current model-driven approaches for dynamic adaptation still tend to be based on the *closed-world* assumption, in which the boundary between the system and the environment is known beforehand and is unchanging [4]. Under this assumption, models at runtime can remain stable for a long time. However, in the unpredictable *open world*, software should react to continuous and unanticipated changes in complex and uncertain contexts [4]. In order to manage uncertainty in the open world, the supporting models should be able to evolve at runtime for better functioning and system "survival". We define *dynamic evolution* as the process of moving the software to a new version (which cannot be supported by predefined dynamic adaptations) in order to manage unknown context events[1] at runtime.

There is a small number of approaches that manage uncertainty using models at runtime. Some of them focus on analyzing the collected context information using an ontology [26], and others focus on modeling uncertainty at the requirements and design phases for later use at runtime [27,10,17]. Even though these approaches are interesting, the models that are created at design time to deal with uncertainty do not evolve at runtime. Therefore, the capacity of reaction to face new unknown context events decreases because the initial models are unable to support them. Moreover, this situation affects the feasibility of models at runtime as a means to guide the dynamic evolution of critical systems that cannot be stopped to modify the supporting models.

Our contribution is a model-based framework that supports the dynamic evolution of context-aware systems to deal with the uncertainty caused by unexpected context events in the open world. This framework is particularly useful in systems that are built upon service operations or component operations that can be activated or deactivated at runtime depending on contextual situations (e.g. in service-oriented applications that can activate a set of services to protect themselves, or in smart-home software that can start or stop components, such as lighting devices). This framework answers the following questions: 1) Which corrective actions can trigger the dynamic evolution of the system to preserve the expected requirements when unknown problematic context events are faced?; 2) Which requirements can be affected by unknown context events?; and 3) How can the system self-evolve to manage arising unknown context events? The answers to these questions are offered through easy-to-understand and highly-abstract

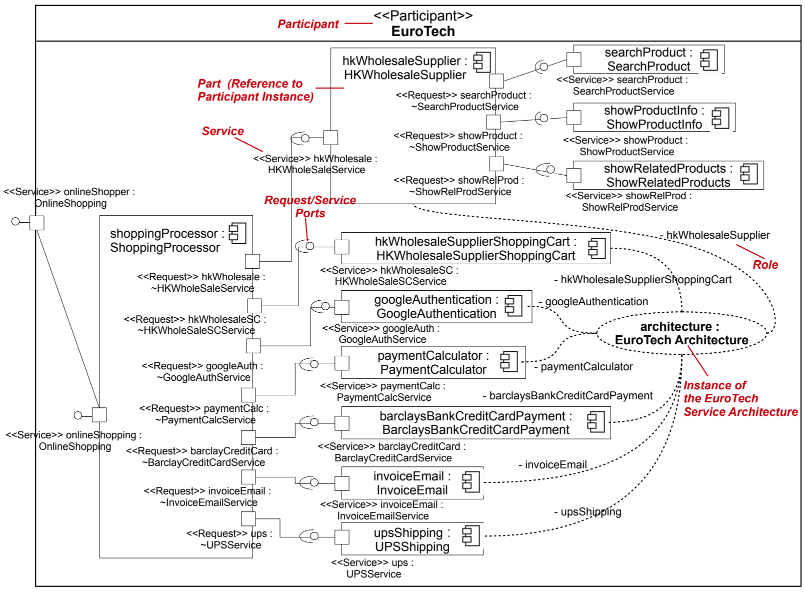
[1] We refer to *unknown context events* as those situations arising in the context that were not foreseen at design time.

models at runtime. In order to exemplify the applicability of our approach, the framework is applied to a typical Web service composition running in a heterogeneous context. A realization methodology and a proof-of-concept prototype are also presented.

The remainder of this paper is structured as follows: Section 2 describes a motivating scenario. Section 3 describes our model-based framework for the dynamic evolution of context-aware systems. Section 4 describes the feasibility of our approach. Section 5 presents related work. Section 6 presents conclusions and future work.

2 Motivating Scenario

In order to illustrate the need for dealing with unexpected context events in the open world, we introduce a critical service composition that supports on-line product shopping at EUROTECH, a multinational retailer of technology products. In Figure 1, the Service oriented architecture Modeling Language (SoaML) [25] is used to design the architecture model for this service composition. The EuroTECH participant connects the SHOPPINGPROCESSOR service provider with other service providers (*participants* are either specific entities or kinds of entities that provide or use services [25]). The collaboration use (called *architecture*) is an instance of the EUROTECH service architecture. This specifies that the SHOPPINGPROCESSOR service provider adheres to that service architecture. The role bindings indicate the roles played by participants.



<<Participant>> EuroTech
Participant
Part (Reference to Participant Instance)
Service
Request/Service Ports
Role
Instance of the EuroTech Service Architecture
hkWholesaleSupplier : HKWholesaleSupplier
searchProduct : SearchProduct
showProductInfo : ShowProductInfo
showRelatedProducts : ShowRelatedProducts
<<Request>> searchProduct : ~SearchProductService
<<Request>> showProduct : ~ShowProductService
<<Request>> showRelProd : ~ShowRelProdService
<<Service>> searchProduct : SearchProductService
<<Service>> showProduct : ShowProductService
<<Service>> showRelProd : ShowRelProdService
<<Service>> hkWholesale : HKWholeSaleService
<<Service>> onlineShopper : OnlineShopping
shoppingProcessor : ShoppingProcessor
hkWholesaleSupplierShoppingCart : HKWholesaleSupplierShoppingCart
<<Service>> hkWholesaleSC : HKWholeSaleSCService
googleAuthentication : GoogleAuthentication
<<Service>> googleAuth : GoogleAuthService
paymentCalculator : PaymentCalculator
<<Service>> paymentCalc : PaymentCalcService
barclaysBankCreditCardPayment : BarclaysBankCreditCardPayment
<<Service>> barclayCreditCard : BarclayCreditCardService
invoiceEmail : InvoiceEmail
<<Service>> invoiceEmail : InvoiceEmailService
upsShipping : UPSShipping
<<Service>> ups : UPSService
<<Request>> hkWholesale : ~HKWholeSaleService
<<Request>> hkWholesaleSC : ~HKWholeSaleSCService
<<Request>> googleAuth : ~GoogleAuthService
<<Request>> paymentCalc : ~PaymentCalcService
<<Request>> barclayCreditCard : ~BarclayCreditCardService
<<Request>> invoiceEmail : ~InvoiceEmailService
<<Request>> ups : ~UPSService
<<Service>> onlineShopping : OnlineShopping
architecture : EuroTech Architecture
- hkWholesaleSupplier
- hkWholesaleSupplierShoppingCart
- googleAuthentication
- paymentCalculator
- barclaysBankCreditCardPayment
- invoiceEmail
- upsShipping

Fig. 1. The EUROTECH participant

The operation for product searching is provided by the SEARCHPRODUCT Web service, which is part of the HKWHOLESALESUPPLIER composite service. The product information is sent to the customer by the SHOWPRODUCTINFO Web service and the information for other related products is listed by the SHOWRELATEDPRODUCTS Web service. Customers can add products to the shopping cart through the HKWHOLESALESUPPLIERSHOPPINGCART Web service. When the customer is ready to checkout, he or she is authenticated by the GOOGLEAUTHENTICATION Web service. The PAYMENTCALCULATOR Web service calculates the total amount to be paid. The payment is done through the BARCLAYSBANKCREDITCARDPAYMENT Web service. Finally, the in-house EMAILINVOICE Web service sends an e-mail to the customer with the invoice and the UPSSHIPPING Web service is invoked to deliver the product.

In order to support the dynamic adaptation of the system to keep this process available 24/7, systems engineers have programmed a set of predefined adaptation actions for specific context events. For instance, if the BARCLAYSBANKCREDITCARDPAYMENT service operation is unavailable, then other service operations can be invoked instead. Nevertheless, implementing scripts with predefined adaptation actions has the following drawbacks: 1) if there are no predefined adaptation actions for a particular context situation, then no adaptation is carried out; and 2) implementing adaptation actions through complex scripts makes it difficult to reason about the system as it grows. These situations help us to identify the following *challenges* for context-aware systems in the open world: 1) context-aware systems should be able to count on corrective actions that trigger the dynamic evolution of the system to preserve the expected requirements when facing unknown context events; 2) the evolution actions to guide the system to a better configuration should be sufficiently expressive and easy-to-understand in order to facilitate the development of the logic behind autonomic management; and 3) the dynamic evolution of the system should be carried out by auto-generated evolution actions in order to avoid human intervention.

3 Model-Based Framework for the Dynamic Evolution of Context-Aware Systems

The dynamic adaptation of context-aware systems is possible by modifying the system's architecture model at runtime through predefined adaptation actions [8,2]. This approach works well under the closed-world assumption. However, predefined adaptation actions are not enough in the open world where several unforeseen context events can arise (e.g. sudden security attacks in a service-oriented system or memory overload in a device that controls lighting in a smart home). These unknown events create uncertainty about the way the system should deal with them (e.g. Should the lighting device be restarted or should a backup device replace it on-the-fly?).

Therefore, we propose the following strategy to manage problematic unknown context events through the dynamic evolution of the system and to meet the three challenges presented in Section 2. First, the corrective actions for dealing with uncertainty are expressed as abstract tactics. *Tactics* are last-resort surviving actions or strategies to preserve the requirements that can be negatively impacted by unknown context events. Therefore, tactics trigger the dynamic evolution of the system to preserve requirements at runtime. Evolution actions are expressed as easy-to-understand models. Also, a model-driven mechanism auto-generates evolution actions to move the system to a better configuration. The prerequisite for carrying out dynamic evolutions is to count on models that are *causally connected* to the system (i.e., if the system changes, they change and vice versa). Since we are interested in managing the uncertainty that arises from the context in which the software is deployed, our approach is related to *external uncertainty* [15].

To make this strategy a reality, we propose a framework that states the models, tools, and artifacts that can support the dynamic evolution of context-aware systems to face uncertainty (see Figure 2). Our framework carries out unanticipated software changes at runtime continuously. It focuses on changing the system architecture by evolving a causally connected architecture model and it supports the autonomic evolution of the system (no human workload or system restarts are required to evolve the system) [7].

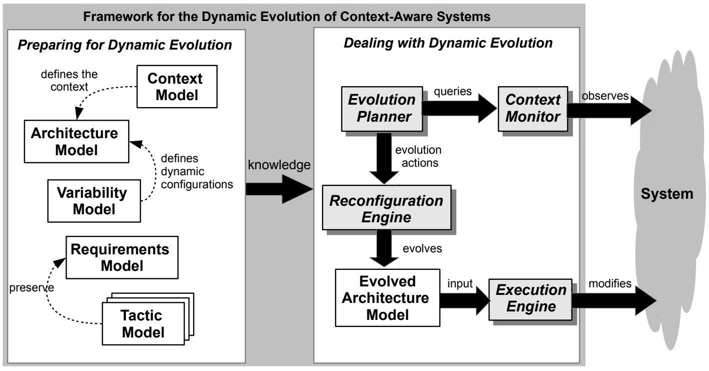


Fig. 2. Framework for the dynamic evolution of context-aware systems

The proposed framework has two phases, namely *Preparing for Dynamic Evolution* and *Dealing with Dynamic Evolution*. The models that are set up in the Preparing for Dynamic Evolution phase to support the dynamic evolution of context-aware systems are: 1) an *architecture model*, which describes the architecture of the system; 2) a *context model*, which formalizes the context knowledge; 3) a set of *tactic models*, which describes surviving tactics to preserve the requirements at runtime; 4) a *variability model*, which describes the dynamic configurations of the system; and 5) a *requirements model*, which describes requirements in an abstract way.

In the Dealing with Dynamic Evolution phase, the knowledge in the models that are created in the previous phase is used to guide the self-evolution of the system. The *context monitor* keeps an eye on context information at runtime. The *evolution planner* queries the information that is collected by the *context monitor* to determine if a requirement in the requirements model may be negatively impacted by an unknown context event. In order to preserve a requirement that has been affected, the *evolution planner* chooses a surviving tactic model. Based on the tactic model chosen and the variability model, the *evolution planner* generates abstract evolution actions. Then, the *reconfiguration engine* uses these actions to evolve the architecture model accordingly. Finally, the *execution engine* uses the evolved architecture model to modify the system.

3.1 Preparing for Dynamic Evolution

The objective of this phase is to set up the models that can be used to deal with unknown context events at runtime. These models can be divided into two groups. The first group consists of an architecture model and a context model. These two models are commonly used to drive the dynamic adaptation of the system [16,24]. This group can also include a requirements model that expresses the requirements to be fulfilled by the context-aware system at runtime. The second group consists of two novel models, a variability model and a set of tactic models, which extend the initial set of models to support the dynamic evolution of the system. The following subsections describe in a general way the aforementioned models. Specific examples illustrate the applicability of our approach. Figure 3 summarizes the models in our case study[2].

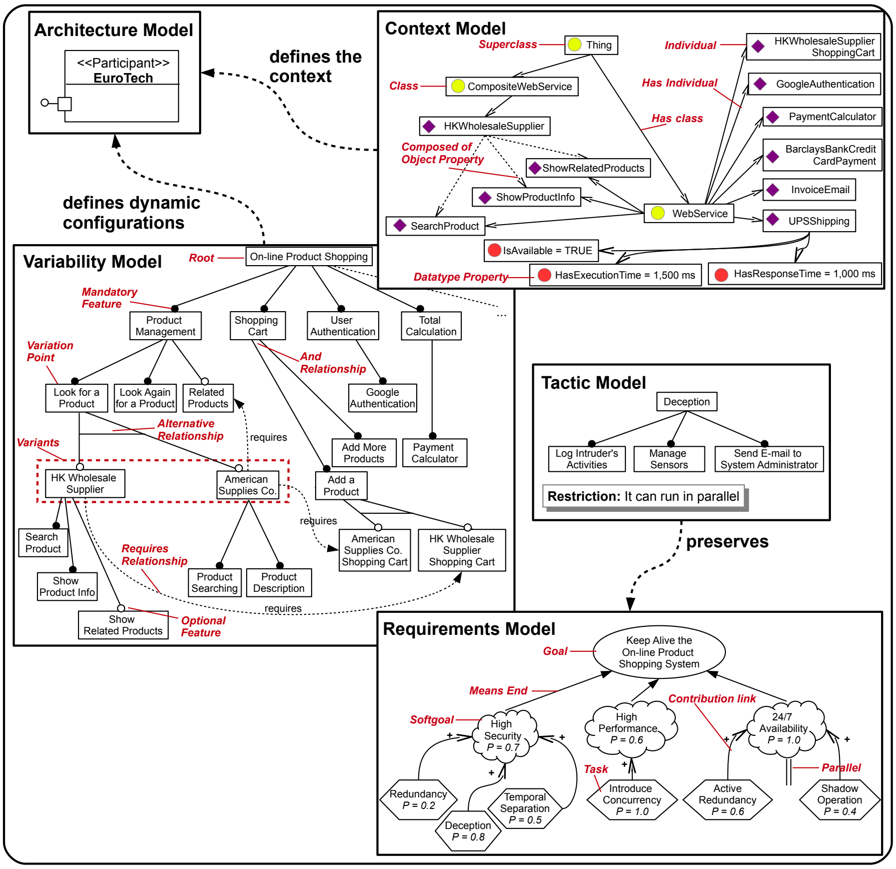


Fig. 3. The models in the Preparing for Dynamic Evolution phase of our case study

[2] The models that are used in the case study can be downloaded from our website [1].

An Architecture Model. This model describes the software architecture, which depends on the solution domain. For example, the architecture of a service composition can be represented by a SoaML architecture model (see Figure 1), or the architecture of a component-based system can be described by an architecture description language (ADL). Our framework is flexible with the notation used to describe this model.

A Context Model. This model supports the formal analysis of the collected context information. Specifically, it keeps the updated context knowledge at runtime to reason about when to trigger an autonomic system change. In order to examine the compliance of predefined situations in the context, systems analysts extract *context conditions* from the context model as Boolean expressions.

For example, in order to solve the need to express the context in a way that supports formal reasoning of its current status, the on-line product shopping system at EUROTECH uses an ontology-based context model that leverages Semantic Web technology. Specifically, it uses the Web Ontology Language[3] (OWL) to support the formal analysis of the context information that is captured at runtime (see Figure 3). *Individuals* have *datatype properties* that are used to represent the current context state. For example, the ISAVAILABLE datatype property indicates whether or not a service operation is currently available. Context conditions in our case study consider QoS parameters in the form of hard bounds. However, probabilistic contracts, which require statistical testing to check deviations of the QoS parameter, can also be used. Each context condition is represented as a triple in the form of *(subject, predicate, object)*. For example, the following predefined context condition is triggered when the current response time of the UPSSHIPPING Web service operation is greater than 2 seconds: *UPSShippingHiRespTime = (UPSShipping, HasResponseTime, >2,000 ms)*.

A Requirements Model. This model is leveraged at runtime to count on the representation of the requirements that the context-aware system must preserve at runtime. Requirements in this model have to be fulfilled despite arising unknown context events. There are several notations for requirements modeling, including UML Use Case diagrams and Goal Modeling (i*, GRL, and KAOS [19]).

Since our case study is particularly interested in keeping non-functional requirements (NFRs) at runtime (e.g. security, performance, and availability), the GRL [22] has been used for requirements modeling because it focuses on NFRs. Figure 3 depicts the *goal model* for the on-line product shopping system at EUROTECH. This model has *softgoals* that describe the NFRs to be kept by the context-aware system in order to reach the top-level goal (e.g. the HIGH SECURITY softgoal). It also contains *tasks* that specify specific surviving tactics to reach the softgoals (e.g. the DECEPTION task[4]). Since tasks represent core assets to keep the QoS of the system, they make a positive contribution to softgoals. Based on the temporal relationships of the tasks, they can be annotated as sequential, parallel or exclusive [28]. In addition, each softgoal has a priority defined at design time. If more than one softgoal has been affected at runtime, a task connected to

[3] `http://www.w3.org/TR/owl-ref/`: OWL Web Ontology Language.

[4] *Deception* (or *honeypot*) ensures survivability by inducing enemy behaviors that may be exploited.

the softgoal with the highest priority is executed first. Also, each task has a predefined priority. Tasks with the highest priority are chosen first at runtime.

A Variability Model. This model describes the dynamic system configurations and the variants of the system. These *variants* may provide better QoS, offer new functionalities that did not make sense in the previous context, or discard some functionalities. Our approach requires a variability modeling technique to implement the variability model, such as feature modeling [5], the Common Variability Language (CVL) [18], or any domain-specific language to express variability.

In the on-line product shopping system at EUROTECH, the variability model has been implemented with a *feature model*, which is a hierarchically arranged set of features. A *feature* is distinguished characteristics of a system (see Figure 3). Feature modeling was chosen to implement the variability model because it offers coarse-grained variability management and has good tool support for variability reasoning. Features are hierarchically linked in a tree-like structure through variability relationships such as optional, mandatory, and alternative [5].

In general terms, a system can be viewed as a set of functionalities that must be preserved at runtime. These functionalities are made up of components or services. These can be added or removed from a system at runtime when new context events arise. To this end, systems can be abstracted as a set of features that represent variant functionalities of the system in a variability model. Therefore, evolution actions can be described in terms of the activation or deactivation of features in the variability model. The set of all currently active features is the *current configuration* of the system.

In feature models, *variation points* are used to express decisions leading to different *variants* at runtime. These variation points can be bound during operation to adapt to contextual changes. Variation points are represented as interior nodes (e.g. the LOOK FOR A PRODUCT feature). Since only one variant can be chosen at a time in a particular variation point, there is an *alternative relationship* between a variation point and its variants. Each variant is denoted with an *optional* feature because it can be added or removed according to specific needs (e.g. the HK WHOLESALE SUPPLIER and the AMERICAN SUPPLIES CO. features).

A Set of Tactic Models. In our approach, *tactics* are considered as the last resort to be used when the system does not have predefined adaptation actions to deal with arising problematic context events. Writing complex scripts to specify tactics can be cumbersome. However, highly-abstract tactic models can be used to express the tactical functionality to be triggered on the underlying system to preserve requirements that are affected (one requirement can be preserved by many tactics). Therefore, tactic models are causally connected to software that implements the tactics. Also, tactic models are merged into the variability model at runtime to include the tactical functionality in the evolved system configuration. The only merging prerequisite is that these two models conform to the same metamodel. Each tactic model can have *restrictions* to indicate whether or not the tactic can run in parallel with other current active system functionalities. If not, it is necessary to activate or deactivate current functionalities to apply the tactic. Tactic models are expressed as broadly as possible in order to be reused.

Since the variability model in our case study is implemented as a *feature model*, the tactic models are also expressed as *feature models* to support their merging at runtime (to conform to the same metamodel). Figure 3 shows the tactic model that describes the functionality for the deception tactic. The root feature indicates the tactic's functionality, and leaf features express coarse-grained functionalities to fulfill the tactic. A *restriction* states that this tactic can run in parallel with the current active features.

3.2 Dealing with Dynamic Evolution

In this phase, the models that have been created in the Preparing for Dynamic Evolution phase are used to manage external uncertainty when facing unknown context events at runtime. To this end, this phase adds a *dynamic evolution* layer upon a *dynamic adaptation* layer (see Figure 4). This layer extends our previous work with dynamic evolution capabilities [8,2].

The dynamic adaptation layer triggers the self-adaptation of the system according to predefined adaptation actions. This layer is supported by the following three tools. The first tool is a *context monitor*. It processes context information that is collected by sensors and updates the context model accordingly (see step 1 in Figure 4). The *context monitor* captures the basic metrics of significant quality attributes from the context. The monitoring frequency is defined at design time. The second tool is a *reconfiguration engine* that self-adapts the supporting architecture model. The third tool is an *execution engine* that uses the modified architecture model to modify the underlying system. Different *execution engines* can be used depending on the solution domain. For example, the *execution engine* for a component-based system can be implemented on the OSGi framework[5]. The *execution engine* can install or uninstall components based on changes in the architecture model [8].

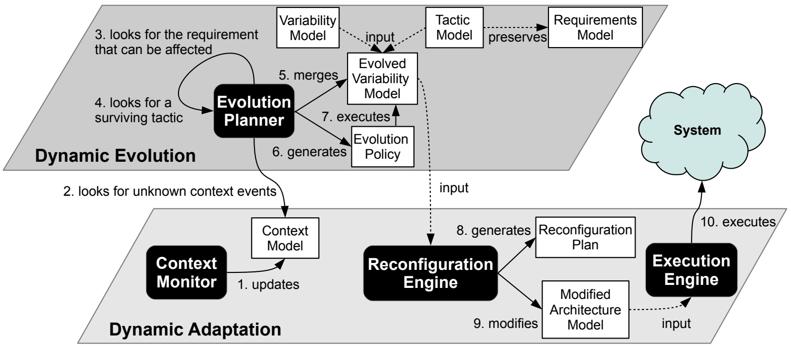


Fig. 4. Dealing with the Dynamic Evolution phase

The *dynamic evolution* layer supports the self-evolution of the system to deal with unknown context events in the open world. The *evolution planner* is constantly looking

[5] http://www.osgi.org: OSGi Alliance.

for unknown context events based on information collected in the context model (see step 2). If the *evolution planner* realizes that there is an unknown context event, then it looks for the requirement that can be affected by this event (see step 3). Afterwards, the *evolution planner* looks for a surviving tactic to preserve the affected requirement (see step 4). According to the discovered tactic, the *evolution planner* merges a tactic model into the variability model to produce an evolved variability model that supports the tactic's functionality (see step 5). Then, it generates and executes an *evolution policy* that indicates the activation or deactivation of features (i.e., evolution actions) in the evolved variability model according to the triggered tactic (see steps 6 and 7). Afterwards, the *reconfiguration engine* creates a *reconfiguration plan* that is based on the generated *evolution policy* (see step 8). The execution of the *reconfiguration plan* modifies the architecture model (see step 9). Then, the *execution engine* uses the modified architecture model to adjust the underlying system (see step 10). The following subsections describe the actions carried out by the *evolution planner* and the *reconfiguration engine*. These are the two main tools that support dynamic evolutions.

Evolution Planner

The *evolution planner* is in charge of generating the evolution actions to face unknown problematic context events. The main objective of the *evolution planner* is to protect the affected requirements with surviving tactics. This tool's output is a new system configuration in terms of an evolved variability model that indicates the functionalities to be activated or deactivated according to the selected tactic.

The *evolution planner* is constantly observing the context model to find situations that may require the dynamic evolution of the system. The most basic situation to trigger dynamic evolutions is when the *reconfiguration engine* does not find any predefined context condition for a context event. In this case, the context event is considered as *unknown*. However, the *evolution planner* can also use other mechanisms such as Gaia [26] to reason about uncertain contexts using mechanisms such as probabilistic logic, fuzzy logic, and Bayesian networks according to different situations. In order to deal with unknown context events, the *evolution planner* carries out the following steps:

- Look for the Requirement That Can Be Affected. The objective of this step is to look for the requirement that can be affected by an unknown context event. This is key information because the surviving tactics are associated to requirements.

In order to find the requirement that can be affected in our case study, the *evolution planner* uses *forward chaining* [21], a well-known method of reasoning in artificial intelligence. This method evaluates arising context *facts* against *rule premises*, which are defined at design time and kept in a knowledge base. A key advantage of forward chaining in the open world is that new context data can trigger new inferences. Figure 5 shows a basic example when the unknown context event *F1* (a fact) is detected. In this case, rule *R1* has a condition that matches this new fact (see step 1). Then, the forward chaining method fires the new fact *F2* (see step 2). The process continues until the fact *F3* is fired (see step 4). *F3* indicates that *the* ***HK Wholesale Supplier Web service operation*** *can affect the* ***High Security softgoal***.

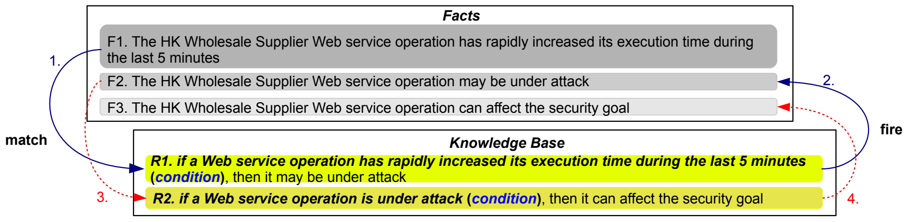

Facts
F1. The HK Wholesale Supplier Web service operation has rapidly increased its execution time during the last 5 minutes
F2. The HK Wholesale Supplier Web service operation may be under attack
F3. The HK Wholesale Supplier Web service operation can affect the security goal
Knowledge Base
R1. if a Web service operation has rapidly increased its execution time during the last 5 minutes (condition), then it may be under attack
R2. if a Web service operation is under attack (condition), then it can affect the security goal
match
fire
1.
2.
3.
4.

Fig. 5. Forward chaining inference example

- Look for a Surviving Tactic. The objective of this step is to discover what to do to preserve a requirement that has been negatively impacted by an unknown context event. To this end, the *evolution planner* looks for a surviving tactic among the set of tactic models.

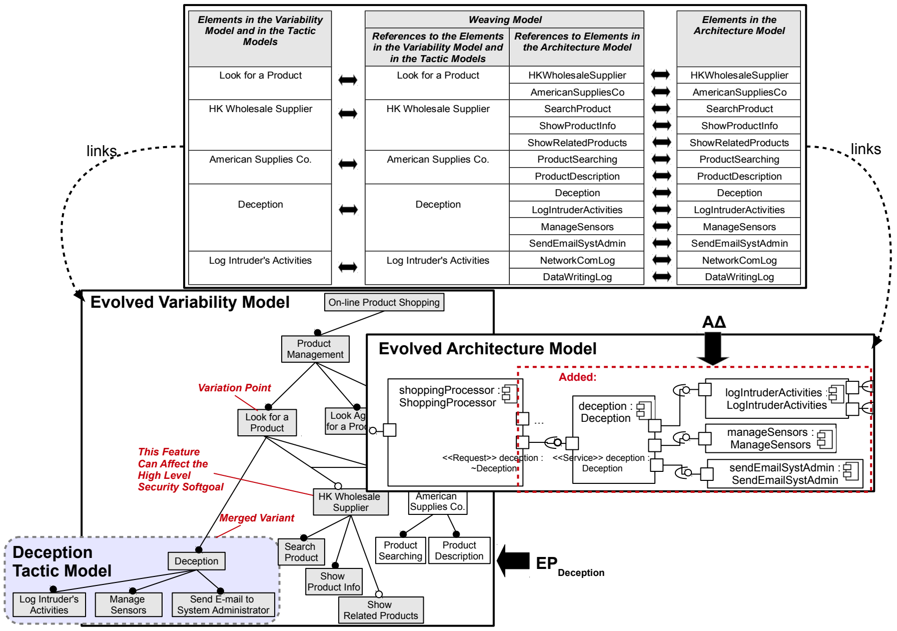
In our case study, this discovery process is supported by querying information from a GRL *goal model*, which implements the requirements model. For example, when the *evolution planner* finds that *the HK Wholesale Supplier Web service operation can affect the High Security softgoal*, it looks for the task with the highest priority under the affected HIGH SECURITY softgoal. In this case, it finds the DECEPTION task.

- Merge a Tactic Model into the Variability Model. In order to inject the functionality of the discovered tactic into the system, this step has two objectives: 1) to identify a tactic model that describes the tactic to be triggered for preserving an affected requirement at runtime; and 2) to merge the required tactic model into the variability model to count on an enriched-evolved variability model that guides dynamic changes in the system architecture.

In our case study, the *evolution planner* carries out two steps to merge tactic models into the feature-based variability model. First, the tactic model is considered as a *variant* that can be added to the variability model. Second, the tactic model is inserted under the *variation point* of the feature that has affected the requirement. For example, in the case of the discovered "*HK Wholesale Supplier Web service operation can affect the High Security softgoal*" context event, the *deception tactic model* is inserted as a variant under the LOOK FOR A PRODUCT variation point (see Figure 6).

- Generate an Evolution Policy. Merging a tactic model into the variability model may cause the activation or deactivation of features in the current configuration of the variability model. Therefore, the objective of this step is to generate an *evolution policy* that contains evolution actions to decide which system features need to be activated or deactivated in the evolved variability model. An *evolution policy* (*EP*) for a particular *tactic* (*T*) can be expressed as a list of pairs (*F*, *S*) where each pair is made up of a *feature* (*F*) in the *evolved variability model* (*EVM*) and the *state* (*S*) of the *feature* (*active* or *inactive*): $EP_T = \{(F,S)\,|\,F\,\varepsilon\,[EVM] \wedge S\,\varepsilon\,\{Active, Inactive\}\}$.

In our case study, the generated *evolution policies* activate or deactivate features according to the following rules: 1) since the features in the inserted tactic model are necessary to keep a particular requirement working, all the features in the tactic model are



Elements in the Variability Model and in the Tactic Models	Weaving Model: References to the Elements in the Variability Model and in the Tactic Models	Weaving Model: References to Elements in the Architecture Model	Elements in the Architecture Model
Look for a Product	Look for a Product	HKWholesaleSupplier	HKWholesaleSupplier
		AmericanSuppliesCo	AmericanSuppliesCo
HK Wholesale Supplier	HK Wholesale Supplier	SearchProduct	SearchProduct
		ShowProductInfo	ShowProductInfo
		ShowRelatedProducts	ShowRelatedProducts
American Supplies Co.	American Supplies Co.	ProductSearching	ProductSearching
		ProductDescription	ProductDescription
Deception	Deception	Deception	Deception
		LogIntruderActivities	LogIntruderActivities
		ManageSensors	ManageSensors
		SendEmailSystAdmin	SendEmailSystAdmin
Log Intruder's Activities	Log Intruder's Activities	NetworkComLog	NetworkComLog
		DataWritingLog	DataWritingLog

links
links
Evolved Variability Model
On-line Product Shopping
Product Management
Variation Point
Look for a Product
Look Ag for a Pro
This Feature Can Affect the High Level Security Softgoal
HK Wholesale Supplier
American Supplies Co.
Merged Variant
Search Product
Show Product Info
Show Related Products
Product Searching
Product Description
Deception Tactic Model
Deception
Log Intruder's Activities
Manage Sensors
Send E-mail to System Administrator
EP Deception
AΔ
Evolved Architecture Model
Added:
shoppingProcessor : ShoppingProcessor
...
<<Request>> deception : ~Deception
<<Service>> deception : Deception
deception : Deception
logIntruderActivities : LogIntruderActivities
manageSensors : ManageSensors
sendEmailSystAdmin : SendEmailSystAdmin

Fig. 6. Evolved models in our case study

activated; 2) if the variant that represents the inserted tactic can run in parallel with other variants, then the features in the evolved variability model keep their current states; and 3) if the tactic model requires the activation or deactivation of features in the initial variability model, then the *evolution planner* triggers the necessary changes. For example, the following *evolution policy* activates the deception functionality in the evolved variability model according to the discovered "*HK Wholesale Supplier Web service operation can affect the High Security softgoal*" context event (the *active* features in the current configuration are highlighted in Figure 6): $EP_{Deception}$ = *{(Deception, Active), (Log Intruder's Activities, Active), (Manage Sensors, Active), (Send E-mail to System Administrator, Active)}*.

Reconfiguration Engine

The *reconfiguration engine* creates a *reconfiguration plan*, which contains a set of reconfiguration actions to adapt the architecture model representing the underlying system. The adapted architecture model keeps the consistency between the evolved variability model and the underlying system. Reconfiguration actions in the *reconfiguration plan* are stated as *architecture increments* ($A\triangle$) and *architecture decrements* ($A\nabla$). These operations take an *evolution policy* as input, and they calculate the modifications to the architecture model by adding ($A\triangle$) or removing ($A\nabla$) model elements.

In our case study, the *evolution policy* is described in terms of the activation or deactivation of features, and the *reconfiguration plan* is described in terms of elements in the architecture model. Therefore, it was necessary to query the mappings in a *weaving model* [12] to determine which elements in the architecture model should be added or removed according to the features in the *evolution policy* (see Figure 6). Each *link* in the weaving model has two *endpoints*. The first endpoint refers to features in the initial variability model and in the set of tactic models (since these features can be present in the evolved variability model). The second endpoint refers to elements in the architecture model. In the case of the discovered "*HK Wholesale Supplier Web service operation can affect the High Security softgoal*" context event, the generated *reconfiguration plan* to reorganize elements in the architecture model is as follows: $A\triangle$ = *{Deception, LogIntruderActivities, ManageSensors, SendEmailSystAdmin, NetworkComLog, DataWritingLog}* (see the resulting model in Figure 6).

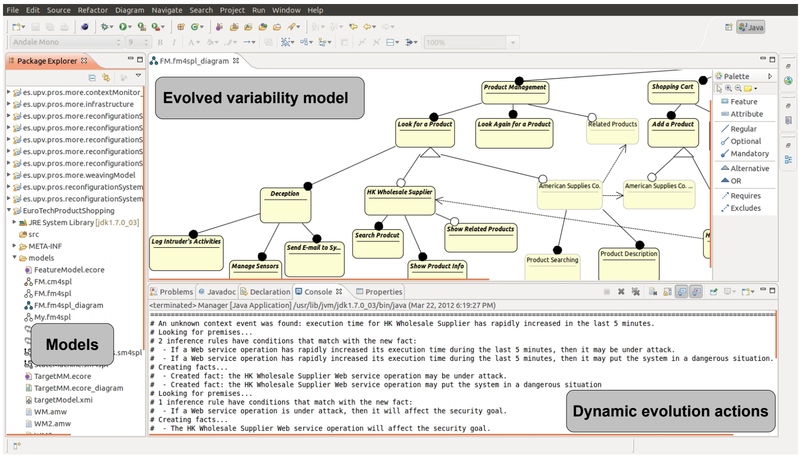
4 Realizing Our Approach: Methodology and Prototype

In order to carry out our approach in a coherent way from requirements to runtime, we propose a supporting methodology and a prototype system. First, the methodology in [1] specifies the tasks, work products, roles and tools needed to create the models in the Preparing for Dynamic Evolution phase. The Eclipse Process Framework (EPF) Composer[6] was used to create a plug-in that contains the method content [1]. The main benefit of this plug-in is that other methodologies can reference it and reuse its content. In our case study, the architecture model, the variability model, the requirements model, and the set of tactic models were specified in the XML Metadata Interchange (XMI) format. They were processed by the software infrastructure provided in the Eclipse Modeling Framework (EMF)[7] to specify and execute queries against them at runtime. The variability model and the set of tactic models conform to the MOSKitt4SPL[8] metamodel; the SoaML architecture model conforms to the SoaML metamodel, which extends the UML2 metamodel to support service modeling [25]; and the *goal model* conforms to the OpenOME metamodel[9]. The variability model and the architecture model were taken as input to create the weaving model in the ATLAS Model Weaver[10] tool.

The above models were used to carry out dynamic evolutions with a prototype (see Figure 7). SALMon [3] was chosen as the *context monitor* because its components are mostly technology-independent and they act as services, making the SALMon architecture customizable for our purpose. Context conditions were specified as SPARQL Protocol and RDF Query Language[11] (SPARQL) queries to the ontology in the context model. By using SPARQL queries, we have implemented the operations to insert new triples in the RDF graph of the ontology and to evaluate the values in the context model in order to find out if a predefined context condition has been accomplished. When an unknown context event arises, the *evolution planner* uses the *generic rule*

[6] `http://www.eclipse.org/epf/`: EPF Project.
[7] `http://www.eclipse.org/modeling/emf/`: EMF.
[8] `http://www.pros.upv.es/m4spl`: MOSKitt4SPL.
[9] `https://se.cs.toronto.edu/trac/ome/`: OpenOME.
[10] `http://www.eclipse.org/gmt/amw/`: ATLAS Model Weaver.
[11] `http://www.w3.org/TR/rdf-sparql-query/`: SPARQL.

reasoner in Jena 2[12] to carry out forward chaining. In addition, the *evolution planner* uses the EMF Compare[13] APIs to merge the tactic model into the variability model. It also uses the EMF Model Query (EMFMQ)[14] to activate or deactivate features in the evolved variability model during execution. The *execution engine* was implemented with Swordfish[15]. Web services were created using the Java API for XML Web services (JAX-WS) and deployed as *bundles*[16] in Swordfish.



```
<terminated> Manager [Java Application] /usr/lib/jvm/jdk1.7.0_03/bin/java (Mar 22, 2012 6:19:27 PM)
# An unknown context event was found: execution time for HK Wholesale Supplier has rapidly increased in the last 5 minutes.
# Looking for premises...
# 2 inference rules have conditions that match with the new fact:
#  - If a Web service operation has rapidly increased its execution time during the last 5 minutes, then it may be under attack.
#  - If a Web service operation has rapidly increased its execution time during the last 5 minutes, then it may put the system in a dangerous situation.
# Creating facts...
#  - Created fact: the HK Wholesale Supplier Web service operation may be under attack.
#  - Created fact: the HK Wholesale Supplier Web service operation may put the system in a dangerous situation
# Looking for premises...
# 1 inference rule have conditions that match with the new fact:
#  - If a Web service operation is under attack, then it will affect the security goal.
# Creating facts...
#  - The HK Wholesale Supplier Web service operation will affect the security goal.
```

Fig. 7. A screenshot of our prototype to manage uncertainty in context-aware systems

5 Related Work

There are several related works in the area of self-adaptive context-aware systems [9]. Recent research has proposed using models at runtime as a feasible way to guide dynamic adaptations [23,24,8,14,16,2]. Nevertheless, these approaches lack support to manage uncertainty in the open world. We focus our discussion on relevant model-driven works that deal with uncertainty in context-aware systems.

In [26], the authors present a pervasive infrastructure to reason about uncertainty using learning that is based on Bayesian networks and rules that are written in probabilistic logic. Similar to our context model, they also use ontologies to reason about uncertain context information. However, they do not offer the mechanisms to adapt the underlying system. There is an interesting research trend towards the analysis of uncertainty

[12] http://incubator.apache.org/jena/: Jena.
[13] http://www.eclipse.org/emf/compare/: EMF Compare Project
[14] http://www.eclipse.org/modeling/emf/: EMF Model Query.
[15] http://www.eclipse.org/swordfish/: Swordfish.
[16] A *bundle* is a module containing Java implementation classes and additional data that can be deployed in an OSGi runtime environment.

using *goal models* at the requirements phase. For instance, in [27], the authors propose attaching claims to softgoal contribution links to record the rationale for a choice of goal realization strategy when there is uncertainty about the optimum choice. Nevertheless, it may be difficult to determine the whole set of claims in large and complex systems. In [10], the authors introduce a goal-based modeling approach to develop the requirements for a dynamic adaptive system while explicitly factoring uncertainty into the process and the requirements. The requirements model in our approach can be specified using their technique to identify high-level goals in order to mitigate uncertainty. Another trend focuses on the generation of models at design time that represent possible target systems that are suitable for different environmental conditions. For example, in [17], the authors propose a digital evolution-based approach to generate these models at design time. Our model-based surviving tactics extend their approach in cases when the model generation misses any unforeseen context event that may attempt against a requirement at runtime. Finally, in [11], the authors describe three sources of uncertainty and explain how they address those in the Rainbow Project. Our approach solves two of them: the identification of a system problem (known and unknown problematic context events) and the selection of an adaptation strategy (generation of an evolution plan). Even though Rainbow includes an architecture model to adapt the system at runtime, uncertainty management techniques are not model-driven.

The aforementioned approaches offer interesting solutions for analyzing the collected context information in uncertain contexts and for modeling uncertainty at the requirements and design phases for later use at runtime. However, the models created at design time do not evolve at runtime to deal with arising problematic unknown context events. According to our best knowledge, our work presents the first generic model-driven framework to handle unknown context events through the dynamic evolution of the system.

6 Conclusions and Future Work

In this paper, we have presented a model-based framework to support the dynamic evolution of context-aware systems that deals with unexpected context events in the open world. A case study illustrates the applicability of our framework and a realization methodology and a proof-of-concept prototype validate the feasibility of the proposed approach. Our framework has several benefits: 1) it can guide the creation of context-aware systems that self-evolve when facing unknown context events in order to preserve the expected requirements; 2) semantically-rich and easy-to-understand models facilitate the development and maintenance of software that needs to self-evolve; 3) the framework can be adjusted to different domains; and 4) human workload is reduced thanks to the autonomic evolution of the system. There are several directions for future work. One possibility is to extend our framework using the approach in [10] to manage uncertainty from the requirements phase. A second possibility is to validate the evolved architecture model at runtime to prevent negative effects in the system. In this manner, if the architecture model reaches an inconsistent state, then the *evolution planner* generates a different strategy. A third possibility is to define mechanisms that can evaluate at design time the completeness of the models to handle unknown context events.

References

1. Alférez, G.H., Pelechano, V.: Dynamic evolution of context-aware systems with models at runtime, http://fit.um.edu.mx/harvey/dynamicevolution/
2. Alférez, G.H., Pelechano, V.: Context-aware autonomous web services in software product lines. In: SPLC, pp. 100–109 (2011)
3. Ameller, D., Franch, X.: Service level agreement monitor (SALMon). In: Proceedings of the Seventh International Conference on Composition-Based Software Systems (ICCBSS 2008), pp. 224–227. IEEE Computer Society, Washington, DC (2008)
4. Baresi, L., Di Nitto, E., Ghezzi, C.: Toward open-world software: Issue and challenges. Computer 39, 36–43 (2006)
5. Batory, D.: Feature Models, Grammars, and Propositional Formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
6. Blair, G., Bencomo, N., France, R.B.: Models@ run.time. Computer 42, 22–27 (2009)
7. Buckley, J., Mens, T., Zenger, M., Rashid, A., Kniesel, G.: Towards a taxonomy of software change. Journal of Software Maintenance 17(5), 309–332 (2005)
8. Cetina, C., Giner, P., Fons, J., Pelechano, V.: Autonomic computing through reuse of variability models at runtime: The case of smart homes. Computer 42, 37–43 (2009)
9. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009)
10. Cheng, B.H.C., Sawyer, P., Bencomo, N., Whittle, J.: A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 468–483. Springer, Heidelberg (2009)
11. Cheng, S.W., Garlan, D.: Handling uncertainty in autonomic systems. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (November 2007)
12. Del Fabro, M.D., Bézivin, J., Valduriez, P.: Weaving models with the Eclipse AMW plugin. In: Eclipse Modeling Symposium, Eclipse Summit Europe (2006)
13. Dey, A.K.: Understanding and using context. Personal Ubiquitous Comput. 5, 4–7 (2001)
14. Elkhodary, A., Esfahani, N., Malek, S.: Fusion: a framework for engineering self-tuning self-adaptive software systems. In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2010, pp. 7–16. ACM, New York (2010)
15. Esfahani, N., Kouroshfar, E., Malek, S.: Taming uncertainty in self-adaptive software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE 2011, pp. 234–244. ACM, New York (2011)
16. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer 37(10), 46–54 (2004)
17. Goldsby, H.J., Cheng, B.H.C.: Automatically Generating Behavioral Models of Adaptive Systems to Address Uncertainty. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 568–583. Springer, Heidelberg (2008)
18. Haugen, O., Møller-Pedersen, B., Oldevik, J., Olsen, G.K., Svendsen, A.: Adding standardized variability to domain specific languages. In: Proceedings of the 2008 12th International Software Product Line Conference, SPLC 2008, pp. 139–148. IEEE Computer Society, Washington, DC (2008)

19. Horkoff, J., Yu, E.: Analyzing goal models: different approaches and how to choose among them. In: Proceedings of the 2011 ACM Symposium on Applied Computing, SAC 2011, pp. 675–682. ACM, New York (2011)
20. Keeney, J.: Completely Unanticipated Dynamic Adaptation of Software. Ph.D. thesis, Trinity College Dublin (2004), http://www.tara.tcd.ie/bitstream/2262/30726/1/TCD-CS-2005-43.pdf
21. Labhart, J., Rowe, M., Matney, S., Carrow, S.: Forward chaining parallel inference. In: Proceedings of the IEEE 1990 National on Aerospace and Electronics Conference, NAECON 1990, vol. 3, pp. 1124–1131 (May 1990)
22. Liu, L., Yu, E.: Designing information systems in social context: a goal and scenario modelling approach. Inf. Syst. 29, 187–203 (2004)
23. Menasce, D., Gomaa, H., Malek, S., Sousa, J.: SASSY: A framework for self-architecting service-oriented systems. IEEE Software 28, 78–85 (2011)
24. Morin, B., Fleurey, F., Bencomo, N., Jézéquel, J.M., Solberg, A., Dehlen, V., Blair, G.: An Aspect-Oriented and Model-Driven Approach for Managing Dynamic Variability. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 782–796. Springer, Heidelberg (2008)
25. OMG: Service oriented architecture modeling language (SoaML) specification (March 2012), http://www.omg.org/spec/SoaML/1.0
26. Ranganathan, A., Al-Muhtadi, J., Campbell, R.: Reasoning about uncertain contexts in pervasive computing environments. IEEE Pervasive Computing 3(2), 62–70 (2004)
27. Welsh, K., Sawyer, P., Bencomo, N.: Towards requirements aware systems: Run-time resolution of design-time assumptions. In: 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 560–563 (November 2011)
28. Yu, Y., Lapouchnian, A., Liaskos, S., Mylopoulos, J., Leite, J.C.S.P.: From Goals to High-Variability Software Design. In: An, A., Matwin, S., Raś, Z.W., Ślęzak, D. (eds.) Foundations of Intelligent Systems. LNCS (LNAI), vol. 4994, pp. 1–16. Springer, Heidelberg (2008)

An Eclipse Modelling Framework Alternative to Meet the Models@Runtime Requirements

François Fouquet[1], Grégory Nain[2], Brice Morin[3], Erwan Daubert[1], Olivier Barais[1], Noël Plouzeau[1], and Jean-Marc Jézéquel[1]

[1] University of Rennes 1, IRISA, INRIA Centre Rennes
Campus de Beaulieu, 35042 Rennes, France
{Firstname.Lastname}@inria.fr
[2] SnT - University of Luxembourg
Luxembourg, Luxembourg
gregory.nain@uni.lu
[3] SINTEF
Oslo, Norway
Brice.Morin@sintef.no

Abstract. Models@Runtime aims at taming the complexity of software dynamic adaptation by pushing further the idea of reflection and considering the reflection layer as a first-class modeling space. A natural approach to Models@Runtime is to use MDE techniques, in particular those based on the Eclipse Modeling Framework. EMF provides facilities for building DSLs and tools based on a structured data model, with tight integration with the Eclipse IDE. EMF has rapidly become the defacto standard in the MDE community and has also been adopted for building Models@Runtime platforms. For example, Frascati (implementing the Service Component Architecture standard) uses EMF for the design and runtime tooling of its architecture description language. However, EMF has primarily been thought to support design-time activities. This paper highlights specific Models@Runtime requirements, discusses the benefits and limitations of EMF in this context, and presents an alternative implementation to meet these requirements.

Keywords: Model@Runtime, EMF, adaptation.

1 Introduction

The emergence of new classes of systems that are complex, inevitably distributed, and that operate in heterogeneous and rapidly changing environments raise new challenges for the Software Engineering community [3]. Examples of such applications include those from crisis management, health-care and smart grids. These applications can be deployed on top of a distributed infrastructure that goes from micro-controller to the Cloud. These systems must be adaptable, flexible, reconfigurable and, increasingly, self-managing [9]. Such characteristics make systems more prone to failure when executing and thus the development and study of appropriate mechanisms for continuous design and runtime validation and monitoring are needed. In the Model-Driven Software Development area, research effort has focused primarily on using models at design, implementation, and deployment stages of development. This work has been highly productive with several techniques now entering the commercialization phase. The use of model-driven techniques for validating and monitoring run-time behavior can also yield significant benefits. A key benefit is that models can be used to provide a richer semantic base for runtime decision-making related to system adaptation and other runtime concerns such as verification and monitoring. Then, Models@Runtime [2] denotes model-driven approaches aiming at taming the complexity of software and system dynamic adaptation. It basically pushes the idea of reflection [11] one step further by considering the reflection layer as a real model: "*something simpler, safer or cheaper than reality to avoid the complexity, danger and irreversibility of reality*" [14], which enables the continuous design of complex, adaptive systems.

A natural approach to Models@Runtime is to use MDE techniques, in particular those based on the Eclipse Modeling Framework (EMF) [1]. For example, Frascati [16] (implementing the Service Component Architecture standard) uses EMF for the design and runtime tooling of its architecture description language. However, EMF has primarily been thought to support design-time activities and its use to support Models@Runtime reaches some limitations. This paper elicits specific Models@Runtime requirements, discusses the benefits and limitations of EMF in this context, and presents an alternative modelling framework implementation to meet these requirements.

The outline of this paper is the following. Section 2 briefly presents the Models@Runtime paradigm and its requirements. An overview of EMF benefits and its limitations regarding its use at runtime are given by Section 3. The contribution of this paper, the Kevoree Modeling Framework(KMF), is described in Section 4. Section 4.2 gives an evaluation of our alternative implementation in comparison to EMF. This contribution is discussed *w.r.t.* related work in Section 5. Section 6 concludes on about this work and presents future work.

2 Models@Runtime Requirements

The Models@Runtime paradigm promises a new approach to MDE, by fading the boundary between design-time (the typical phase where MDE is employed) and runtime. More precisely, the goal of Models@Runtime is to enable the continuous design, evolution, verification of eternal running software systems [2]. A typical usage of Models@Runtime is to manage the complexity of dynamic adaptation or verification in complex, **distributed** and **heterogeneous** systems, by offering a more abstract and safer abstraction layer on top of the running system than reflection. Heterogeneity and distribution creates specific requirements for Models@Runtime infrastructure. (i) The overhead inevitably induced by this advanced reflection layer should not prevent smaller (*i.e.* resource constrained)

[1] http://www.eclipse.org/emf/

devices to benefit from the advantages of Models@Runtime (e.g. Java Embedded, Android,...). Modeling framework and all its needed dependencies must be compatible with such devices in terms of memory footprint. (ii) The use of models to drive the running configuration of a software system should enable required features of a distributed reflection layer such as efficient (un)-marshalling, efficient model cloning and model thread safety access.

2.1 Reduced Memory Footprints

The memory footprint of a Models@Runtime engine basically determines the types of nodes able to run this engine. The more demanding is the Models@Runtime engine in terms of memory, the more difficult it is to deploy it on the smallest devices (e.g. Android phones, gateways with low power CPUs), and the more centralized should the adaptation/verification be. Lazy loading technique can be used to virtually reduce the memory overhead by not loading unused model elements. In this case, only a few large devices would be able to reason and make decisions for all the smaller devices. This would reduce the reliability of the overall adaptation and verification process: if the large devices fail, the overall system cannot safely adapt anymore. Moreover, model exchanges for the synchronization of the system in this strategy would dramatically increase network load.

2.2 Dependencies

The number and size of dependencies is also an important criteria. Each device must provision all the dependencies needed by the modeling framework to run a Models@Runtime based distributed application. As these applications are based on a structured data model, this data model should not generate useless dependencies. Heavy dependencies would indeed increase the time needed to initialize a node or update it when new versions of those third parties are available.

2.3 Thread Safety

A Models@Runtime is generally used in highly concurrent environment. For instance, different probes integrated in a device update a context model. This model is then used for triggering the adaptation reasoning process. This context model should enable safe and consistent read and write for the reasoners to take accurate decisions. The Models@Runtime infrastructure must ensure that the multiple threads of your application can access and modify the models without worrying about the concurrent access details. In particular, it should be possible to navigate in parallel the collections defined in the model to implement fast, yet safe, validation or reasoning algorithms on multi-core/thread nodes.

2.4 Efficient Model (Un)Marshalling and Cloning

A device should be able to locally clone its own model for verification or reasoning purposes so that it can reason on a fully independent and safe representation of itself, which can later on be re-synchronized with the current model. Also, devices should be offered efficient means to communicate their Models@Runtime to neighbors so that collective decisions can be made. The Models@Runtime infrastructure must thus provide efficient model cloning and (un)marshalling capabilities.

2.5 Connecting Model@Runtime to Classical Design Tools

This requirement is directly bound to the first goal of fading the boundary between design-time and runtime. A Models@Runtime infrastructure must provide a transparent compatibility with design environments. For example, a graphical simulator used for the design of finite state machines (FSM) should be pluggable on an application that keeps FSM at runtime and serve as a debugger or a monitor of the running system [1,7].

3 EMF Benefits and Limitations

A natural way to implement a Models@Runtime platform is to rely on tools and techniques well established in the MDE community, and in particular, the *de facto* EMF standard. This section provides a brief overview of EMF and then discusses the suitability of this modelling framework with respect to the requirements identified in the previous section.

3.1 EMF Overview

EMF is an eMOF implementation and code generation facility for building tools and other applications based on a structured data model. From a model specification, EMF provides tools and runtime support to create Domain Specific Language (DSL) on top of the Eclipse platform. Most important, EMF provides the foundation for interoperability with other EMF-based tools and applications using a default serialization strategy based on XMI. Consequently, EMF has been used to implement a large set of tools and thus, evolved into an efficient Java implementation of a core subset of the MOF API.

3.2 Advantages

As a first real benefit, EMF provides a transparent compatibility of the Models@Runtime infrastructure with several design environments. All the tools built with frameworks such as Xtext [10,4], EMFText [8], GMF [15] or ObeoDesigner [2] can be directly plugged on the Models@Runtime infrastructure to monitor the running system. The generated code is clean and provides an embedded visitor pattern and an observer pattern [6]. EMF also provides an XMI marshaller and unmarshaller that can be used to easily share models. Finally EMF offers lazy loadings of resources allowing the loading of single model elements on demand and caching them softly in an application.

[2] `http://www.obeodesigner.com/`

3.3 Limitations

To highlights the limitations of EMF, we will use the following experiment based on a simple Finite State Machine (FSM) metamodel with four meta-classes (FSM, State, Transition and Action in Fig. 1). A FSM contains the States of the machine and references to initial, current and final states. Each State contains its outgoing Transitions, and Transitions can contain Actions. From this tiny example, we discuss thread safety and dependencies, and we evaluate the memory footprint, as well as model (un)marshaling and cloning.

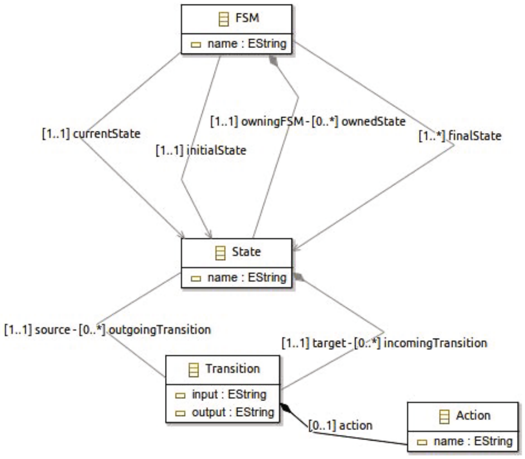


Fig. 1. Finite State Machine Metamodel used for Experiments

Large Dependency Set. Figure 2 shows the plugin/bundle dependencies for each new EMF generated code. By analyzing these dependencies one can see that the generated code is tightly coupled to the Eclipse environment and to the Equinox runtime (Equinox is the version of OSGi by the Eclipse foundation). Although this is not problematic when the data model is integrated as an Eclipse plugin (with all dependencies imposed by the Eclipse environment); these dependencies are more difficult to resolve and provision when this metamodel is used outside Eclipse, *i.e.* in a standalone context.

For the simple FSM metamodel, a standalone JAR executable outside of the Eclipse tool (a Java archive that including all dependencies) has a size of **15 MB** for only **55 KB** generated files. This footprint is rather difficult to reduce with tools like ProGuard[3], since it contains a large number of reflexive calls, which could potentially and implicitly affect any code in these 15 MB.

[3] ProGuard is a code shrinker (among other features not relevant for this paper) for Java bytecode: `http://proguard.sourceforge.net/`

The large number and size of dependencies is one of the main limitations of EMF when the model must be embedded at runtime.

Static Registries and Multi-class Loader Incompatibility. Many runtime for dynamic architecture (e.g. OSGi, Frascati, or Kevoree) need to use their own class loader to properly manage and improve dynamic class loading. Consequently, the second limitation comes from the use of static registries in EMF, that leads to incompatibilities with runtime using multi-class loaders.

Lack of Thread Safe Access to the Models. EMF does not provide thread safe accesses to the models [4]. This requirement is important for a Models@Runtime infrastructure, because the support for dynamic and distributed architectures requires concurrent access to models.

Cloning Overhead. Another limitation is the large memory footprint of marshaling, unmarshaling and cloning in the EMF implementations. To measure this limitation, we programmatically created a model with 100,000 State instances, with a transition between each state and an action for each Transaction. The results for EMF are the following.

On a Dell Precision E6400 with a 2.5 GHz iCore I7 and 16 GB of memory, the model creation lasts 376 ms, its marshaling to a file lasts 7021 ms and

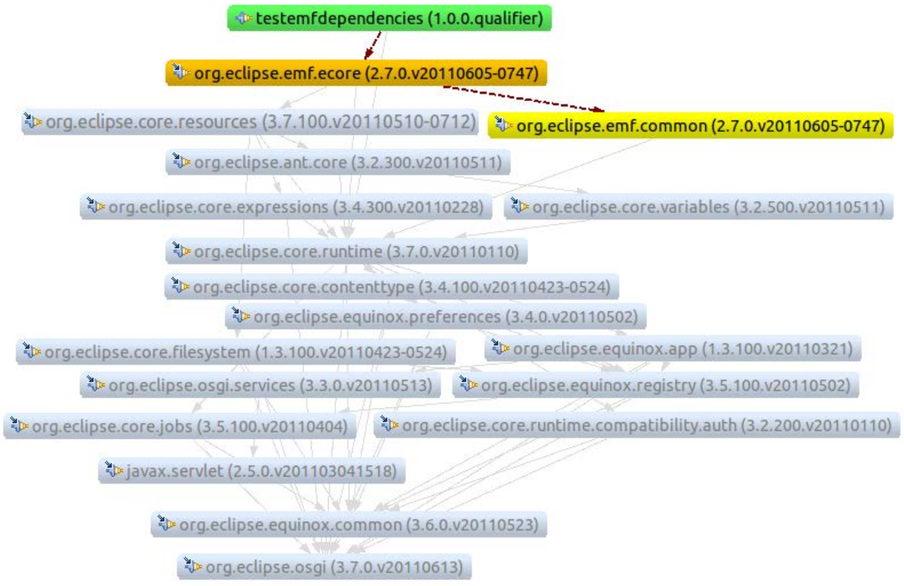

testemfdependencies (1.0.0.qualifier)
org.eclipse.emf.ecore (2.7.0.v20110605-0747)
org.eclipse.core.resources (3.7.100.v20110510-0712)
org.eclipse.emf.common (2.7.0.v20110605-0747)
org.eclipse.ant.core (3.2.300.v20110511)
org.eclipse.core.expressions (3.4.300.v20110228)
org.eclipse.core.variables (3.2.500.v20110511)
org.eclipse.core.runtime (3.7.0.v20110110)
org.eclipse.core.contenttype (3.4.100.v20110423-0524)
org.eclipse.equinox.preferences (3.4.0.v20110502)
org.eclipse.core.filesystem (1.3.100.v20110423-0524)
org.eclipse.equinox.app (1.3.100.v20110321)
org.eclipse.osgi.services (3.3.0.v20110513)
org.eclipse.equinox.registry (3.5.100.v20110502)
org.eclipse.core.jobs (3.5.100.v20110404)
org.eclipse.core.runtime.compatibility.auth (3.2.200.v20110110)
javax.servlet (2.5.0.v201103041518)
org.eclipse.equinox.common (3.6.0.v20110523)
org.eclipse.osgi (3.7.0.v20110613)

Fig. 2. Dependencies for each new metamodel generated code

[4] http://wiki.eclipse.org/EMF/FAQ#Is_EMF_thread-safe.3F

uses 104 MB of heap memory. The cloning using `EcoreUtil` lasts 3588 ms, and loading the model from a file lasts 5868 ms[5].

3.4 Synthesis

Table 1 summarizes the advantages and limitations for the usage of EMF as a foundation for a Models@Runtime infrastructure. For each criterion, we put a $\vee$ when EMF provides advantages, $\times$ when we see limitations of using EMF for building Models@Runtime infrastructure, $\sim$ when we see possible improvements.

Table 1. EMF features compared with Models@Runtime requirements

Memory footprints	$\times$
Lazy loading	$\vee$
Dependencies	$\times$
Thread safety	$\times$
Efficient model (un)marshalling and cloning	$\sim$
Connecting design tools	$\vee$

4 Kevoree Modeling Framework

KMF, or Kevoree Modeling Framework [6], is our alternative realization of EMF, which was formerly developed as part of our Kevoree Models@Runtime engine. This section presents the design choices we made to support a generic and efficient Models@Runtime infrastructure compatible with EMF. The general idea of KMF is threefold:

1. KMF aims at keeping the compatibility with EMF to guarantee the compatibility with design environment and the marshalling and unmarchalling of models.
2. KMF aims at leveraging the powerful features provided by modern programming languages (here, Scala) [13] to provide a proper design to handle models.
3. KMF aims at providing a generic Models@Runtime infrastructure to ease the heterogeneity and the distribution management.

4.1 Model Handling

Regarding the Table 1, KMF provides the same features than EMF for code generation facilities and models (un)marshalling. All the generated artefacts

[5] This experiment can be downloaded http://goo.gl/CyLLC

[6] https://github.com/dukeboard/kevoree-modeling-framework

are written in Scala. Scala is a general purpose programming language designed to express common programming patterns in a concise, elegant, and type-safe way. It smoothly integrates features of object-oriented (Traits) and functional languages, and provides bytecode compatibility with Java [13]. Scala uses type inference to combine static safety with the concise syntax of dynamically typed languages. The Scala features particularly relevant for KMF are the following: ByteCode compatibility with Java libraries, concept of traits, concept of *Option*, XML embedded, immutable List, concept of closure and efficiency.

Domain classes are generated as a set of Scala traits to ease the support of multiple inheritance and meta-model extension [5]. Traits are seen from Java Code as a Type. They can only be initialized through the generated Factory (as in EMF). Note that the generated Traits do not inherit from EMF `EObject` and that all references are initialized. In particular, collections are initialized and references to single objects with a lower bound of 0 rely on Options. These Scala options are a neater way to deal with null pointers[7]. Indeed, Option does not save the developer from ever having null, but that developer can only get null when he wants it. If it is semantically impossible for a value to be null, the type checker enforces it. The getters on collection use immutable lists. The generated code provides helpers to add and remove model elements on collections, and it also provides specific methods to ease mixed Java/Scala development. XML template are directly embedded in the Scala generated code and type checked by the Scala type checker. Fig. 3 shows an excerpt of the Scala traits generated for the domain model meta-classes. It shows the use of immutable list and Option for 0..n reference and 0..1 reference respectively.

4.2 Memory Footprint

To limit the dependencies, we decided to restrict the inheritance relationships in our generated code only to generated classes and to classes from the Java and the Scala frameworks. In this way dependencies are limited to the Java and Scala frameworks. A standalone JAR for the same metamodel in KMF has a size of 7 MB. After applying ProGuard, we obtain a JAR of 1.7MB. Indeed, the Scala dependencies load many packages that are not used by KMF, such as Scala-Swing, Scala-actors, etc.

Consequently, KMF has successfully been used on top of Dalvik [8], Avian [9], JamVM[10] or JavaSE for embedded Oracle Virtual Machine [11].

[7] http://www.scala-lang.org/api/current/scala/Option.html
[8] http://www.dalvikvm.com/
[9] http://oss.readytalk.com/avian/
[10] http://jamvm.sourceforge.net/
[11] http://www.oracle.com/technetwork/java/embedded/downloads/javase/index.html

```
trait Transition extends FsmSampleContainer {
        private var input : java.lang.String = ""

        private var output : java.lang.String = ""

        private var source : fsmSample.State = _

        private var target : fsmSample.State = _

        private var action : Option[fsmSample.Action] = None
        ...
}


trait FSM extends FsmSampleContainer {

        // 0..n reference
        private var ownedState : scala.collection.mutable.ListBuffer[
            fsmSample.State] = scala.collection.mutable.ListBuffer[
            fsmSample.State]()
        ....
        // 0..n reference method helpers
        def getOwnedState : List[fsmSample.State] = {
                ownedState.toList
        }
        def getOwnedStateForJ : java.util.List[fsmSample.State] = {
                import scala.collection.JavaConversions._
                ownedState
        }

        def setOwnedState(ownedState : List[fsmSample.State] ) {
                this.ownedState.clear()
                this.ownedState.insertAll(0,ownedState)
                ownedState.foreach{e=>e.setEContainer(this,Some(()=>{
                    this.removeOwnedState(e)}))}

        }

        def addOwnedState(ownedState : fsmSample.State) {
                ownedState.setEContainer(this,Some(()=>{this.
                    removeOwnedState(ownedState)}))
                this.ownedState.append(ownedState)
        }

        def addAllOwnedState(ownedState : List[fsmSample.State]) {
                ownedState.foreach{ elem => addOwnedState(elem)}
        }

        def removeOwnedState(ownedState : fsmSample.State) {
                if(this.ownedState.size != 0 ) {
                        this.ownedState.remove(this.ownedState.indexOf(
                            ownedState))
                        ownedState.setEContainer(null,None)
                }
        }

        def removeAllOwnedState() {
                this.ownedState.foreach{ elem => removeOwnedState(elem)}
        }

        def getClonelazy(subResult : java.util.IdentityHashMap[Object,
            Object]): Unit = {
            ...
        }
}
```

Fig. 3. Excerpt of generated Scala code for domain meta-classes

4.3 Multi-thread Access

Model@Runtime serves as a common software reflection layer concurrently exploited by many processes; protection against such accesses may be coarse or fine grain.

At fine grain the model essentially needs to be read concurrently while allowing modifications. The KMF generated code realizes such protection by internally using mutable collections (for performance reasons) but only exposing cloned immutable list to outside via its public API. As a result processes can navigate the cloned list while others perform CRUD operations on it. Each process needs to actively ask a new cloned version to access to the modifications. Moreover, the mutator methods (setter) can be protected behind synchronized blocks.

At coarse grain the model representation is entirely hidden behind a safe model care tracker as in the Memento pattern [6]. This safe model care tracker systemically clones the model on *get* operations and keeps a master representation. This structure is particularly useful to keep an history of model representation at runtime. KMF can optionally generate such structure using Scala actors to protect concurrent access (get / put) to model care tracker.

4.4 Loader, Serializer and Cloner

When working with models, two tasks are essential and used before and after each action on a model, namely marshalling and unmarchalling.

Where EMF offers a generic loader we propose to use the generation phase to also generate a specific loader for each meta-model.

The EMF generic loader takes a model to load and its meta-model as parameters and intensively uses reflection mechanisms to perform the loading task. If this kind of mechanism allows creating a single loader, its usage is not efficient. The generated KMF code then provides meta-model specific loader, saver and cloner to improve their efficiency. We use the XML API which is part of the Scala standard library to parse and print XMI representations of object models, with no need for extra dependencies, and because it is efficient.

The loading and cloning are performed in two phases. The first phase consists in traversing the models for creating the objects in the order they are found in the XMI file. The last step links the objects together according to the references previously cached. Currently, the Maven plugin we propose is only able to generate loaders and serializers for the XMI file format. EMF compatibility is obtained through the XMI file format. A direct API compatibility can be performed when EMF will separate the Ecore interfaces and Ecore implementation in different bundles to avoid useless dependencies.

4.5 Experiment and Synthesis

Besides, the memory footprints used to store, load, save or clone a model has decreased compared to the reference EMF implementation. To measure the memory footprints, (un)marshalling and cloning, we do the same experiment. We

Table 2. EMF and KMF efficiency

	EMF	*KMF*	comparison
Model creation	376 ms	313 ms	1.2 times faster
Model clone	3588 ms	398 ms	9 times faster
Model save	7021 ms	2630 ms	2.66 times faster
Memory footprint	104MB of heap memory	61MB of heap memory	1.70 times lighter

Table 3. EMF and KMF regarding models@runtime requirements

	EMF	*KMF*
Memory footprints	×(104MB)	∨(61MB)
Dependencies	× (15MB)	∨ (Scala standard library) (1.7MB)
Lazy Loading	∨	∨ (Proxy support)
Thread safety	×	∨ (immutable lists, no registry)
Efficient model (un)marshalling and cloning	∼	∨ (see Table 2)
Design tools compatibility	∨	∨ (through XMI compatibility)

programmatically create models with 100 000 States with a transition between each state and an action in each Transaction.

The results for KMF are the following. On a Dell Precision E6400 with an Intel 2.5GHz iCore I7 CPU and 16GB of RAM, it takes 313ms to create the models, 2630 ms to save it in a file, 61 Mbytes of Heap memory, 398ms to clone and 3000s to load the model from a file[12]. Table 2 highlights the quantitative performance comparison results between EMF and KMF.

Table 3 provides the qualitative comparison results between EMF and KMF.

5 Discussion

5.1 Refactoring Impact

The first major consequence of removing the runtime dependencies with EMF is that all the methods defined in `EObject` are now unavailable. This could have a significant impact on the existing code that uses these methods. In order to limit the refactoring impact of this removal, we re-implemented the `eContainer` mechanism of EMF in our generated code.

Another important design choice we made for KMF was to use Scala as the default language for the generation and use of the code. Java has also been considered as a language since Scala code is fully compatible with Java. However, Scala code is not always friendly to use from a Java program[13]. To ease the use

[12] This experiment can be download http://goo.gl/9Huwa (for eclipse project) or `http://goo.gl/OsWRo` (for the maven project)

[13] To give an idea, a Scala list built by concatenating the empty list (Nil) and the element 1 would be written `1::nil` in Scala. The construction of the same Scala list in Java would yield `$colon$colon$.MODULE$.apply((Integer) 1, nil);`

of KMF in a Java environment, we also provide a standard Java API, which in particular exposes Java lists, by duplicating some methods that are suffixed with "4J". That requires all model navigation-related code to be rewrote to use these new methods. The dual strategy could of course have been implemented: generating Java code and exposing in addition a Scala API.

The main rationale behind the choice of Scala was that the Scala standard library provides many facilities that are useful for Models(@runtime). In particular, the systematic introduction of Scala `Option` for each optional (*i.e.* having lower bounds equals to 0) attribute or reference implies to explicitly test if the element is defined or not, in a neater way than `if (myRef == null)` in Java, and in a safer way than a `NullPointerException` popping at runtime. Here again, this mechanism enforces developers to consider the optional aspect of these elements and avoids lots of null-checks, but requires a deep refactoring.

5.2 Limitations

KMF has formerly been developed and tested using the Kevoree metamodel originally designed with EMF tools. This first step allowed for setting up the basis for model, loader and serializer generation. The use of a fairly different metamodel (from Kermeta [12]) highlighted some missing features in the generation process, and strongly helped in improving KMF. However, KMF still has some limitations.

For instance, reverse relations have already be flagged as missing in the generated code.

EMF allows model elements to have some relations with elements from other metamodels (references, attribute types, inheritance, etc). This mechanism has been partially realized on a concrete use case, but it still need improvements and implementation discussions.

Moreover, the generation of loaders and serializers relies on containment relations between model elements, and it requires a root container element. Until now, we considered metamodels that have only one single model element as root of the containment tree, but this is not the general case. Indeed, all model elements must have a container, but not necessarily under a single root for the metamodel. There could be several containment roots (and several containment trees) in a metamodel, with references to each other, but also across different metamodels. Generators of loaders and serializers are not ready to accept such kind of metamodels, but meeting this requirement is already considered as future work.

KMF addresses performances issues of models in memory (heap). Complementary approaches like CDO addresses the management of models in persistence memory (databases). The CDO (Connected Data Objects) Model Repository [14] is a distributed shared model framework for EMF models and metamodels. CDO has a 3-tier architecture supporting EMF-based client applications, featuring a central model repository server and leveraging different types of pluggable data

[14] `http://www.eclipse.org/cdo/`

storage back-ends like relational databases, object databases and file systems. The default client/server communication protocol is implemented with the Net4j Signalling Platform. This solution offers tools to easily do collaborative work and history management. However, it uses EMF as a local representation. Consequently it inherits a part of the drawbacks coming from EMF (e.g. dependencies for client side, memory footprints). Moreover the use of external server to manage history is not useful for pervasive systems that may have a sporadic network and even if it can embed server side on client, the overhead of the dependencies is not suitable for lightweight systems.

6 Conclusion

This position paper has discussed the needs for adapting the *de facto* standard in the MDE community, *i.e.* the Eclipse Modelling Framework (EMF), for a more dynamic usage of models in the context of Models@Runtime. After highlighting requirements related to Models@Runtime, this paper has presented an initial adaptation of EMF, named Kevoree Modelling Framework (KMF), implemented in Scala and generating code for this language. Even if KMF only supports XMI serialisation, it provides a significant speedup on model creation, model (un)marshalling and model cloning. It also has a lighter memory footprint than the reference implementation, and its runtime dependencies are limited to the Java and Scala libraries, whereas the EMF generated code has tight dependencies to Eclipse and Equinox. This significantly hinders the reusability of the EMF code outside Eclipse, while KMF code can run Eclipse-free on various Java virtual machines. Finally, and unlike EMF which is not thread-safe, KMF provides a built-in support for in-memory safe concurrent access to models.

KMF is still at an early stage of existence and needs to be improved through usage. Future work on KMF already addresses the limitations and points discussed in section 5. Independently from the improvement of existing features of KMF, we think that additional tools could promote its adoption.

Set Operations. Model merging or model comparison are common operations implemented by tools that use models as representations of their internal data. Implementing mergers or comparators is often a complex, lengthy and error prone task. As a future work, we plan to offer the possibility to generate meta-model specific set operations such as union, difference or intersection. These operations could decrease the complexity of implementing model mergers.

Customizable Generation Plugin. In its current implementation, the plugin allows for generation of all features (model, cloner, loader and serializer) or only model and cloner. This customization of the plugin behavior will be improved to enable the separate generation of each feature. Moreover, the loader and serializer generators are hard coded in the plugin. It is thus problematic to use other generators to create loaders and serializers that use another serialization format (namely XMI). In the future, we plan to improve the plugin parameterization to allow users to change the generators. This will also enable the seamless integration of other generators (e.g. to create set operations).

References

1. Ballagny, C., Hameurlain, N., Barbier, F.: Mocas: A state-based component model for self-adaptation. In: Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems, SASO 2009, San Francisco, California, USA, September 14-18, pp. 206–215. IEEE Computer Society (2009)
2. Blair, G.S., Bencomo, N., France, R.B.: Models@run.time. IEEE Computer 42(10), 22–27 (2009)
3. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009)
4. Eysholdt, M., Behrens, H.: Xtext: implement your language faster than the quick and dirty way. In: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, SPLASH 2010, pp. 307–309. ACM, New York (2010)
5. Fouquet, F., Barais, O., Jézéquel, J.-M.: Building a kermeta compiler using scala: an experience report. In: Workshop Scala Days 2010, Lausanne, Switzerland (2010)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional (1994)
7. Georgas, J.C., van der Hoek, A., Taylor, R.N.: Using architectural models to manage and visualize runtime adaptation. Computer 42(10), 52–60 (2009)
8. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Derivation and Refinement of Textual Syntax for Models. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 114–129. Springer, Heidelberg (2009)
9. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. Computer 36(1), 41–50 (2003)
10. Merkle, B.: Textual modeling tools: overview and comparison of language workbenches. In: Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion, SPLASH 2010, pp. 139–148. ACM, New York (2010)
11. Morin, B., Barais, O., Nain, G., Jézéquel, J.-M.: Taming Dynamically Adaptive Systems with Models and Aspects. In: 31st International Conference on Software Engineering (ICSE 2009), Vancouver, Canada (May 2009)
12. Muller, P.-A., Fleurey, F., Jézéquel, J.-M.: Weaving Executability into Object-Oriented Meta-languages. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 264–278. Springer, Heidelberg (2005)

13. Odersky, M., Spoon, L., Venners, B.: Programming in Scala: A Comprehensive Step-by-step Guide, 1st edn. Artima Incorporation, USA (2008)
14. Rothenberg, J., Widman, L.E., Loparo, K.A., Nielsen, N.R.: The Nature of Modeling. In: Artificial Intelligence, Simulation and Modeling, pp. 75–92. John Wiley & Sons (1989)
15. Seehusen, F., Stølen, K.: An Evaluation of the Graphical Modeling Framework (GMF) Based on the Development of the CORAS Tool. In: Cabot, J., Visser, E. (eds.) ICMT 2011. LNCS, vol. 6707, pp. 152–166. Springer, Heidelberg (2011)
16. Seinturier, L., Merle, P., Rouvoy, R., Romero, D., Schiavoni, V., Stefani, J.-B.: A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. In: Software: Practice and Experience (2011)

Automated and Transparent Model Fragmentation for Persisting Large Models

Markus Scheidgen[1], Anatolij Zubow[1], Joachim Fischer[1], and Thomas H. Kolbe[2]

[1] Humboldt Universität zu Berlin, Department of Computer Science
{scheidge,zubow,fischer}@informatik.hu-berlin.de
[2] Technical University Berlin, Institute for Geodesy and Geoinformation Science
thomas.kolbe@tu-berlin.de

Abstract. Existing model persistence frameworks either store models as a whole or object by object. Since most modeling tasks work with larger aggregates of a model, existing persistence frameworks either load too many objects or access many objects individually. We propose to persist a model broken into larger fragments.

First, we assess the size of large models and describe typical usage patterns to show that most applications work with aggregates of model objects. Secondly, we provide an analytical framework to assess execution time gains for partially loading models fragmented with different granularity. Thirdly, we propose meta-model-based fragmentation that we implemented in an EMF based framework. Fourthly, we analyze our approach in comparison to other persistence frameworks based on four common modeling tasks: create/modify, traverse, query, and partial loads.

We show that there is no generally optimal fragmentation, that fragmentation can be achieved automatically and transparently, and that fragmentation provides considerable performance gains.

1 Introduction

Modeling frameworks (e.g. the Eclipse Modeling Framework (EMF) [22] or Kermeta [11]) can only work with a model when it is fully loaded into a computer's main memory (RAM), even though not all model objects are used at the same time. This limits the possible size of a model. Modeling frameworks themselves provide only limited capabilities to deal with large models (i.e. resources and resource lazy loading in EMF [22]). Model persistence frameworks (e.g. Connected Data Objects (CDO) [1]), on the other hand, store models in databases and load and unload single model objects on demand. Only those objects that are used at the same time need to be maintained in main memory at the same time. This allows one to work with models larger than the main memory can hold otherwise.

We claim that existing model persistence solutions may provide a main memory efficient solution to the model size issue, but not a time efficient one. In this paper, time efficiency always relates the time it takes to execute of one of four abstract modeling tasks. These tasks are (i) creating/modify models, (ii) traverse models (e.g. as necessary during model transformation), (iii) query models, and (iv) partially loading models (i.e. loading a diagram into an editor).

An obvious observation is that some of these modeling tasks (especially traversing models and loading parts of models) require to load large numbers of model objects eventually. Existing persistence frameworks, store and access model objects individually. If a tasks requires to load a larger part of the model, all its objects are still accessed individually from the underlying database. This is time consuming.

Our hypothesis is that modeling tasks can be executed faster, if models are mapped to larger aggregates within an underlying database. Storing models as aggregates of objects and not as single objects reduces the number of required database accesses, or as Martin Fowler puts it on his blog: *"Storing aggregates as fundamental units makes a lot of sense* [...]*, since you have a large clump of data that you expect to be accessed together"*, [7]. This hypothesis raises three major questions: Do models contain aggregates that are often *accessed together*? How can we determine aggregates automatically and transparently? What actual influence on the performance has the choice of concrete aggregates?

To answer these question, we will proceed as follows: First (section 2), we look at three typical modeling applications: which model sizes they work with and what concrete modeling tasks they perform predominantly. This will give us an idea of what aggregates could be and how often objects can be expected to be actually accessed as aggregates. Secondly (section 3), we will present our approach to finding aggregates within models. This approach is based on fragmenting models along their containment hierarchy. We will reason that most modeling tasks need to access sub-trees of the containment hierarchy (fragments). In the related work section 4, we present existing model persistence frameworks and interpret their strategies with respect to the idea of fragmentation. Furthermore, we discuss key-value stores as a basis for persisting fragmented models. The following section provides a theoretical analysis and upper bound estimation for possible performance gains with optimal fragmentation. In section 6, we finally present a framework that implements our fragmentation concept. The next section is the evaluation section: we compare our framework to existing persistence frameworks with respect to time and memory efficient execution of the four mentioned abstract modeling tasks. Furthermore, we use our framework to measure the influence of fragmentation on performance to verify the analytic considerations from section 3. We close the paper with further work and conclusions.

2 Applications for Large Models

In this section, we look at examples for three modeling applications. We do this for two reasons. The first reason is to discuss the actual practical relevance of large models. The second reason is to identify model usage patterns: which of the four modeling tasks (create, traverse, query, partial load) are actually used, in what frequency, and with what parameters. At the end of this section, we provide a tabular summary of our assessment.

2.1 Software Models

Model Driven Software Development (MDSD) is the application that modelling frameworks like EMF were actually designed for. In MDSD all artifacts including traditional software models as well as software code are understood as models [23], i.e. directed labelled graphs of typed nodes with an inherent containment hierarchy.

Model Size: Since models of software code (code models) provide the lowest level of abstraction, we assume that models of software code are the largest software models. In [18], we give an approximation for the size of code models based on counting abstract syntax tree nodes in the Linux kernel and analyzing the Linux kernels GIT repository. We also transferred all ratios learned from the Kernel to other OS software projects and publicly reported LOC counts. The results are presented in Fig. 1.

Usage Patterns: There are two major use cases in today's software development: editing and transforming or compiling. The first use case is either performed on diagrams (graphical editing) or on compilation units (e.g. Java-files, textual editing). Diagram contents roughly corresponds to package contents. Both packages and compilation units are sub-trees within the containment tree of a software model. Transformations or compilations are usually either done for the whole model or again on a per package or compilation unit basis. Within these packages or compilation units, the (partial) model is traversed. A further use-case is analysis. Analysis is sometimes performed with single queries. But due to performance issues, model analysis is more often performed by traversing the model and by executing multiple queries with techniques similar to model transformations. Software models are only accessed by a few individuals at the same time.

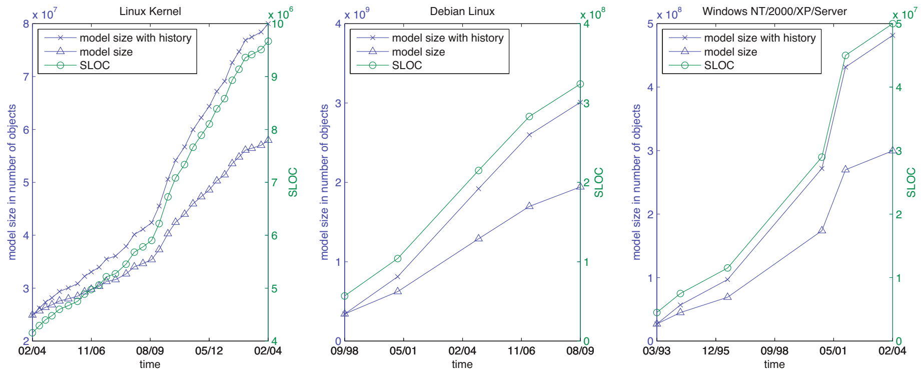


Fig. 1. Rough estimates for software code model sizes based on actual SLOC counts for existing software projects

2.2 Heterogeneous Sensor Data

Sensor data usually comprises time series of measured physical values in the environment of a sensor. Our research group build the *Humboldt Wireless Lab* [24], a 120 node wireless sensor network that produces heterogeneous sensor data: data from a 3 axis accelerometers, data from monitoring all running software components (mostly networking protocols), and other system parameters (e.g. CPU, memory, or radio statistics). We represent and analyze this data with EMF based models ([19]).

Model Size: HWL's network protocols and system software components provide 372 different types of data sets. Each data set is represented as an XML document. Per second each node in the network produces XML entities that translate into an average of 1120 EMF objects. A common experiment with HWL involves 50 nodes and measures of a period of 24 h. During such an experiment, the network produces a model of 5×10^9 objects.

Usage Patterns: There are two major use-cases: recording sensor data and analyzing sensor data. Recording sensor data means to store it faster than it is produced and (if possible) in a manner that supports later analysis. Sensor data is rarely manipulated. Analysis means to access and traverse individual data sets (mostly time series). Each data set or recorded set of data sets is a sub-tree in the sensor data model. Recording and analysis is usually performed by only a single (or a few) individuals at the same time.

2.3 Geo-spatial Models

3D city models are a good example for structured geo-spatial information. The CityGML [8] standard, provides a set of XML-schemata (building upon other standards, e.g. GML) that function as a meta-model. CityGML models represent the features of a city (boroughs, streets, buildings, floors, rooms, windows, etc.) as a containment hierarchy of objects. Geo spatial models usually come in different levels of details (LOD); CityGML distinguishes 5 LODs, 0-4 [8].

Model Size: As for many cities, a CityGML model is currently established for Berlin [21]. The current model of Berlin covers all of Berlin, but mostly on a low-medium level of detail (LOD 1-2). To get an approximation of the model's size, we counted the XML entities. The current Berlin model, contains 128×10^6 objects. Based on numbers and average sizes per feature sizes in the Berlin model, a complete LOD 3-4 model of Berlin would consist of 10^9 objects. Extrapolating numbers to the world's population that lives in cities, a LOD3-4 *world 3D city model* would contain 10^{12}.

Usage Patterns: Compared to model manipulation, model access is far more common and its efficient execution is paramount. If accessed, users usually load a containment hierarchies (sub-tree) corresponding to a given set of coordinates or address (geographic location): partial loads. Queries for distinct feature characteristics within a specific geographic location (i.e. with-in such a partial load) are also common. Geo-spatial models are accessed by many people at the same time.

Summary

The following table summarizes this section. Two + signs denote that execution times of the respective tasks are vital for the success of the application; a single + denotes that the task is executed often, but performance is not essential; a − denotes that the task is of minor importance.

application	model size	create/mod.	traverse	query	partial load
software models	$0 - 10^9$	+	++	+	+
sensor data	10^9	++	++	-	++
geo-spatial models	$10^9 - 10^{12}$	-	-	++	++

3 Model Fragmentation

3.1 Fragmentation in General

All models considered in this paper can be characterized as directed labeled graphs with a fix spanning-tree called *containment hierarchy*. In EMF based models, the containment hierarchy consists of *containment references*; other graph edges are *cross-references*.

Model fragmentation breaks (i.e. *fragments*) a model along its containment hierarchy. All *fragments* are disjoint; no object is part of two fragments. Fragmentation is also always complete, i.e. each object is part of one fragment. The set of fragments of a model is called *fragmentation*. References between fragments are called inter-fragment and references within a fragment are called intra-fragment references. [1]

3.2 Fragmentation Strategies

Originally a model is not fragmented; once it was fragmented, the fragmentation needs to be maintained when the model is modified. Further, we have to assume that fragmentation has an influence on performance (refer to sections 5 and 7). We denote a set of algorithms that allows us to create and maintain a fragmentation as *fragmentation strategy*.

There are two trivial strategies: *no fragmentation* and *total fragmentation*. No fragmentation means the whole model constitutes of one fragment, such as in EMF (without resources). Total fragmentation means each object constitutes its own fragment. There are as many fragments as objects in the model. This strategy is implemented by existing persistence frameworks like CDO.

[1] Based on these characteristics, fragments can be compared to EMF's resources (especially with containment proxies); refer to section 6, where we use resources to realize fragmentation.

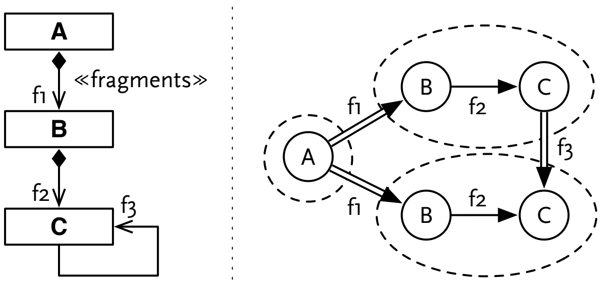


Fig. 2. Example meta-model (left) and model (right). In the model: dashed ellipses denote fragments, double lines inter- and normal lines intra-fragment references. The references of feature *f1* determine the fragments, the reference of *f3* is a inter-fragment cross-reference by accident.

3.3 Meta-model Based Fragmentation

In this paper, we propose and use *meta-model based fragmentation* as fragmentation strategy. A meta-model defines possible models by means of classes and their attribute as well as reference features. Whereby, the meta-model determines which reference features produce containment and which produce cross-references. The meta-modeler already uses containment reference features to aggregate related objects.

In meta-model based fragmentation, we ask the meta-modeler to additionally mark those containment reference features that should produce inter-fragment containment references. This way, the meta-model determines where the containment hierarchy is broken into fragments, and it becomes easy to create and maintain fragmentations automatically and transparently (ref. to section 6). Only containment reference features determine fragmentation, cross-references can become inter-fragment references by accident. See Fig. 2 for an example.

4 Related Work

4.1 Model Persistence

EMF: Models are persisted as XMI documents and can only be used if loaded completely into a computer's main memory. EMF realizes the *no fragmentation* strategy. The memory usage of EMF is linear to the model's size.

There are at least three different approaches to deal with large EMF models: (1) EMF resources, where a resource can be a file or an entry in a database; (2) CDO [1] and other object relational mappings (ORM) for Ecore; (3) morsa [15] a EMF data-base mapping for non-relational databases.

First, EMF resources [22]: EMF allows clients to fragment a model into different resources. Originally, each resource could only contain a separate containment hierarchy and only inter-resource cross-references were allowed. But since

EMF version 2.2 containment proxies are supported. EMF support lazy loading: resources do not have to be loaded manually, EMF loads them transparently once objects of a resource are navigated to. Model objects have to be assigned to resources manually (*manual fragmentation*). To actually save memory the user has to unload resources manually too. The framework MongoEMF [10] maps resources to entries in a MongoDB [16] database.

Secondly, CDO [1]: CDO is a ORM for EMF. [2] It supports several relational databases. Classes and features are mapped to tables and columns. CDO was designed for software modeling and provides transaction, views, and versions. Relational databases provide mechanisms to index and access objects with SQL queries. This allows fast queries, if the user understands the underlying ORM.

Thirdly, morsa [15]: Different to CDO, Morsa uses mongoDB [16], a *NoSQL* database that realizes a key-value store (see below). Morsa stores objects, their references and attributes as JSON documents. Morsa furthermore uses mongoDB's index feature to create and maintain indices for specific characteristics (e.g. an objects meta-class reference).

4.2 Key-Value Stores

Web and cloud computing require scaleability (replication and sharding[3] in a peer-to-peer network) from a database, and traditional ACID [9] properties can be sacrificed if the data store is easily distributeable. This explains the popularity of *key-value stores*. Such stores provide only a simple map data structure: there are only keys and values. For more information and an comparison of existing key-value stores refer to [14].

Model fragmentation does not need any complex database structure, since a fragment's content can be serialized (e.g. with XMI) and fragments can be identified by keys (e.g. URIs). Key-value stores on the other hand provide good scaleability for large models (sharding) or for parallel access (replication).

There are three different applications that inspired three groups of key-value stores. First, there are web applications and the popular MongoDB [16] and CouchDB [3] databases. These use JSON documents as values and provide additional indexing of JSON attributes.

Secondly, there is cloud computing and commercial Google Big-Table [4] and Amazon's Dynamo [6] inspired data stores. HBase [12] and Cassandra [13] are respective open source implementations. Those databases strive for massive distribution, they provide no support for indexing inner value attributes, but integrate well into map-reduce [5] execution frameworks, such as Hadoop (HBase is Hadoop's native data store).

A third application is high performance computing. Scalaris [20] is a key-value store optimized for massive parallel, cluster, and grid computing. Scalaris

[2] Lately, CDO also supports non-relational databases, such as MongoDB [16]. Such features were not evaluated in this paper; but one can assume characteristics similar to those of Morsa.

[3] *Sharding* denotes horizontal partitioning of a database, i.e. to put different parts of the data onto different nodes in the network

provides mechanisms for consistency and transactions and brings some ACID to key-value stores.

5 Possible Performance Gains from Model Fragmentation

In this section, we analyze the theoretically possible execution times of partially loading models with fragmentations of different granularity. This includes an assessments for performance gains from optimal fragmentation strategies compared to no or total fragmentation.

To keep this analysis simple, we have to make two assumptions that will probably seldom hold in reality, but still lead to analysis results that provide reasonable upper bounds for possible gains. The first assumption: we only consider fragmentations where all fragments have the same size f. This means a fragmentation for a model of size m consist of $\lceil m/f \rceil$ fragments[4]. The second assumption: all fragmentations are optimal regarding partial loads. This means to load a model part of size l, we only need to load $\lceil l/f \rceil$ fragments at most.

To determine the execution time for partial loading depending on the parameters model size m, fragment size f and size of the model part l, we need two functions that determine the time it takes to read and parse a model and to access a value in a key value store. The read and parse function is linear depending on parsed model size s: $parse(s) = \mathcal{O}(s)$, the access function is logarithmic depending on the number of keys k: $access(k) = \mathcal{O}(log(k))$. Most key-value stores, including HBase (that we use for our implementations) provide $\mathcal{O}(log)$ accesses complexity (ref. also to Fig. 4).

With the given assumptions, parameters, and functions the time to execute a partial load is:

$$t_{m,f}(l) = \overbrace{\left\lceil \frac{l}{f} \right\rceil}^{\text{number of fragments to load}} \underbrace{\left(access(\left\lceil \frac{m}{f} \right\rceil) + parse(f) \right)}_{\text{time to load one fragment}}$$

To actually use this cost function, we need concrete values for $parse$ and $access$. We measured the execution times for $parse$ with EMF's XMI parser for models of various sizes and fit a linear function to the measured values (Fig. 3). For $access$ we measured the execution time for accessing keys in HBase for database tables with various numbers of keys k. For $k < 10^6$ we use a linear function and for $k \geq 10^6$ a logarithmic function as a fit (Fig. 4).

Now, we can discuss the influence of fragment size f on $t_{m,f}(l)$. First, we use a model of size $m = 10^6$ and vary $f \in \{10^0, \ldots, 10^6\}$. Fig. 5 shows the computed times t over loaded model objects l for the different fragment sizes f. We can observe four things. First, there is no optimal fragment size. Depending on the number of loaded objects, different fragment sizes are optimal. But intermediate fragment sizes provide good performance. With fragment size $f = 10^2$ for

[4] $\lceil x \rceil$ denotes the ceiling of x.

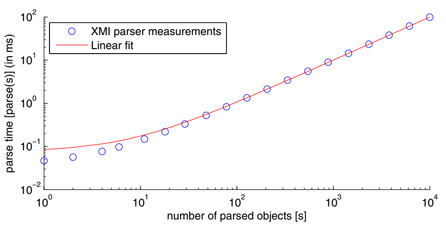


Fig. 3. EMF's XMI parser performance

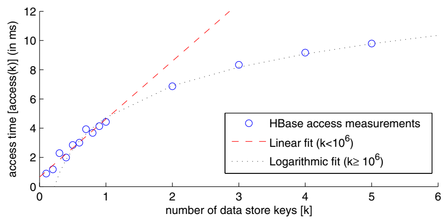


Fig. 4. HBase accesses performance

example, all partial loads take three times the optimal time at most. Second, total fragmentation ($f = 1$) requires roughly 100 times more time than optimal fragmentation, when larger numbers of objects $\geq 10^2$ are loaded. Thirdly, no fragmentation ($f = m$) is only a time efficient option, if we need to load almost all of the model. But in those cases no fragmentation is usually not practical for memory issues. Fourthly, for small partial models total fragmentation is far better than no fragmentation, for large partial models no fragmentation provides better performance.

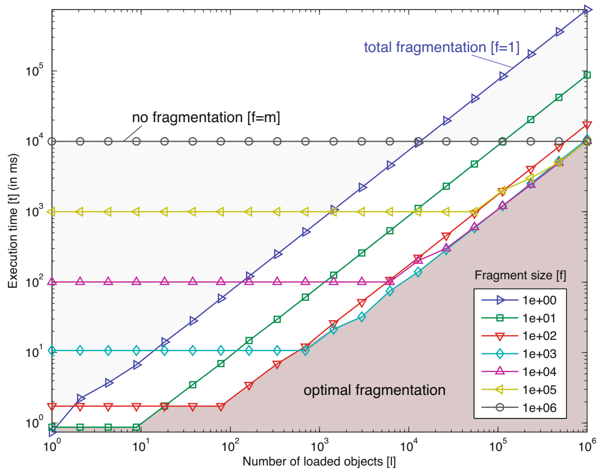


Fig. 5. Computed execution times for partial loads from a model with 10^6 objects and fragmentations of different granularity

6 Implementation of Model Fragmentation

In this section, we present the EMF based persistence framework EMFFrag [17] which implements the presented meta-model based fragmentation strategy (refer to section 3).

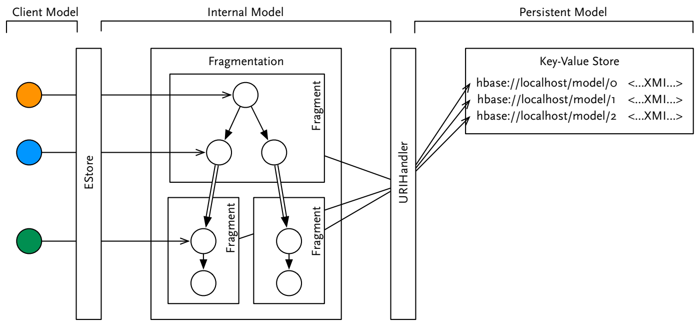

Client Model
Internal Model
Persistent Model
EStore
Fragmentation
Fragment
Fragment
Fragment
URIHandler
Key-Value Store
hbase://localhost/model/0 <...XMI...>
hbase://localhost/model/1 <...XMI...>
hbase://localhost/model/2 <...XMI...>

Fig. 6. EMFFrag partially loads a persisted model as internal model of dynamic EMF objects and exposes the model as client model via EMF generated model code with feature delegation

Design Goal: The main goal in our implementation is to (re)use EMF resource as much as possible. EMF resources already provide many required functionalities: they realize partial model persistence, resources manage inter-resource references through proxies, resources lazy-load, they can be added and deleted, and objects can be moved between resources. EMFFrag extends the existing implementations of EMF resources. EMFFrag could be realized with a very small code base of less than 800 lines of code.

Underlying Key-Value Store: EMFFrag uses a simple interface that abstracts from concrete key-value stores. We provide an implementation for HBase (this was used for all measurements in this paper). EMFFrag implements EMF's `URIHandler` interface to realize key-value store values as resources. Each fragmented model is stored in its own table.

Fragments and Fragmentation: EMF `XMIResource`s are used as fragments and `ResourceSet`s act as fragmentations. The model is internally realized as a purely dynamic (no generated sources) EMF model.

Transparent Load and Unload of Fragments: Fragments, Fragmentations, and internal model are hidden from clients (ref. to Fig. 6). Clients use the model through the usual EMF generated interfaces and classes. Those are configured with reflective feature delegation to an `EStore` ([22] explains the concept). EMFFrag's `EStore` implementation simply delegates all calls to internal objects. If necessary, it creates an internal object for each client created object, and a client object for each internal object. Client objects hold references to their internal counterparts. Fragments manage client objects that correspond to the internal objects they contain via Java's *weak references*. When clients loose all strong references to a fragment's contents, the JVM collects the client objects as garbage (despite existing weak references) and notifies the owning fragment. Thus, fragments know if clients hold references to their objects, and they can safely unload once no more client reference to their contents exist.

Inter-fragment Containment References: Client model classes have to be generated with enabled containment proxies (see [22]) to allow containment references between resources (i.e. fragments). Users can use EMF Ecore annotation to mark containment reference features as inter-fragment features. When EMFFrag's `EStore` implementation delegates a call that manipulates an inter-fragment containment feature, it creates or deletes fragments accordingly and puts objects into their respective fragments.

Inter-fragment Cross References: EMF persists references between XMI resources with URIs. The first part of an URI identifies the resource (i.e. the fragment within a key-value store). The second URI part (URI fragment part) identifies the referenced object within the containing resource. For all inter-fragment containment references and for cross references within a fragment EMF's default *intrinsic ID's* [22] are used.

Intrinsic IDs are similar to XPath expressions and identify an object via its position in the containment hierarchy. Intrinsic IDs cannot be used for inter-fragment cross references: when an object is moved, its intrinsic ID (URI fragment) changes and all persisted referencing object use invalid URIs. For this reason EMFFrag uses model-wide unique *extrinsic IDs* (an existing EMF functionality). EMFFrag maintains a secondary index (i.e. another table in the key-value store) that maps extrinsic IDs to respective intrinsic IDs. When an object moves this entry is updated and all cross-references are updated automatically. Extrinsic IDs and secondary index are only maintained for objects that are actually cross referenced from another fragment to keep the index small.

7 Evaluation

This section has two goals. First, we want to compare our fragmentation approach to other model persistence frameworks. Secondly, we want to verify our findings from section 5. All measurements were performed on a Notebook computer with Intel Core i5 2.4GHz CPU, 8 GB 1067 MHz DDR3 RAM, running Mac OS 10.7.3. All experiments were repeated at least 20 times, and all present results are respective averages. Code executing all measurements and all measured data can be downloaded as part of EMFFrag [17].

7.1 Fragmentation Compared to Other Persistence Frameworks

To compare fragmentation to EMF's XMI implementation, CDO, and Morsa, we measured execution time for the three tasks (i) create/modify, (ii) traverse, and (ii) query. To analyze traverse and query, we used example models from the Grabats 2009 contest [2] as benchmarks. Those were already used to compare Morsa with XMI and CDO here [15]. There are five example models labeled *set0* to *set4* and they all model Java software based on the same meta-model. Please note: even though the models increase in size, their growth is not linear and the internal model structure is different. To measure create/modify performance, we used a simple test model. We don't provide any comparative measures for partial loads. Partial loads are extensively measured for EMFFrag in the next section.

Fig. 12 shows the number of fragments that each framework produces for each model. Morsa and CDO implement total fragmentation and the number of fragments is also the number of objects in the model. For XMI there is always only one fragment, because it implements *no fragmentation.* For EMFFrag, we provided two different meta-model based fragmentations. The first one puts each Java compilation unit and class file into a different fragment (labeled *EMFFrag coarse*). The second one additionally puts the ASTs for each method block into a different fragment (*EMFFrag fine*). The number of fragments differs significantly for *set2* and *set3* which have to contain a lot of method definitions. We could not measure CDO's performance for *set3* and *set4*: the models are too large to be imported with a single CDO transaction, and circular cross-references do not allow us to import the model with multiple transactions.

Create/Modify: To benchmark the performance of instantiating and persisting objects, we used a simple one class. We created test models with 10^5 objects, a binary containment hierarchy, and two different densities of cross references: one cross reference per object and no cross references. We used a transaction size of 10^3 objects for CDO and a fragment size of 10^3 for EMFFrag.

Fig. 7.1 shows the average number of objects that could be persisted within one second. The number of cross-references has only a minor influence on the performance of CDO, Morsa, and EMFFrag. EMFFrag is a little slower than XMI depending on the fragment size (Fig. 8). CDO and Morsa (both based on complex indices that have to be maintained) can only create less than one tenth of the objects per seconds that could be created with XMI and EMFFrag. Fig. 8 shows EMFFrag's create performance for different fragment sizes.

Traverse: Fig. 9 shows the measurement results in traversed objects per second. XMI performs well for small models, but numbers deteriorate for large models. Interestingly, Morsa and CDO both use *total fragmentation* and achieve both a comparable low 4,500 objects per second. EMFFrag performs depending on the number of fragments: the less fragments the better. With the Grabats models, fragmentation gives us about 10-18 times the number of objects per second traversed than with CDO or Morsa do.

Query: The Grabats contest also provides an example query: find all Java type declarations that contain a static method which has its containing type as return type. Depending on the persistence framework, queries can be implemented in different ways. With XMI and EMFFrag there are no indices that would help to implement the query and we have to traverse the model until we found all type declarations. CDO allows us to use SQL to query and Morsa provides a meta-model class to objects index. We measured both: executing the queries with these specific query mechanisms and with the previously mentioned traverse based implementation.

The results are shown in Fig. 10. XMI performs badly for large models. CDO and Morsa with SQL and meta-class index perform best. But even though

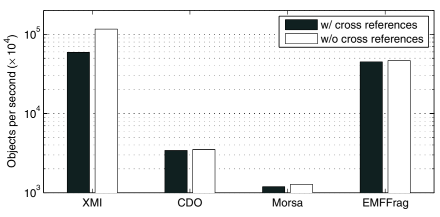


Fig. 7. Number of objects per second that can be created with the different persistence frameworks

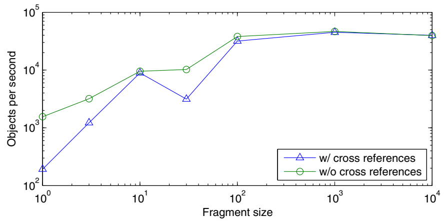


Fig. 8. Number of objects per second that can be created with the EMFFrag and different fragmentation granularity

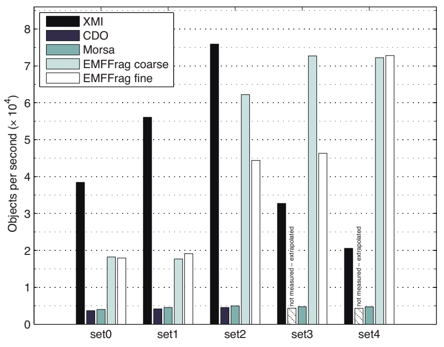


Fig. 9. Number of model objects traversed per second during traversing the different Grabats models

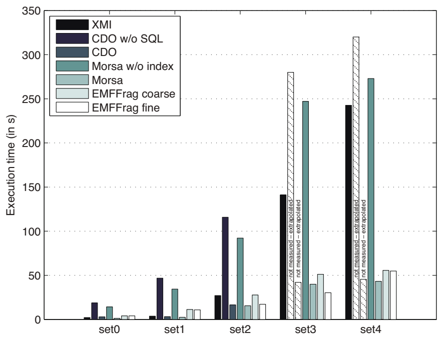


Fig. 10. Execution time for querying the different Grabats models with the example query

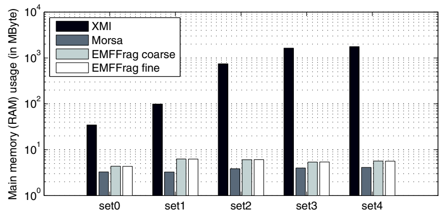


Fig. 11. Memory usage during traversal of the different Grabats models

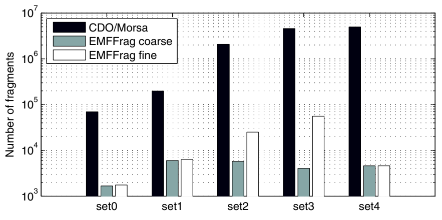


Fig. 12. Number of fragments used by the different persistence frameworks

EMFFrag needs to traverse the model its performance is similar to CDO and Morsa. For *set3* and fine fragmentation, EMFFrag even outperforms Morsa's index. Remember, with the fine fragmentation, EMFFrag does not need to load any method bodies to execute the query (partial load). Using the traverse implementation, CDO's and Morsa's performance difference to EMFFrag is similar to the measures for model traverse (here we basically perform a partial traverse).

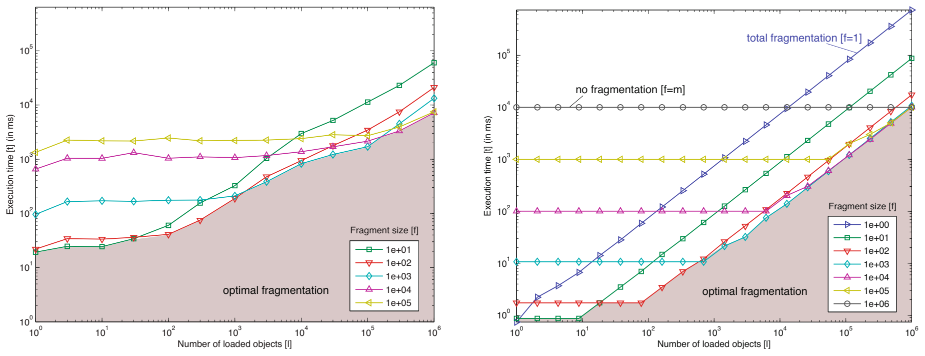


Fig. 13. Execution times for loading model parts with different fragmentation granularity measured with EMFFrag (left) and analytical (right)

Memory Usage: During model traverse, we also measured the memory usage (Fig. 11). XMI's memory usage is proportional to model size, because it needs to load the full models into memory. All other approaches need a comparable constant quantity of memory independent of model size.

7.2 The Influence of Fragmentation on Partial Load Performance

In section 5, we looked at fragmentation analytically and provided a plot (Fig. 13, right) that describes the expected influence of fragmentation granularity on partial load execution times. New, we create the same plot, but based on data measured with EMFFrag. For this purpose, we used the same simple meta-model as before (to measure create/modify) and generated models of size 10^6 with different fragment sizes f. We measured the execution times for loading parts of different sizes l. The results are presented in Fig. 13, left.

The plots show a similar picture with comparable values. Although, the measured times are generally larger due to additional EMFFrag implementation overhead that was not considered in our theoretical examination.

8 Future Work

Sorted and Distributed Key-Value Stores: Our fragmentation strategy is based on unsorted key-value store accesses with $\mathcal{O}(log)$ complexity. Neither our analysis, nor our implementation EMFFrag, or our evaluation consider sorted key-value stores that allow us to access sequential keys with constant time (scans). Neither did we consider distributed key-value stores which would allow us parallel access. Key-value stores are easily distributed in peer-to-peer networks. This is done for two scaleability reasons: replication (allows more users to access the same data in less time) and sharding (distributes data to allow users faster and larger storage). Fragmentation can have an influence on both.

Transactions: If multiple user access/modify a model transactions become a necessity. Transaction can either be provided by the underlying data store (e.g. with Scalaris [20]) or can be implemented into EMFFrag. On non-distributed data stores, the usual transaction mechanisms can be implemented. More interesting is to explore the influence of fragmentation on transactions (and versioning), because fragmentation granularity also determines the maximum transaction granularity.

Large Value Sets: In large models, single objects can become very large themselves if they hold large sets of attribute values and references. CDO maps an object's feature values to individual entries in a database table and can manage such objects, but does this slowly. EMFFrag (and Morsa), on the other hand, consider objects as atomic entities and large object can become a performance burden. We need to extend the fragmentation idea to large value sets. Similar to all consideration in this paper, strategies for large value sets have to be optimized and evaluated for the abstract tasks manipulation, iteration (traverse), indexed access (query), and range queries (partial load).

9 Conclusions

Large software models consist of up to 10^9 objects. Models from other application can have a size of up to 10^{12} objects. Traversing models and loading larger aggregates of objects are common tasks (section 2). Depending on fragment size, partially loading models can be done faster than loading whole models or loading models object by object. There is no optimal fragment size, but intermediate fragment sizes provide a good approximation (sections 5 and 7). We provide a persistence framework that enables automatic and transparent fragmentation, if appropriate containment features are marked as fragmentation points in the meta-model (sections 3 and 6). We compared our framework to existing frameworks (EMF's XMI implementation, CDO and Morsa) and our framework performs significantly better for the tasks create/manipulate, traverse, and partial loads. Execution times are 5 to 10 times smaller. Model queries (that favor object-by-object based model persistence with indexes, such as in CDO and Morsa) can be executed with comparable execution times (section 7). All together, fragmentation combines the advantages of both worlds, low memory usage and fast queries like with CDO or Morsa, and traverse and partial load execution times similar to those of XMI.

Model fragmentation also determines the granularity of transactions, which can be a disadvantage. Further problems are single objects with features that can hold large value sets; the fragmentation approach has to be extended for fragmentation of such value sets (section 8). Our framework stores fragments in key-value stores. Those scale easily (both replication and sharding is supported) and integrate well with peer-to-peer computation schemes (e.g. map-reduce). Fragmentation is therefore a good preparation for modeling in the cloud applications (section 4).

References

1. Connected Data Objects (CDO), http://www.eclipse.org/cdo/
2. Grabats 2009, 5th International Workshop on Graph-based Tools: A Reverse Engineering Case Study (July 2009), http://is.tm.tue.nl/staff/pvgorp/events/grabats2009
3. Anderson, J.C., Lehnardt, J., Slater, N.: CouchDB: The Definitive Guide Time to Relax, 1st edn. O'Reilly Media, Inc. (2010)
4. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A Distributed Storage System for Structured Data. In: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2006, vol. 7, p. 15. USENIX Association, Berkeley (2006)
5. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Commun. ACM 51, 107–113 (2008)
6. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's Highly Available Key-Value Store. In: Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles, SOSP 2007, pp. 205–220. ACM, New York (2007)
7. Fowler, M.: Aggregate Oriented Databases (January 2012), http://martinfowler.com/bliki
8. Gröger, G., Kolbe, T.H., Czerwinski, A., Nagel, C.: OpenGIS City Geography Markup Language (CityGML) Encoding Standard, Version 1.0.0. Tech. Rep. Doc. No. 08-007r1, OGC, Wayland (MA), USA (2008)
9. Haerder, T., Reuter, A.: Principles of Transaction-Oriented Database Recovery. ACM Comput. Surv. 15, 287–317 (1983)
10. Hunt, B.: Mongoemf, http://github.com/BryanHunt/mongo-emf/wiki
11. Jézéquel, J.M., Barais, O., Fleurey, F.: Model Driven Language Engineering with Kermeta. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2011. LNCS, vol. 6491, pp. 201–221. Springer, Heidelberg (2011)
12. Khetrapal, A., Ganesh, V.: HBase and Hypertable for Large Scale Distributed Storage Systems A Performance evaluation for Open Source BigTable Implementations. Tech. rep., Purdue University (2008)
13. Lakshman, A., Malik, P.: Cassandra: Structured Storage System on a P2P Network. In: Proceedings of the 28th ACM Symposium on Principles of Distributed Computing, PODC 2009, p. 5. ACM, New York (2009)
14. Orend, K.: Analysis and Classification of NoSQL Databases and Evaluation of their Ability to Replace an Object-relational Persistence Layer. Master's thesis, Technische Universität München (2010)
15. Pagán, J.E., Cuadrado, J.S., Molina, J.G.: Morsa: A Scalable Approach for Persisting and Accessing Large Models. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 77–92. Springer, Heidelberg (2011)
16. Plugge, E., Hawkins, T., Membrey, P.: The Definitive Guide to MongoDB: The NoSQL Database for Cloud and Desktop Computing, 1st edn. Apress, Berkely (2010)
17. Scheidgen, M.: EMFFrag – Meta-Model-based Model Fragmentation and Persistence Framework (2012), http://code.google.com/p/emf-fragments
18. Scheidgen, M.: How Big Are Models – An Estimation. Tech. rep. (2012)
19. Scheidgen, M., Zubow, A., Sombrutzki, R.: ClickWatch – An Experimentation Framework for Communication Network Test-beds. In: IEEE Wireless Communications and Networking Conference, France (2012)

20. Schütt, T., Schintke, F., Reinefeld, A.: Scalaris: Reliable Transactional P2P Key/Value Store. In: Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG, ERLANG 2008, pp. 41–48. ACM, New York (2008)
21. Stadler, A.: Making interoperability persistent: A 3D geo database based on CityGML. In: Lee, J., Zlatanova, S. (eds.) Proceedings of the 3rd International Workshop on 3D Geo-Information, pp. 175–192. Springer, Seoul (2008)
22. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0, 2nd edn. Addison-Wesley Professional (2009)
23. Thomas, D.: Programming With Models – Modeling with Code. Journal of Object Technology 5(8) (2006)
24. Zubow, A., Sombrutzki, R.: A Low-cost MIMO Mesh Testbed based on 802.11n. In: IEEE Wireless Communications and Networking Conference, France (2012)

Formally Defining and Iterating Infinite Models [⋆]

Benoit Combemale[1], Xavier Thirioux[2], and Benoit Baudry[3]

[1] University of Rennes 1, IRISA, France
[2] INPT ENSEEIHT, IRIT, France
[3] Inria, Centre Rennes Bretagne Atlantique, France

Abstract. The wide adoption of MDE raises new situations where we need to manipulate very large models or even infinite model streams gathered at runtime (*e.g.,* monitoring). These new uses cases for MDE raise challenges that had been unforeseen by the time standard modeling framework were designed. This paper proposes a formal definition of an infinite model, as well as a formal framework to reason on queries over infinite models. This formal query definition aims at supporting the design and verification of operations that manipulate infinite models. First, we precisely identify the MOF parts which must be refined to support infinite structure. Then, we provide a formal coinductive definition dealing with unbounded and potentially infinite graph-based structure.

1 Introduction

The growing adoption of Model-Driven Engineering (MDE) at all steps of software development comes with new requirements for MDE theories and tools. In particular, this work focuses on the need to process (i) finite but very large models, and (ii) infinite models. A major challenge to process these categories of models consists in understanding the exact meaning of a query over a model for which the interpretation does not know the size at a given point in time.

To illustrate the need to process finite but very large models, let us consider the complete model representing the entire Eclipse platform (the minimal workbench with OSGi). This model includes about 5 million model elements. Current model processing tools require all the model elements in memory (*e.g.*, Eclipse Modeling Framework (EMF) [1]). With EMF, the model of the Eclipse platform Java code requires 900MB in RAM memory. Steel et al. [2] provide an even bigger example: when they adopted a MDE approach to analyze civil engineering models, they had to deal with more than 7.3 million computational objects. Programming languages provide a good source of inspiration to deal with these issues. Through the notion of lazy evaluation, programming languages allow (lazy) iterations on potentially infinite data-structures. Even in Java, which does not support laziness natively, skilled programmers tend to manually postpone object instantiations as much as possible, *i.e.* only when needed, in order to reduce instantaneous memory consumption. Recent work were inspired by this approach to propose lazy model transformations to process very large models [3], or NoSQL-based approaches for model persistence [4].

[⋆] This work is partially supported by the EU FP7-ICT-2009.1.4 Project N°256980, NESSoS: Network of Excellence on Engineering Secure Future Internet Software Services and Systems.

Beyond the problem of very large models, stands the issue of processing infinite models. This requirement becomes more and more critical with the growing adoption of the *models@runtime* paradigm [5]. For instance, if we consider a monitoring system which relies on a model at runtime to abstract information from a complex event processing (the CEP) engine. The CEP will indefinitely provide information about the environment and thus one cannot consider that the model at runtime will be bounded as it could grow indefinitely. Another illustration can be found in the realm of reactive systems, which are modeled by transition systems intended to run forever. In this case, the model at runtime that records the trace of states and events triggered during execution is another form of infinite model. To deal with infinite models, we could leverage mechanisms established in the area of web feeds, such as RSS syndication. In these cases, data following a predefined format is timely and infinitely appended to an initial model (even though the RSS file usually corresponds to a sliding window because older elements are removed). However, as far as we know, no model processing solution adopts the notion of sliding window over an infinite flow of model elements in order to deal with infinite models.

If we look at the current state of MDE theories and tools to deal with large and infinite models, we make two observations. First, metamodeling formalisms and most of the tools are deeply rooted on the assumption that models include a bounded number of model elements and that this bound is known when computing a query over the model. Second, there exist some ad-hoc implementations to deal with these issues, but there is no formal definition of infinite models and no reference formal semantics for a query over an infinite model. The consequence of these two observations is that it is currently impossible to formally verify operations that process very large or infinite models. This is a major challenge for the adoption of MDE in the use cases discussed above.

In this paper, we tackle the two limitations listed above through two major contributions.

- a detailed analysis of current metamodeling standards and a precise identification on how they prevent the definition of infinite models. In order to face this, we propose an extension to the MOF formalism to enable the local definition of an infinite part of a model.
- a coinductive semantics for a query operation over an infinite part of a model. This semantics relies on a formal definition of infinite models, and provides both a reference for various implementations and the foundations for the verification of operations that must process models of unknown size.

The paper is organized as follows. Section 2 illustrates through a concrete example how the current metamodeling formalisms such as MOF prevent the definition and manipulation of infinite models. Based on this observation, Section 3 introduces MOF extensions supported by a formal definition of infinite models, and Section 4 proposes a coinductive semantics supporting the manipulation of such infinite models. Since the proposed formal semantics is independent of any implementation choice, we discuss in Section 5 the various existing and possible implementations of coinductive operators. Finally, we conclude and outline our perspectives in Section 6.

2 Illustrative Example: How MOF Does Not Support Infinite Models

Model Driven Engineering (MDE) considers software artifacts as abstract typed graphs (i.e., models conforming to precisely defined metamodels). As discussed in the introduction, we have to deal with increasingly large models. In many cases these models may even be considered of unbounded and infinite size (i.e., their size is *a priori* unknown). Since models are conforming to metamodels, such situations must be considered in the definition of metamodels. These metamodels are themselves implemented using a meta-language, usually compliant with the *Meta Object Facility* (MOF) [6], such as Ecore [1].

This section illustrates how a meta-language such as MOF ties MDE practices to a vision of finite and bounded models and thus prevents the definition and manipulation of infinite models. We illustrate these issues with the UML2 State Machine formalism [7].

2.1 UML2 State Machine as an Illustrative Example

The state machine sketched in the bottom left corner of Figure 1 conforms to the State Machine metamodel displayed in the middle left of Figure 1 (see the metamodel level). This metamodel defines a *StateMachine* as composed of several *State* elements, including an initial state, as well as several *Event* elements which it may react to. States are pairwise linked through *Transition* elements as *source* and *target* states. Each transition is triggered by a set of events (*Trigger*) and in return sends events (*SendEvent*) as the *effect* of its firing.

The execution semantics of such a state machine processes as follows: The first *RuntimeEvent* to be processed (that is, in our case, an *InjectEvent* element) is popped from the *eventToProcess* queue belonging to the state machine. This element represents an *EventOccurrence* of some event. A *RuntimeEvent* is either locally raised by the state machine transitions (*endogenous*) or brought by the environment (*exogenous*) and several other preceding events, which have been previously processed, constitute the *cause* for which it occurs. Runtime events keep on being popped and put aside as unhandled, until this set contains enough events, for some *outgoing* transitions of the current state to be triggered. Then, actually firing the transition pushes some new events at the end of the event queue, changes the current state of the machine and removes the triggering events from the unhandled ones.

Since a model conforms to a precisely defined metamodel, the underlying model of a state machine (graph of objects) follows the constrained structure expressed in the metamodel, according to the MOF semantics.

2.2 Bounded Collections of Properties

As illustrated in Figure 1, the upper bound of the collection *eventToProcess* is relative to its cardinality. In MOF, this cardinality is reified in terms of the `lower` and `upper` attributes of the `Property` construct (cf. top of Fig. 1). According to the OMG

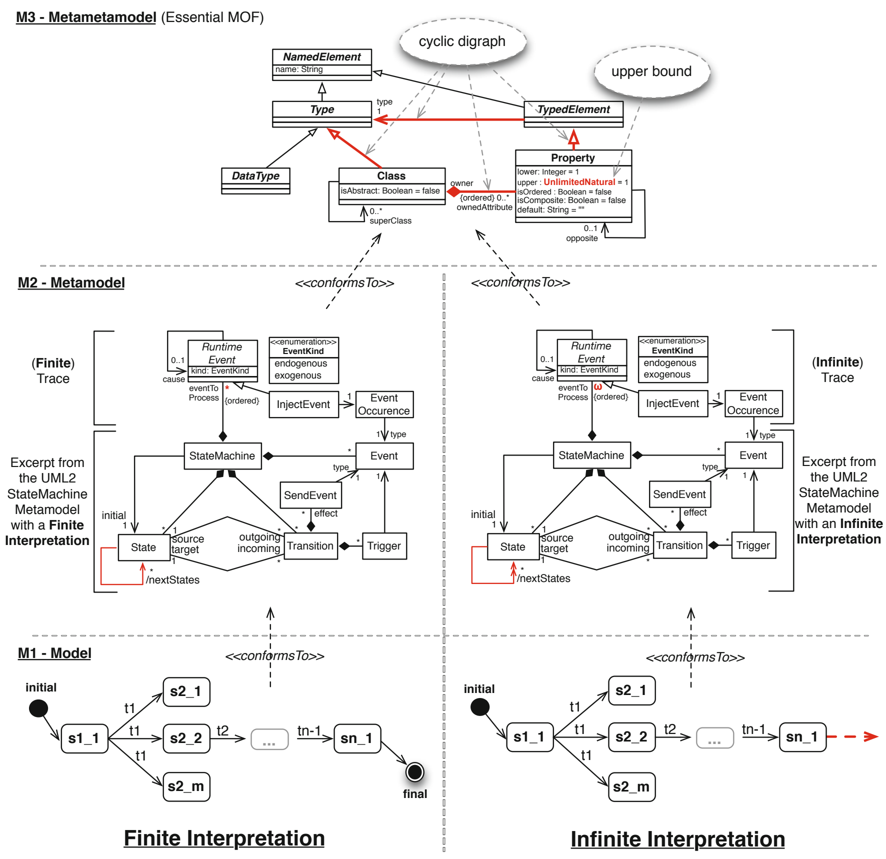


Fig. 1. Finite Model *vs.* Infinite Model

MOF Specification, the value of the `upper` attribute is typed by `UnlimitedNatural` and must be of kind `LiteralUnlimitedNatural` taken from the UML's Kernel [6, §12.4 and §14.4]. The UML's kernel leaves open the concrete semantics implementation but involves a notation for the unlimited value (`*`) which *"denotes unlimited (and not infinity)"* [7, §9.11.7]. In the same specification, *unlimited* is reasonably interpreted as bounded (i.e., finite) in the type *Collection* used for the resulting collections, for instance by navigating through relationships. Indeed *"the semantics of the collection operations is given in the form of a postcondition that uses the `IterateExp` of the `IteratorExp` construct."* [8, §11.6.1]. Its execution semantics, which refers to `IterateExpEval`, is explicitly bounded in the specification [8, §10.3.2.14]. Consequently, the execution semantics of the iterators (*e.g.,* the ones coming from OCL) on the collection *eventToProcess* is bounded. This means that the iterators assume that all elements are considered as available at any time of the iteration.

2.3 Transitive Closure

The underlying cyclic directed graph structure of any MOF-based metamodel (cf. top of Fig. 1) raises the issue of evaluating the transitive closure of a cycle. Such an issue can be shown in Figure 1, with the execution semantics of the closure over the next states (obtained from a given state with $outgoing \rightarrow collect(target)$). Since OCL 2.3, the standard includes a closure operation [8, §7.7.5]. This is very useful to specify recursive OCL operations. For instance, in Figure 1, the *reachable states* from a given state can be specified as proposed in the following OCL expression.

```
context State :: reachableStates () : Set( State ) body :
    { self }−>closure(outgoing−>collect( target ));
```

As stated by the OCL specification for the operator `closure`, *"the collection type of the result collection is the unique form (Set or OrderedSet) of the original source collection. If the source collection is ordered, the result is in depth first preorder."*. Here again, the underlying semantics refers to the type *Collection*. Consequently, the execution semantics of the closure is a finite processing, which assumes that the whole model is available for evaluation.

2.4 Discussion

Iterating both over the collection *eventToProcess* or the corresponding closure of the *reachable states* is thus a finite process for which the whole model is required (*e.g.,* a state machine defined at design time to model the behavior of a given class). The collection *eventToProcess* is bounded by the semantics of the attribute `upper` of a MOF property, and sets the "width" (i.e., number of outgoing edges from the same node) of the underlying graph of a conforming model. The *reachable states* process is finite, as defined by the underlying unfolding semantics as considered in the OCL operator `closure`, and sets the "depth" (i.e., length of a path with unique nodes) of the underlying graph of a conforming model.

These are strong limitations imposed by current metamodeling formalisms since, as illustrated in the introduction, such a state machine should also be considered as locally infinite (*e.g.,* state machine continuously updated to monitor at runtime a running and non-terminating program). In the following sections, we introduce slight modifications in MOF, which broaden its scope. In the context of infinite models, these modifications support the definition of a formal semantics for the MOF attribute `upper` and the unfolding semantics as used in the OCL operator `closure`.

3 Defining Infinite Models

This section starts with two proposals to extend the scope of MOF and allow the identification in metamodels of the infinite parts of the conforming models (i.e., parts which need to be manipulated despite their unknown size). These extensions are then used to provide a formal definition of infinite models.

3.1 Intuitive Presentation

While the semantics described by the OMG in the MOF specification involves a finite interpretation of models, some situations require an infinite interpretation of the same structure. For instance, a state machine can be considered as infinite (cf. right part of Fig. 1) if it abstracts the execution trace of a non-terminating program. In practice, this execution trace can be lazily built at design time while exploring the graph of reachable states, or continuously built at runtime during the system execution (*e.g.,* monitoring).

As seen in the previous section and illustrated in the example depicted in Figure 1, a model can be infinite in two situations, respectively in *the width* and in *the depth* of the underlying graph. In the following, we come back into these two situations and we propose MOF extensions with a concrete syntax to locally characterize in a metamodel the infinite parts of the conforming models.

- The collection defined in Figure 1 by the relation eventToProcess on StateMachine may be considered as infinite in case of a non terminating execution (*i.e.,* an infinite sequence of runtime event). We are noting ω the upper bound of the multiplicity of an infinite collection (cf. right part of Fig. 1), compared to the $*$ value which defines an unbounded but finite collection (cf. left part of Fig. 1). We assume that an infinite collection is ordered and then countable.
- A reflexive relation may be indefinitely unfolded and then, the computation of the closure may not terminate (currently, common practices consider that the closure computation terminates). This situation may even be mandatory if we consider in this relation a multiplicity with a lower bound greater than zero. We propose to graphically note such reflexive relations with infinite unfoldings as an arrow with two heads (cf. $\twoheadrightarrow$ in Fig. 1, right)[1]. We are aware that infinite unfolding may come from more complex cycles in a graph-based metamodel. For example in Figure 1, we may consider the cycle between State and Transition as an infinite unfolding. Common textual and graphical metamodeling notations cannot easily characterize a cycle. Nevertheless, a reflexive relation may be derived from an OCL expression characterizing the cycle. For example, the reflexive reference nextStates on State in Figure 1 is derived as specified by the following OCL expression. This derived reference characterizes the cycle between State and Transition.

```
context State :: nextStates : Set( State ) derive :
    self .outgoing−>collect( target );
```

According to this new notation, the classical example proposed in the left of Figure 1 is modified as shown in the right of Figure 1 in order to locally consider infinite structures (in our case to consider a possible infinite execution of a state machine). Note that several collections and cycles voluntarily keep the initial semantics based on a finite part of the model. For example, the collection of states (resp. transitions) which compose a state machine is unbounded but remains finite, and the transitive closure of

[1] Finite unfoldings where an explicit bound is known could be interesting. We could then add an annotation on relations, belonging to the same type as the upper bound of the multiplicity of collections. This extension is not taken into account in the scope of this paper.

the reflexive relation cause always terminate. So this syntax allows to clearly define in a metamodel the parts of a model which should be interpreted as infinite.

3.2 Formal Presentation

We propose a formal definition relying on the previous intuitive presentation of our extended metamodeling facilities.

In the following formal definition, we assume that we have finite sets of meta-elements (`MetaElements`) and relations (`Relations`), i.e., a finite metamodel. We are also using `Elements` as the set of possible model elements without any type information.

Definition 1 (Infinite Model). *Let $\mathscr{ME} \subseteq \texttt{MetaElements}$ be a bounded set of meta-elements. Let $\mathscr{R} \subseteq \{\langle me_1, r, me_2\rangle \mid me_1, me_2 \in \mathscr{ME}, r \in \texttt{Relations}\}$ be the bounded set of relations among meta-elements such that $\forall me_1 \in \mathscr{ME}, \forall r \in \texttt{Relations},$ $card\{me_2 \mid \langle me_1, r, me_2\rangle \in \mathscr{R}\} \leq 1$.*

We define an infinite model $\langle E, L\rangle \in \texttt{Model}(\mathscr{ME}, \mathscr{R})$ as a multigraph built over an unbounded set E of typed elements and an unbounded set L of labeled links such that:

$$E \subseteq \{\langle e, me\rangle \mid e \in \texttt{Elements}, me \in \mathscr{ME}\}$$
$$L \subseteq \left\{ \langle\langle e_1, me_1\rangle, r, \langle e_2, me_2\rangle\rangle \,\middle|\, \langle e_1, me_1\rangle, \langle e_2, me_2\rangle \in E, \langle me_1, r, me_2\rangle \in \mathscr{R} \right\}$$

We define the auxiliary type $Natural^{\omega} = \mathbb{N} \cup \{*, \omega\}$. $Natural^{\omega}$ is an extension of the `UnlimitedNatural` type provided by the OMG MOF specification. It is used to represent a range of possible numbers of instances. Unbounded finite ranges can be modeled using the $*$ value whereas unbounded infinite ranges can be modeled using the ω value. The type $Natural^{\omega}$ also comes equipped with the following order: $m < * < \omega$, for all $m \in \mathbb{N}$.

We aim at characterizing the presence of infinity both at the level of collections (i.e. the width of the underlying graph) and reflexive relations unfolding (i.e. the depth of the underlying graph).

First, regarding the width of the graph, we define the *upper* property which aims at distinguishing finite collections from infinite ones. Either for attributes or references (i.e., relation), a maximum number of instances of a target concept can be defined using the *upper* attribute, which value *n* is reflected in the following definition. Whether the upper bound of a relation is finite or infinite impacts the semantics (and implementation as well) of the model elements fetching operator *get* (cf. Section 4.2).

Definition 2 (Upper). *The upper property characterizes an upper bound n of a multiplicity of a given relation, this bound been taken from $Natural^{\omega}$.*

$$upper(\langle me_1, r, me_2\rangle \in \mathscr{R}, n \in Natural^{\omega}) \triangleq \langle E, L\rangle \mapsto$$
$$\forall \langle e, me\rangle \in E, me = me_1 \Rightarrow card(\{m_2 \in E \mid \langle\langle e, me_1\rangle, r, m_2\rangle \in L\}) \leq n$$

where the card function returns either $m \in \mathbb{N}$ or ω.

Second, regarding the depth of the graph, we introduce the property *ru_unstable* applying on primitive as well as derived relations. It requires the definitions of (maximal) model paths as the results of relation unfolding.

Definition 3 (Model Path). *Let $\langle E, L\rangle \in \texttt{Model}(\mathscr{ME}, \mathscr{R})$ be an infinite model. A model path is a relation path through model elements, i.e. a sequence of triples $\{\langle\langle e_i, me_i\rangle, r_i, \langle e_{i+1}, me_{i+1}\rangle\rangle\}_{i\in I} \in L$, where either $I = [0, sup[$ is finite ($sup \in \mathbb{N}$) or $I = \mathbb{N}$ is infinite. Relying on the previous model path definition, we also assimilate a model path to the (behavioral) trace of the model element creations.*

Definition 4 (Maximal Path). *Let $\langle E, L\rangle \in \texttt{Model}(\mathscr{ME}, \mathscr{R})$ be an infinite model. A maximal path is a model path, such that if it is finite, then the final element of the sequence has no relation to any element of the model.*

The property *ru_unstable* states that a given relation only gives rise to finite unfoldings, whatever the maximal model path considered. We need to focus on maximal paths as we express properties about possibly infinite unfoldings. Whether a given relation of a model satisfies the *ru_unstable* property or not impacts the semantics and implementation of the OCL *closure* and other iteration operators applied to this relation (cf. Section 4.3).

Definition 5 (Unstable Reflexive Unfolding). *Considering model paths as creation traces (cf. definition 3), a relation has only finite unfoldings if and only if it is unstable in any maximal model path π. This condition is rephrased as the following Linear Temporal Logic (LTL) property: $\Box\Diamond\neg r$ or equivalently $\neg\Diamond\Box r$. Relying on our model definition, it amounts to directly defining the following property:*

$$\begin{aligned} &ru\_unstable(\langle me, r, me\rangle \in \mathscr{R}) \triangleq \langle E, L\rangle \mapsto \forall \pi \in maximalpaths(\langle E, L\rangle) \\ &\quad I_\pi = \mathbb{N} \Rightarrow \exists i \in I_\pi, me_i \neq me \vee r_i \neq r \end{aligned}$$

where I_π is the set of indexes of π and satisfies definition 3.

4 A Coinductive Semantics to Iterate Infinite Models

We discuss in this section the ways to manage model elements in the context of an infinite model (Subsection 4.1) and we propose a formal definition of the operators needed to querying such models. First we formalize the common operators for getting model elements (Subsection 4.2), and then we rely on them to formalize an alternative of the main iterators inspired from the OCL language (Subsection 4.3). We finally put into practice the proposed operators, among others for the manipulation of infinite state machines as defined in the right of Figure 1 (Subsection 4.4).

4.1 Reasoning on Model Elements: From Finite to Infinite Model

Standard inductive semantics aims at defining finite data-structures as well as reasoning and programming with them. Therefore, an inductive structure comes naturally equipped first with an induction principle, allowing proofs by induction on the elements of this data-structure ; and second with a generic reduce programming primitive, allowing terminating recursive traversal of these data-structures. Induction principles are commonplace for reasoning about terminating algorithms and finite data-structures. It comes in many flavours such as induction on natural numbers, lists, binary trees, etc. As for the reduce and alike operators, they are also pervasive in programming paradigms, mostly in functional languages, but also in Java (e.g. the `Iterator` interface) and OCL (the `iterate` construction that operates on a finite collection).

Dually, coinductive semantics aims at defining potentially infinite data-structure. Therefore, a coinductive data-structure comes equipped with a coinduction principle and also a produce operator. The coinduction principle, among various usages, is at work when typing compilation units in languages supporting separate compilation. In Java for instance, you can type-check a bunch of classes, even if they are totally abstract and don't contain any piece of code. In this respect, checking type safety of two mutually dependent classes `A` and `B` works as if you were producing an infinite proof under the form: `A` is type-safe if `B` is type-safe if `A` is type safe, etc. There, type-safety is only proved to be a stable relation, with no base case at all. On the contrary, type-checking a concrete method amounts to reason inductively on the code structure, with assumed well-typed parameters and local object creations as base cases.

The produce operator (dual of the reduce operator) aims at producing potentially infinite data (as it cannot obviously perform a terminating recursive full traversal of an infinite structure), through the repeated execution of a piece of code that generates new values each time, appended to the growing structure. The resulting structure is the limit of this maybe infinite creation process, much akin to the fractal structures resulting from infinite iterations of a subdivision process. Often enough, as it is the case for instance in stream processing languages, the output structure is produced by a piece of code that consumes/explores in turn another corecursive input structure, one element at a time.

These coinductive concepts are mandatory in order to define the formal semantics of our MOF extensions (as used in the right part of Figure 1) independently of a particular implementation. The infinite state sequences of a state machine (cf. right lower part) should follow the semantics of the metamodel, just as plain finite models follow the inductive semantics of the metamodel. These state sequences may be defined as being produced from infinite sequences of transition-enabling events, following the execution semantics of the state machine. A standard inductive viewpoint on these sequences would be for instance to define an `allInstances()` OCL constraint that checks that each $n+1^{th}$ state is the result of executing a transition from a n^{th} state. As each OCL operator is supposed to work on a collection of states taken as a whole, evaluating the constraint on an infinite model would yield a non-terminating behavior and no outcome at all. Moreover, how such a collection may be produced still remains an open question in this case.

As can be seen, coinductive semantics, which amounts to producing infinite proofs and data, may be found in various areas, even though not always presented as such. Defining and formalizing a (coinductive) structure of models will help at elaborating important and practical tools for infinite model manipulation. A coinductive semantics may be implemented in several ways: in a programming language with lazy constructs that will evaluate only the finite browsed part of the model, or with data stream primitives randomly producing new model elements consumed afterwards by the execution semantics when possible, or else with a prefetch semantics that estimates how many model elements should be evaluated in advance, even if not needed. Each such implementation is interesting in its own right as it corresponds to a well-known class of applications. In the remainder of this section, we propose a formalization of a coinductive model semantics and the implementation standpoint is discussed in the next section.

4.2 Getting Model Elements

On a model $m = \langle E, L \rangle$, we first assume the operator getRoots() which corresponds to a minimal set of model elements from which any other model element can be accessed (i.e., a covering set). The accessibility predicate is defined as the existence of a finite model path from a model element to another. A set of model elements is defined as minimal when it is a covering set such that no proper subset is also covering.

$$\begin{array}{ll} accessibility(e, e' \in E) & \triangleq \exists \{\langle e_i, r_i, e_{i+1} \rangle\}_{i \in I} \in model\,path(\langle E, L \rangle), \\ & \quad e_0 = e \wedge \exists i \in I, e_i = e' \\ covering(S \subseteq E) & \triangleq \forall e' \in E, \exists e \in S, accessibility(e, e') \\ minimality(S \subseteq E) & \triangleq covering(S) \wedge \forall e, e' \in S, \neg accessibility(e, e') \end{array}$$

We can now specify the getRoots() operator which corresponds to the entry point facility on the model:

$$m.\mathsf{getRoots}() \in \{S \subseteq E \mid minimality(S)\}$$

We assume that the getRoots() operator return a finite set of roots. We note also that an alternative model definition based upon rooted multigraph may be adopted and would yield a non under-specified definition. In this case, the set of roots is uniquely defined.

Relying on the entry point previously defined by the getRoots() operator, we mainly consider the get() operator for getting model elements from a model $\langle E, L \rangle$. Usually, this operator allows to access to a property r from a model element e (written $e.r$ in OCL).

$$e.\mathsf{get}(r \in \texttt{Relations}) \triangleq \{e' \in E \mid \langle e, r, e' \rangle \in L\}$$

where $e \in E$. In our formalization, we assume that the return value of the get() operator is always a collection of model elements, tagged as either finite or infinite in our MOF extension and processed accordingly with the appropriate iterator. We are aware that in most model management APIs (e.g., EMF) or in the OCL query language, the return value may exhibit different types according to the known multiplicity of the relation.

4.3 Iterating Model Elements

We propose in this section a formalization of an alternative version – called coiterate – of the main generic OCL iterator iterate [8, § 7.6.5], from which all other iterators may be defined. Both iterators allow to browse a collection returned by the get() operator and to unfold a reflexive relation, for instance in order to compute its closure.

A particularity of OCL finite collections is that they are implicitly ordered. Indeed, the nth element may easily be retrieved with the help of the iterate operator. It may merely appear as a design clumsiness in the API for collections in the finite case. Yet, the ordering of elements is mandatory in the infinite case, as elements will be processed sequentially, one by one, and the user may observe intermediate results of this infinite computation. So we assume that collections, whether finite or infinite, are ordered.

The iterate operator processes the successive values of an (ordered) finite collection in order to compute the final value of an accumulator of any type. Dually to this semantics, the coiterate operator starts from an initial value which processing produces a potentially infinite collection of new values. This approach, which applies the coinduction principles, allows to produce a new collection either from an already existing infinite collection or more generally as the sequence of values built from successive assignments of a variable of any type.

In order not to depart too much from the iterate operator, from a syntactic viewpoint, we define coiterate as follows, first recalling the definition of iterate. We require that $coll$:Collection(A) possesses the three basic operations provided on Collection by OCL: isEmpty() which tests a collection for emptiness, first() which returns the first element of a collection, and append(elem:A) which appends a new element elem at the end of the collection. For the sake of readability, we also define the operation tail() which returns the collection without the first element[2]. Note that, as e_1, e_2 and e_3 are expressions and not values in the following definitions, we must use substitutions[3] in order to define recursively (resp. corecursively) these iterators.

$coll$->iterate(elem: A; acc : B = e_1 | acc = e_2) $\triangleq$
 $if\ coll$->isEmpty() $then\ e_1\ else\ let\ e_1' = e_2[coll$->first() | elem$][e_1$ | acc$]$ in
 $coll$->tail()->iterate(elem: A; acc : B = e_1' | acc = e_2)

$coll$->coiterate(acc : B = e_1 | acc = e_2; elem: A = e_3) $\triangleq$
 $if\ e_1 =$ null $then\ coll\ else\ let\ e_1' = e_2[e_1$ | acc$]$ in
 $coll$->append($e_3[e_1$ | acc$]$)->coiterate(acc : B = e_1' | acc = e_2; elem: A = e_3)

The coiterate operator starts from a finite collection $coll$, to which it will append a potentially infinite sequence of elements. For that purpose, it considers a variable acc, initialized with value e_1. From the current value of acc, if not equal to null, a new element $elem$ with value e_3 built from acc must be appended to the current resulting collection, and the next value of acc is given by e_2. This process is repeated as long as acc is not null. So the resulting collection is finite if and only if acc finally becomes null, otherwise the collection is infinite. Moreover, both iterators are able to handle several accumulating variables of any name at once, provided the variable $elem$ is defined.

4.4 Putting the Coiterate Iterator Into Practice

We illustrate how the coiterate iterator may be used, through the following two basic examples. In the first one, the infinite collection of the natural numbers is built and in the second one, the infinite collection of the squares of even numbers is built from the first collection.

[2] The operation tail() is defined in OCL by: $coll$->excluding(coll->first()).

[3] $e[u \mid u']$ is the expression e where any occurrence of sub-term u' has been replaced by u.

1. *Naturals* $\triangleq$

   ```
   Set{}->coiterate(acc:Integer = 0 |
        acc = acc+1; elem = acc)
   ```

2. *Squares* $\triangleq$

   ```
   Set{}->coiterate(acc:Collection(Integer) = Naturals |
        acc = acc->tail()->tail(); elem = acc->first()*acc->first())
   ```

Similarly, the operator coiterate may be used to browse the infinite collection of events to process (cf. eventToProcess in the right part of Fig. [1]) and then to specify the simulation of a state machine. For instance, the following listing define the body of an operation simulate() on StateMachine which creates the trace (*i.e.,* a sequence of states) according to the events to process[4].

```
context StateMachine::simulate() : Sequence(State) body :
    Set{}->coiterate(
        acc : Sequence(EventOccurence) = eventToProcess ;
        current : State = self.initial |
        current = self.step(current, acc->first()) ;
        acc = acc->tail() ;
        elem = current
    )
```

Our definition of the iterate (resp. coiterate) iterator was shown to browse a collection (resp. create a collection). However, it seems to be an elegant way to use these operators in order to also (co)iterate over unfoldings of a reflexive relation. In the following listing, the coclosure operator of a reflexive relation is defined in terms of coiterate and corresponds to a breadth-first traversal of the underlying graph. From this generic coclosure operator we also specify an operation on *State* computing the potentially infinite set of reachable states from a given state:

```
context T::coclosure( relation ) : Sequence(T) body :
    Set{}->coiterate(
            acc : Sequence(State) = Sequence{self} |
            acc  = acc->tail()->union(acc->first(). relation
                ->asSequence())
            elem = acc->first() ;
    )

context State :: reachableStates () : Sequence(State) derive :
    { self }->coclosure( nextStates );
```

[4] We consider the operation step() which returns the current state according to the event given as a parameter.

5 On the Implementation of Coinductive Operators

We have proposed in the previous section a formal specification to define and evaluate infinite models relying on coinduction principles. Stemming from this first milestone, our immediate next goal will be to explore pragmatical solutions to implement such principles.

In all cases, and by definition, it is not possible to store the entire infinite model in memory. When a causal dependency exists between the model producer and the model consumer, the model can be lazily interpreted (from the producer) in order to build, on demand (by the consumer), only the necessary model elements. In our example, the state machine may be lazily executed during a step-by-step simulation. Such lazily interpreted constructs have been recently proposed at the metamodel level in ATL [3]. In this case, infinite model processing is close to existing lazy interpretation (also called *call-by-need*) of a data structure [9] and relies on the following properties :

– the model elements in the infinite model are only built when it is necessary,
– once an infinite model element has been built, it is never built twice.

However, in some situations it may be necessary to relax these properties. For example, the last property implies *memoization*[5] and may be relaxed in some cases because it may render certain things impossible (e.g., because more elements may have to be remembered than system memory permits) or very inefficient (e.g., because of synchronization issues in concurrent systems). A non-memoized lazy evaluation also called *call-by-name* [10] may then be of interest. For example, such strategies have been explored in [4] as part of an approach for model persistence based on NoSQL databases.

Even if lazy interpretation seems promising, specifying the level of laziness is also important. In the context of the MDE, different solutions are possible. For instance, does model element creation entails creation of its references atomically? Or are these references also created on demand?

When the model consumption is disconnected from the model production, the order in which model elements arrive is out of control. Such an infinite model relies on the following properties (inherited from data stream [11]):

– model elements in the infinite model arrive online,
– once an element from an infinite model has been processed it is discarded or archived.

Different semantics may be considered depending on several options. For instance, model elements delivery must take into account the following options from [12]: *Pull vs. Push*, *Aperiodic vs Periodic* and *Unicast Vs. 1-to-N*. Then, mechanisms for model elements delivery may be inspired from known protocols in data stream, such as *Request/Response* (Aperiodic Pull), *polling* (Periodic Pull), *Publish / Subscribe* (Aperiodic Push) or *Broadcast Disks* (Periodic Push). Moreover, an info gatherer may be used to change the appearance of the real model elements delivery at the query language level. In this case, we have also to take into account the conformity points when model interpretation is coherent, for instance using an operation-based model representation [13].

[5] With memoization, computation of a model element is not repeated: the element is kept in a cache after its first usage.

6 Conclusion and Perspectives

The contributions of this paper are motivated by the need to define and manipulate infinite models (i.e. models whose comprehensive set of model elements is too large to be loaded or even not available). After pinpointing how current metamodeling formalisms prevent such situations, we first propose MOF extensions to locally characterize in a metamodel the infinite parts of the conforming models. Then we introduce a formal alternative semantics of the OCL operator iterate, called coiterate, providing an implementation-independent semantics for manipulating infinite models. The coiterate operator can be used both to browse an infinite collection, and to compute the infinite closure of a transitive relation (or more generally a cycle in a model).

Such a coiterate operator would support the formal verification of operations manipulating infinite models. This verification activity can be partially automated by proof assistant which supports coinductive semantics. Among others, Coq is a valuable candidate in that respect [14]. This paper also discusses various possible implementations of the coiterate operator, whether for reasons of partial availability or partial loading of models.

More generally, iterate and coiterate iterators may be used for model transformation (*e.g.,* QVT [15]). Indeed, the accumulator may be a model (i.e., *modeling in the large*) which is finitely extended and returned (iterate) or indefinitely browsed in order to produce a new model (coiterate). If we abstract away the pieces of code used as arguments of these iterators, it turns out that iterate needs a function of type[6] $1 + A \times B \rightarrow B$ whereas coiterate needs a function of type $B \rightarrow 1 + A \times B$. In order to apply these iterators to model production or browsing, the types A and B may be generalized to dependent types that denote arbitrary predicates over a (mega-)model structure, aiming at identifying patterns of interest and providing entry points in these patterns. In this context, we would need functions with respective types $\forall e \in E.1 + A(e) \times B(e) \rightarrow B(e)$ and $\forall e \in E.B(e) \rightarrow 1 + A(e) \times B(e)$, where e are elements of the (mega-)model. Here, $A(e)$ and $B(e)$ represent query or creation patterns, the functions induce transformation rules and the iterators represent the execution of a global transformation engine. Finally, specifying models and model transformations with type predicates allows to talk about their respective properties within a single language of types. Turning questions about models into typing problems also brings a rich amount of results in the scope, about type checking, type inference and subtyping issues for models and model transformations at once, as investigated in [16].

To conclude with it, some substantial amount of work will be necessary to draw all the consequences of this promising approach for model transformation and to thoroughly compare and cross-fertilize it with existing solutions.

References

1. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley (2008)

[6] In abstract set algebra settings, "1" denotes the set with only one element, "+" denotes the disjoint sum and "$\times$" the cartesian product.

2. Steel, J., Drogemuller, R., Toth, B.: Model interoperability in building information modelling. Software and Systems Modeling (SoSyM) 11, 99–109 (2012)
3. Tisi, M., Perez, S.M., Jouault, F., Cabot, J.: Lazy Execution of Model-to-Model Transformations. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 32–46. Springer, Heidelberg (2011)
4. Espinazo-Pagán, J., Cuadrado, J.S., Molina, J.G.: Morsa: A Scalable Approach for Persisting and Accessing Large Models. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 77–92. Springer, Heidelberg (2011)
5. Blair, G., Bencomo, N., France, R.B.: Models@ run.time. Computer 42, 22–27 (2009)
6. Object Management Group, Inc.: Meta Object Facility (MOF) 2.4.1 Core Specification. Final Adopted Specification (August 2011)
7. Object Management Group, Inc.: Unified Modeling Language (UML) 2.4.1 Infrastructure. Final Adopted Specification (August 2011)
8. Object Management Group, Inc.: Object Constraint Language (OCL) 2.3.1 Specification (January 2012)
9. Henderson, P., James, H., Morris, J.: A Lazy Evaluator. In: 3rd ACM Symposium on Principles on Programming Languages (POPL), pp. 95–103. ACM (1976)
10. Douence, R., Fradet, P.: A systematic study of functional language implementations. ACM Transactions on Programming Languages and Systems 20(2), 344–387 (1998)
11. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: 21st ACM Symposium on Principles of database systems (PODS), pp. 1–16 (2002)
12. Franklin, M., Zdonik, S.: A framework for scalable dissemination-based systems. SIGPLAN Not. 32(10), 94–105 (1997)
13. Blanc, X., Mounier, I., Mougenot, A., Mens, T.: Detecting model inconsistency through operation-based model construction. In: 30th International Conference on Software Engineering (ICSE), pp. 511–520. ACM (2008)
14. Bertot, Y.: Coinduction in coq. CoRR abs/cs/0603119 (2006)
15. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification, version 1.0 (April 2008)
16. Steel, J., Jézéquel, J.M.: On model typing. Software and Systems Modeling (SoSyM) 6(4), 401–414 (2007)

Query-Driven Soft Interconnection of EMF Models[⋆]

Ábel Hegedüs, Ákos Horváth, István Ráth, and Dániel Varró

Budapest University of Technology and Economics,
Department of Measurement and Information Systems,
1117 Budapest, Magyar tudósok krt. 2
{hegedusa,ahorvath,rath,varro}@mit.bme.hu

Abstract. Model repositories based on the Eclipse Modeling Framework (EMF) play a central role in the model-driven development of complex software-intensive systems by offering means to persist and manipulate models obtained from heterogeneous languages and tools. Complex EMF models can be assembled by interconnecting model fragments by hard links, i.e. regular references, where the target end points to external resources using storage-specific URIs. This approach, in certain application scenarios, may prove to be a too rigid and error prone way of interlinking models. As a flexible alternative, we propose to combine derived features of EMF models with advanced incremental model queries as means for soft interlinking of model elements residing in different model resources. These soft links can be calculated on-demand with graceful handling for temporarily unresolved references. In the background, the interlinks are maintained efficiently and flexibly by using incremental model queries as provided by the EMF-IncQuery framework.

1 Introduction

The Eclipse Modeling Framework (EMF) [1] serves as the underlying model management infrastructure for various industrial development tools, especially in the avionics and automotive domain. These domains necessitate the handling of large models with potentially millions of model elements. For maintainability and scalability reasons, such EMF models are not persisted in a single XMI document, but stored as an interconnected network of model fragments where each fragment stores a certain part of the entire system model. In other application scenarios, complete EMF models are used which are complemented with external traceability models to explicitly persist traceability links between requirements models, design models, analysis models or source code, for instance. In both scenarios, EMF models are frequently manipulated by several development or verification tools in complex toolchains operated by different design teams.

[⋆] This work was partially supported by the CERTIMOT (ERC_HU-09-01-2010-0003) project, the TÁMOP (4.2.2.B-10/1–2010-0009) grant and the János Bolyai Scholarship.

Unfortunately, the interconnection of complex EMF-based system models imposes several technical problems due to the identification strategies of model elements in the EMF infrastructure. When serializing a model, a model element is either identified by a unique identifier generated by EMF, or by a relative path of containment hierarchy in the given EMF resource. These techniques are used when interconnecting models using associations (EReferences) e.g. for internal traceability purposes: the target end of the association points to an object resided in a different model resource. Such interconnections are also used in external traceability scenarios where inter-model links are introduced from traceability metamodel elements to existing metamodels which cannot be altered.

These scenarios demonstrate various shortcomings of the core EMF technology. First, (1) interconnected EMF model fragments with circular dependencies including only regular references cannot be serialized. Furthermore, without truly intelligent multi-resource transaction management, (2) local changes in a model fragment may introduce broken links unless all dependent model fragments are manipulated together in working memory. Such broken links require tool-specific resolutions — with a worst case scenario of fixing the links manually by the designer using text editors (and not the modeling tool). Finally, (3) all traceability links captured by associations are explicitly persisted every time even if traceability links could be derived from existing unique identifiers.

In the paper, we provide an approach[1] for the soft interconnection of EMF models based on *derived features* and *incremental model queries*. Derived features are attributes and relations of the model calculated at runtime, and their values are often not stored explicitly. When using derived relations, the corresponding links only exist after the models are loaded. Therefore, model fragments can be (de)serialized in arbitrary order issuing warnings about broken links when certain resources are unavailable or not loaded. In order to provide an efficient and flexible handling of such soft links, we use the incremental model query framework EMF-IncQuery as a technical foundation. As a result, it is sufficient to identify a model element by a query instead of local or global identifiers, and less amount of information needs to be persisted for traceability purposes. Furthermore, the underlying model query technique provides excellent performance with little memory overhead [2] for managing inter-model links[2].

The rest of the paper is structured as follows. First, we illustrate interconnected EMF models in Section 2 on an industrial case study and propose derived features for managing soft interconnections. Then we propose model queries as specification means for derived features, and thus for soft links (Section 3). An incremental maintenance technique for soft links is described in Section 4. In Section 5 the application of soft interconnections is described for traceability modeling. Finally, related approaches and tools are described in Section 6 and Section 7 concludes our paper.

[1] Fully implemented and documented at `http://viatra.inf.mit.bme.hu/incquery/new/examples/query-driven-soft-links`

[2] The paper does not include performance specific contributions to EMF-IncQuery, more details are available at `http://viatra.inf.mit.bme.hu/performance`

2 Soft Interconnection of EMF Models: An Overview

2.1 Case Study: Modeling and Managing Business Processes

Our approach will be demonstrated on an business modeling case study inspired by a project carried out together with an industrial partner. While the actual metamodels (shown in Figure 1) are significantly simplified here due to space restrictions, they still demonstrate many practical industrial problems of interconnecting EMF models. In the case study, semi-automatic workflows (captured as a *process model*) contain both automated and manual tasks. Architectural-level deployment decisions are captured by a separate *system architecture model* comprising of jobs and data resources referring to tasks in the business process model. Finally, the instances of the processes managed by operators are captured in an *operation* model containing a checklist for each process with task entries assigned for each operator.

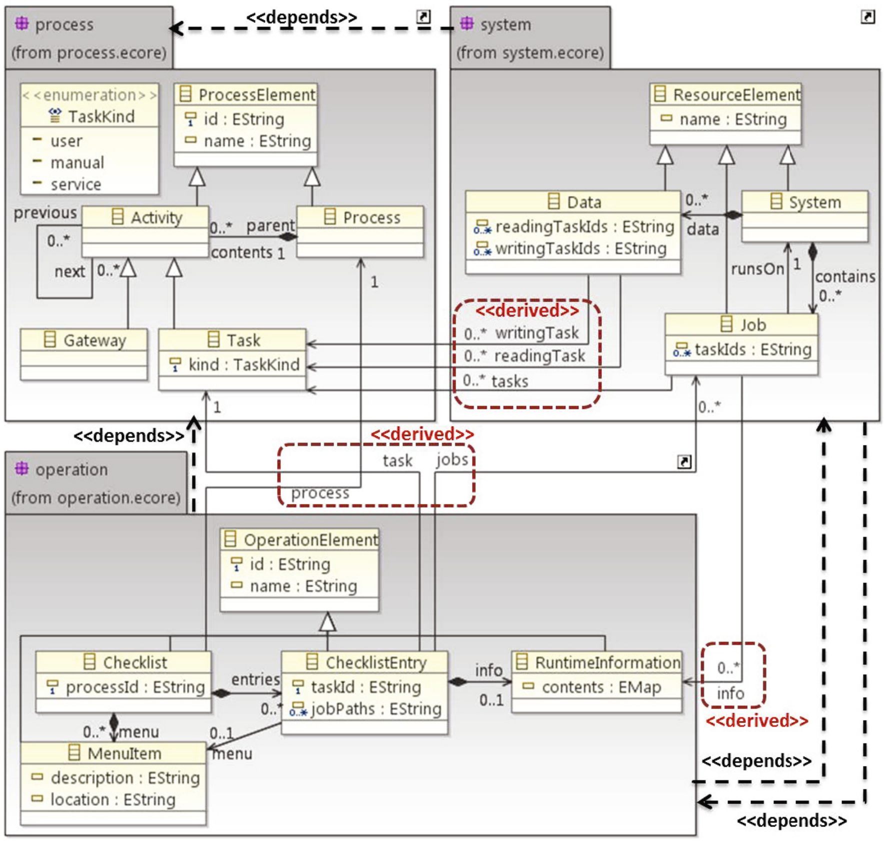


Fig. 1. The metamodels of the case study

Business process metamodel Business processes (process package) are defined by a fragment of the standard XPDL [3] metamodel. The ProcessElement top-level type defines id (unique identifiers) and name attributes for each element. A Process includes Activities that are either Tasks (atomic workflow steps) or Gateways (e.g. fork-join, decision, loops), while the control flow of the process is represented by the next and previous relations between activities. Based upon their kind attribute, tasks can be service (for automated execution through API calls), manual (where the operator initiates some job) and user (when the task itself is performed by an operator or other assigned personnel).

System architecture metamodel The system architecture metamodel (system package) defines a top-level ResourceElement that defines a name for each element. This simplified architecture includes Systems (representing larger components), Data elements that represent application data (e.g. configuration, input or output files) that can be read or written during the execution of tasks in the processes and Jobs (e.g. scripts or one-shot programs) that run on Systems. We assume that each system must have a unique name and each job contained in the same system must have different names. Otherwise, names are not globally unique in this domain.

Operation metamodel The operation metamodel (operation package) is used for representing Checklists followed by operators when performing the manual tasks in processes. The top-level OperationElement adds a name and a unique identifier for each model element. Each Checklist is related to a Process, and includes a number of entries and a menu. The menu contains MenuItems that have textual descriptions and a location, where the operator can access it. The entries are ChecklistEntry elements, each corresponding to one task, an arbitrary number of jobs, and optionally to a MenuItem. Finally, each entry can contain further information (e.g. historical statistics or requirements) stored by a RuntimeInformation element using a content map.

Inter-model connections These metamodels and thus the corresponding model instances heavily depend upon each other (see Figure 1). The following logical interconnections are present in our example: (1) a Job (from system) can be linked to a Task (in process); (2) a Process is a referenced from Checklist; (3) a ChecklistEntry links to both to a process Task and a system Job; (4) a RuntimeInformation (from operation) can be attached to a Job.

Many industrial tools (including the TIBCO Business Studio [4] used in our industrial project for capturing XPDL models and the AUTOSAR standard [5]) store identifiers of external (inter-model) elements using (a list of) simple string (or integer) attributes. In contrast, EMF uses EReferences (corresponding to lazily initialized inter-object pointers) to interconnect different models (or model fragments), which are resolved during the first traversal. For the current paper, they are referred to as *hard links*, as all such cross-model references are explicitly stored in a serialized model. In order to implement such standards over the EMF infrastructure, the main challenge is to provide a transparent reference maintenance mechanism that maps the textual identifiers to in-memory pointers and also allows their modification.

2.2 Soft Links for EMF Models by Query-Based Derived References

In the paper, we propose a soft linking technique for interconnecting EMF models by combining derived features and incremental model queries. The term "query-based soft links" refer to the fact that (1) certain model interconnections only exist at runtime but they are not maintained explicitly in instance models, but (2) the interconnected model elements can be accessed and navigated in a type-safe way along derived features. Furthermore, (3) our query based technique allows to define complex, n-ary interconnections of several model elements, and (4) to identify model elements dynamically based upon query results (instead of static unique IDs).

Derived features in EMF models represent computed information which can be calculated from other model elements. Essentially, we distinguish between *derived attributes*, which provide a data store for a(n instance of a) class and *derived references*, which represent “virtual” interconnections between model element instances (represented graphically by the derived stereotype in Figure 1). Derived features for soft links will be defined by using a declarative, high-level graph-based query language (Section 3) and evaluated truly incrementally (Section 4) as offered by the advanced model query framework EMF-IncQuery [6]. Our soft interconnection technique offers the following advantages:

– **Handling circular dependencies:** Circular dependencies between EMF models can be handled easily with soft links. For instance, metamodels system and operation are mutually dependent on each other along references jobs and info, which can materialize in a circular dependency on the model level preventing serialization using auto-generated regular EMF methods. As soft links are not serialized, this problem no longer occurs.
– **Graceful management of broken links.** When EMF models are manipulated by multiple tools, inter-model links can be easily broken, which result in runtime exceptions when the corresponding model element is attempted to be accessed along a broken link. Soft links provide graceful behavior in case of broken links by issuing warnings in case of unresolved elements.
– **Improved persistence.** Whenever a model interconnection can be calculated by a query, this does not necessarily have to be explicitly persisted into traceability models. As result, the load time of complex interconnected models can be reduced.
– **High performance.** Due to the incremental caching mechanism of EMF-IncQuery [2], derived features can be reevaluated very efficiently even in case of complex definitions (e.g. transitive closures [7]). As a result, the maintenance of soft links will be efficient with low memory overhead even for large models with complex traceability structures.

3 Definition of Soft Links as Model Queries

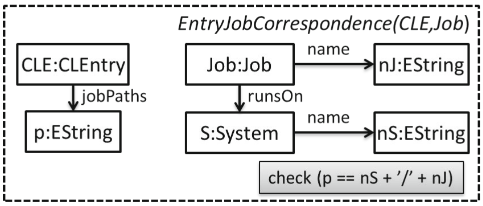
In order to support the runtime management of soft interconnections between models using derived features of EMF models, the graph pattern based model query language of EMF-IncQuery is used as the specification language for derived features. Therefore a brief introduction to this query language is provided first, followed by a detailed description on how this general purpose query language is adapted to specify the derived features for soft interconnections.

3.1 Model Queries by Graph Patterns: An Overview

Graph patterns [8] are an expressive formalism used for various purposes in model-driven development, such as defining declarative model transformation rules, capturing general-purpose model queries including model validation constraints, or defining the behavioral semantics of dynamic domain-specific languages. A graph pattern (GP) represents conditions (or constraints) that have to be fulfilled by a part of the instance model. A basic graph pattern consists of *structural constraints* prescribing the existence of nodes and edges of a given type, as well as *expressions* to define *attribute constraints*. A *negative application condition* (NAC) defines cases when the original pattern is *not* valid (even if all other constraints are met), in the form of a negative sub-pattern. A match of a graph pattern is a group of model elements that have the exact same configuration as the pattern, satisfying all the constraints (except for NACs, which must not be satisfied). The complete query language of the EMF-IncQuery framework is described in [9], while several examples will be given below.

3.2 Soft Links as Model Queries

Sample Soft Link First, we demonstrate on an example how the graph pattern EntryJobCorrespondence(CLE,Job) (Figure 2) can be used to express the soft links captured by the derived EReference jobs (connecting *ChecklistEntry* and *Job* in Figure 1), that is, to identify those jobs that correspond to a task execution as part of the checklist entry.



```
// ChecklistEntry.jobs link
pattern EntryJobCorrespondence
 (CLE, Job) = {
  Job.name(Job,JobName);
  System.name(System,SysName);
  Job.runsOn(Job,System);
  ChecklistEntry.jobPaths
   (CLE,JobPath);
  check(JobPath ==
   SysName+'/'+JobName);}
```

Fig. 2. Model query to define EntryJobCorrespondence in graphical and textual syntax

This model query formulated as a graph pattern has two parameters: *CLE* and *Job*, denoting the source and the target end of the soft link. The query defines the designated set of jobs by checking the names of the given job element *Job* and the system S it *runs on* (nJ and nS, respectively) and the path p stored in the entry. Model queries for the other soft links captured by derived features defined in the metamodel are defined similarly in Listing 1.1 and Listing 1.2.

```
// Job.tasks link
pattern JobTaskCorrespondence
 (Job,Task) =
{
  Task.id(Task,TaskId);
  Job.taskIds
   (Job,TaskId);
}
// Data.readingTasks link
pattern DataTaskReadCorrespondence
 (Data,Task) = {
  Task.id(Task,TaskId);
  Data.readingTaskIds
   (Data,TaskId);}
// Data.writingTasks link
pattern DataTaskWriteCorrespondence
 (Data,Task) = {
  Task.id(Task,TaskId);
  Data.writingTaskIds
   (Data,TaskId);}
```

Listing 1.1. Resource-Process mapping

```
// Job.info link
pattern JobInfoCorrespondence
 (Job,Info) = {
  ChecklistEntry.info(CLE,Info);
  RuntimeInformation.id
   (Info,InfoId);
  find EntryJobCorrespondence
   (CLE, Job);}
// ChecklistEntry.task link
pattern EntryTaskCorrespondence
 (CLE, Task) = {
  Task.id(Task, TaskId);
  ChecklistEntry.taskId
   (CLE,TaskId);}
// Checklist.process link
pattern ListProcessCorrespondence
 (Checklist, Process) = {
  Process.id(Process,ProcessId);
  Checklist.processId
   (Checklist,ProcessId);}
```

Listing 1.2. Checklist entry mapping

The query language also supports the following language constructs:

– check(JobPath == SysName + '/' + JobName) checks that the model element bound to variable *JobPath* is equal to the concatenated value of *SysName* and *JobName* (note that the evaluation will use String.equals to compare the value of EStrings).
– Using the find keyword, graph patterns are allowed to reuse other graph patterns. Therefore, if a soft link is defined as a model query by a corresponding graph pattern, this definition can be reused in other queries, and thus, in other soft links (along derived features).

The soft links defined as model queries using the graph pattern based language of EMF-IncQuery in the case study have two parameters, the first parameter denotes the source (i.e. the container EClass) while the second parameter denotes the target of the soft link. However, in the actual query language, this rule can also be satisfied by using *pattern annotations* for multi-parameter queries that explicitly specify which of the parameters is the context and which one will correspond to the target (or value). Furthermore, the adherence to this rule is checked at editing time by a built-in query language validator in the EMF-IncQuery tooling [6].

4 From Incremental Query Evaluation to Soft Links

In this section, we outline how the soft links can be managed using the efficient querying features of the EMF-IncQuery framework. Our approach can be integrated to notification based applications (like EMF) in a deep and transparent way by mapping model changes to the values of derived features using incremental evaluation.

4.1 Incremental Evaluation of Queries: an Overview

The key to efficient evaluation and change notification for derived features is the incremental graph pattern matching infrastructure of the EMF-IncQuery framework (first introduced in [10]), see the internal architecture in Figure 3.

The input for the incremental graph pattern matching process is the EMF instance model and its Notification API where callback functions can be registered to instance model elements that receive notification objects (e.g. ADD, REMOVE, SET etc.) when an elementary manipulation operation is carried out.

Based on a query specification, EMF-IncQuery constructs a RETE rule network [10] that processes the contents of the instance model to produce the query result at its output node. Query results are then post-processed by *auto-generated query components* to provide a type-safe access layer for easy integration into applications. This RETE network remains in operation as long as the query is needed: it continues to receive elementary change notifications and propagates them to produce *query result deltas* through its *delta monitor* facility, which are used to incrementally update the query result. These deltas can also be processed externally, which is a key feature for the integration of derived features (Section 4.2).

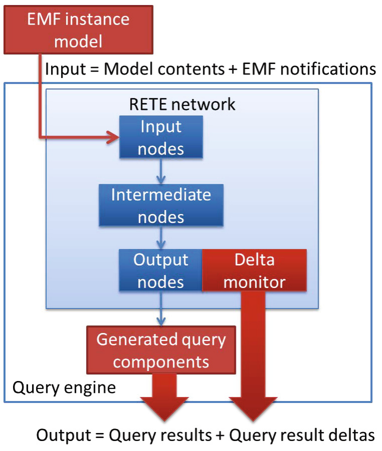

EMF instance model
Input = Model contents + EMF notifications
RETE network
Input nodes
Intermediate nodes
Output nodes
Delta monitor
Generated query components
Query engine
Output = Query results + Query result deltas

Fig. 3. The EMF-IncQuery architecture

By this approach, the query results (i.e. the match sets of graph patterns) are continuously maintained as an in-memory cache, and can be instantaneously retrieved. Even though this imposes a slight performance overhead on model manipulation, and a memory cost proportional to the cache size (approx. the size of match sets), EMF-IncQuery can evaluate very complex queries over large instance models very efficiently. These special performance characteristics, reported in [2], allow EMF-IncQuery-based derived features to be evaluated instantly in most cases, regardless of the complexity of the query or the size of the instance model.

4.2 Integration Architecture

To support soft links captured as derived features, the outputs of the EMF-IncQuery engine need to be integrated into the EMF model access layer at two points: (1) *query results* are provided in the getter functions of derived features, and (2) *query result deltas* are processed to generate EMF Notification objects that are passed through the standard EMF API so that application code can process them transparently. The overall architecture of our approach is shown in Figure 4.

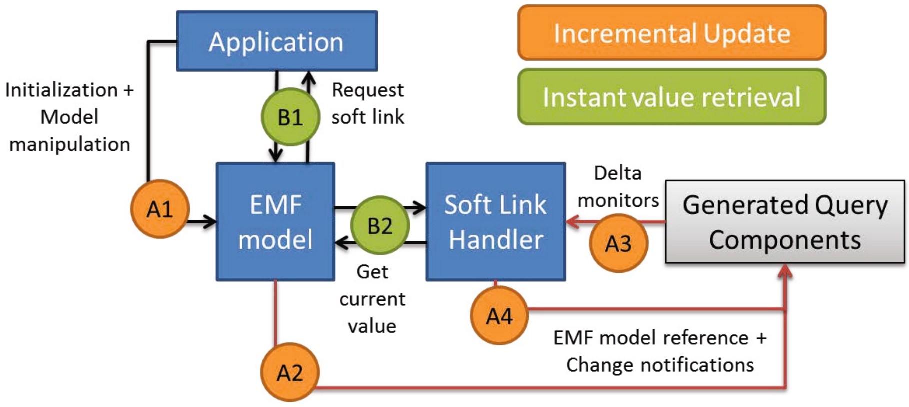


Fig. 4. Overview of the integration architecture, adopted from [11]

The application accesses both the model and the query results through the standard EMF model access layer – hence, no modification of application source code is necessary. In the background, as a novel feature, *soft link handlers* are attached to the EMF model objects that integrate the generated query components (pattern matchers). This approach follows the official EMF guidelines of implementing derived features and does not require more effort to integrate than ad-hoc Java code, or OCL expression evaluators. Note that these handlers can be used for managing regular derived features as well.

When an EMF application intends to read a soft link (B1), the current value is provided by the corresponding handler (B2) by simply retrieving the value from the cache of the related query. When the application modifies the EMF model (A1), this change is propagated to the generated query components of EMF-IncQuery along notifications (A2), which may update the delta monitors of the handlers (A3). Changes of soft links and derived features may in turn trigger further changes in the results sets of other derived features (A4).

Illustrative Example. Figure 5 illustrates a detailed elaboration EMF-IncQuery handlers, which process elementary model manipulation notifications to update, and generate notifications for derived features. The figure corresponds to a case where the user assigns a new Job to a ChecklistEntry through the Editor which is essentially a cle.getJobPaths().add(jobPath) method call on the Model. During the add method, the ChecklistEntry EObject sends an ADD notification to the Notification Manager, which will notify the EMF-IncQuery Query Engine about the model modification. The Query Engine updates the match sets of each query and registers the match events in the Deltamonitor. Once its finished with updating the RETE network, it invokes the callback method of each IncqueryFeatureHandler. Each handler has a Deltamonitor from which it retrieves the new and lost match events since the last callback to processes them. During the processing, the handler may send notifications of its own (e.g. the value set of the info soft link of job is updated) that is propagated to listeners. Anytime the soft link value is retrieved from the model (e.g. job.getInfo()), it accesses the handler for the current value of the derived feature, which is returned instantly.

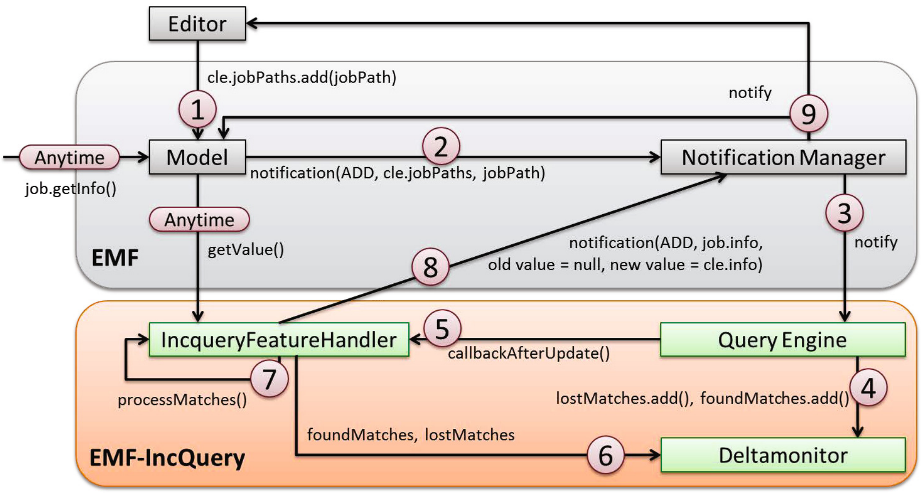

Editor
cle.jobPaths.add(jobPath)
notify
1
9
Anytime
Model
2
Notification Manager
job.getInfo()
notification(ADD, cle.jobPaths, jobPath)
Anytime
3
EMF
getValue()
notification(ADD, job.info, old value = null, new value = cle.info)
notify
8
IncqueryFeatureHandler
5
Query Engine
callbackAfterUpdate()
7
4
processMatches()
lostMatches.add(), foundMatches.add()
foundMatches, lostMatches
EMF-IncQuery
6
Deltamonitor

Fig. 5. Elaboration of the execution

Summary. In summary, the combined pattern matching and notification processing ensures that EMF-IncQuery-based soft links (and derived features) behave exactly as reeegular features of EMF instance models. This behavior ensures that user interfaces, model validators etc. can safely depend on soft interconnections built on soft links, without on-demand querying.

5 Applications in Traceability Modeling

The approach proposed in this paper can be interpreted in an external traceability modeling context. Figure 6 illustrates a typical architecture applied to

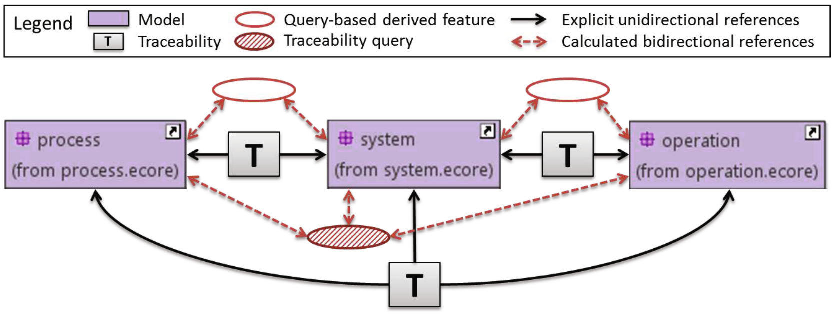


Fig. 6. External traceability modeling scenario

the examples of Section 2.1, where interconnections between three distinct models (belonging to the *process*, *system* and *operation* domains, respectively) are augmented with *explicit (external) traceability models* T.

In such a scenario, trace models T in EMF typically conform to a custom traceability metamodel that may describe simple binary (source-target) relationships with the help of association classes that use explicit unidirectional references to point to elements of the host models. In more complex cases, T may also include ternary (or hyper-) edges that interrelate multiple elements (e.g. three element types from all three domains, as in Figure 6).

5.1 Traceability-Specific Challenges

While this commonly used approach has an obvious advantage over *internal traceability/correspondence links* (as used in our previous examples), namely that the external models do not require the modification of the host metamodels, it also involves a number of frequently encountered problems as mentioned in Section 2.2:

- *Fragility*: Cross-resource hard EReferences are fragile, they may break when a host model is manipulated without the traceability model being loaded simultaneously. Additionally, in some scenarios, such as when using file-based EMF resources, traceability links may even break during external operations (e.g. when the files are moved within the workspace [12]).
- *Identification of target elements*: to work around the fragility issue, traceability modeling solutions may use IDs or fully qualified naming schemas (as presented in our previous examples) to store cross-references, even for external traceability models. However, such identifying attributes need to be present in the host models, and also necessitates an auxiliary mechanism that ensures consistency rules (such as uniqueness) within the host domains. If these prerequisites are not met, then additional, auxiliary techniques have to be used (such as ECore annotations, or genmodel modifications to add ID

maintenance capabilities to EMF domains, as used e.g. by EMFStore [13] and CDO [14]).

– *Persistence scalability issues*: in complex system modeling scenarios, the amount of EReferences can grow to be the dominant factor in storing the entire model space, in terms of both in-memory and serialized persistence overhead [2]. Hence, the performance of all model management-related operations (e.g. serialization) may be severely negatively affected as the size of the model resources grow, especially when taking the fragility issue into consideration (i.e. that traceability models with hard EReferences need to be loaded and manipulated together with host model fragments).

5.2 Traceability Management with Soft Links and Queries

The traceability architecture (components with black outline in Figure 6) can be augmented or even replaced with model-integrated *soft links* (symbolized by red outlined empty ovals) and *traceability queries* that can be accessed through the EMF-IncQuery API (oval with dashed fill). Both techniques share incremental, on-the-fly evaluation as their background.

Soft Links in a Traceability Context. From the traceability perspective, the most important advantages of soft links are that they are (logically) *bidirectional* references that are *maintained on-the-fly*. Thus, given that host metamodels are allowed to be augmented, such traceability links can be added without regard for circular serialization dependencies, that is, it is entirely up to the language designer to specify where such EReferences are going to reside, making trace link navigation also starting from host model elements feasible.

Additionally, as soft links provide graceful behavior for broken traceability references, erroneous trace records may be marked with warning markers, instead of throwing exceptions or runtime errors. These markers can then be corrected by e.g. a user-aided, on-demand resolution process, which may be further supported by helper queries that locate the most likely target host model element (esp. in the case when non-ID keys are used to identify model elements, such as EntryJobCorrespondence in Figure 2 – in this case, a helper query may enumerate those elements whose local names are similar).

As EMF-IncQuery query results can be represented by derived features as well as generic collections of EObjects, this feature may be used in a straightforward way to fine-tune which EReferences are going to be explicitly persisted and which ones are going to be calculated on-demand, when the models are loaded into memory. This gives the tool developer precise control over performance vs. compliance considerations (i.e. when certain traceability information is required to be stored persistently).

Finally, soft links behave exactly like normal EReferences (send notifications), easing the integration with user interface components or on-the-fly validators.

Using Traceability Queries for N-ary Links. If host metamodels cannot be modified, or hyperedges (multilinks, connecting three or more element types) are desired for traceability modeling, the architecture of Figure 6 can be augmented with generic queries. Such a case is illustrated by Figure 7. In this case, a ChecklistEntry is connected to Tasks and, consecutively, to Data elements to represent the traceability information between data elements that are read by a given check list element. Such a ternary relationship (with * multiplicities) may be implemented by the DataReadByChecklistEntry pattern (shown on the left in Figure 7).

```
pattern DataReadByChecklistEntry
 (CLE, Task, Data) = {
 find ChecklistEntryTaskCorrespondence
  (CLE,Task);
 find DataTaskReadCorrespondence
  (Data, Task);
}
```

```
DataReadByChecklistEntryMatcher matcher =
  DataReadByChecklistEntryMatcher.FACTORY
    .getMatcher(resourceSet);
int matchNum = matcher.countMatches(cle, null, null);
System.out.println("Found " + matchNum +
  " matches for entry " + cle.getName());
matcher.forEachMatch(cle, null, null,
  new IMatchProcessor<DataReadByChecklistEntryMatch>() {
    public void process(DataReadByChecklistEntryMatch match) {
      Task task = (Task) match.getTask();
      Data data = (Data) match.getData();
      System.out.println("Entry reads data " + data.getName()
        + " through task " + task.getName());
    };
  });
```

Fig. 7. Ternary links with traceability queries

This approach shares the functional benefits of soft links, with the one exception that it is not integrated into the EMF model layer and as such, it is not API-transparent to EMF-based tools. Instead, the query results can be accessed through an additional API provided by EMF-IncQuery (illustrated on the right in Figure 7). Here, the results of the DataReadByChecklistEntry pattern are processed using a generated DataReadByChecklistEntryMatch data transfer class and the IMatchProcessor<> visitor interface. Though not shown in Figure 7, the EMF-IncQuery API also exposes the *delta monitor* facility (Section 4) that allows to track the changes in the result of such a query.

Summary. Soft links and traceability queries can be used to overcome the challenges presented by traceability-specific applications by complementing external traceability models and supporting incrementally maintained bidirectional links between interconnected model elements.

6 Related Work

In this section we first give an overview of existing approaches and tools that deal with interconnection between models, then we briefly describe other model query techniques for EMF. Finally, we list approaches that rely on derived features and therefore may take advantage of our incremental evaluation techniques.

Interconnecting EMF Models. In [15] correspondences between models are handled by matching rules defined in the Epsilon Comparison Language, where the application conditions (called guards) use queries similarly to our approach. Additionally, Epsilon also manages model integrity between EMF models using the novel Concordance framework [12]. It is able to handle intermodel links when models are moved/renamed and helps in correcting invalid models caused by metamodel changes. Anwar [16] introduces a rule-driven approach for creating merged views of multiple separate UML models and relies on a correspondence metamodel and OCL expressions to support model merging and composition. VirtualEMF [17] allows the composition of multiple EMF models into a virtual model based on a composition metamodel, and provides both a model virtualization API and a linking API to manage these models. The approach is also able to add virtual links based on composition rules. In [18] an ATL-based method is presented for automatically synchronizing source and target models of a given transformation, based on the definition of the transformation.

Compared to them, the main distinctive features of our approach is (1) the fully incremental evaluation of queries for model interconnections, and (2) flexible support for query-based, computed soft links. It is a nice task for future research to combine the benefits of our current approach with the benefits of these existing solutions.

Model Query Approaches. OCL [19] is a standardized navigation-based query language, applicable over a range of modeling formalisms. Taking advantage of the expressive features and wide-spread adoption of OCL, the project Eclipse OCL through its *Essential OCL* language provides a powerful query interface that evaluates OCL expressions over EMF models. Additionally, it also supports the definition of invariants and operations to enrich the Ecore metamodel using either the *Complete OCL* [20] or the *OCLinEcore* [21] languages. Balsters [22] presents an approach for defining database views in UML models as derived classes using OCL. The derived classes in this case are the result set of queries, which is similar to the match sets provided by EMF-IncQuery.

There are several technologies for providing declarative model queries over EMF, e.g. EMF Model Query 2 [23] and EMF Search [24]. Other graph pattern based techniques like [25,26] have been successfully applied in an EMF context.

Cabot et al. [27] present an algorithm for incremental runtime validation of OCL constraints and uses promising optimizations, however, it works only on boolean constraints. An interesting model validator over UML models [28] incrementally re-evaluates constraint instances whenever they are affected, but relies on environments that support the recording of read-only access to the model, unlike EMF. Additionally, general-purpose model querying is not viable.

These approaches provide possible alternatives to implement model queries, thus, they can potentially be used for providing soft links. However, many of them lack incremental evaluation support or require significantly more integration effort to enable their use for soft links.

Application of Derived Features. The PROGRES language [29] allows the rule-based programming of graph rewriting systems and uses derived attributes for encoding dynamic semantics. ConceptBase.cc [30] is a database (DB) system for metamodeling and method engineering and defines active rules that react to events and can update the DB or call external routines, the latter could be applied in models as derived features representing data stored in the ConceptBase.cc DB. Neither tool has adopted EMF up to our best knowledge.

In [31] Diskin describes a formal framework for model synchronization that uses derived references for propagating changes between corresponding models. A recent work by Diskin et al. [32] proposes a theoretical background for model composition based on queries using Kleisli Categories, in their approach derived features are used for representing features merged from different metamodels. The conceptual basis is similar to our approach in using query-based derived features, however, it offers algebraic specification, while our approach might serve as an implementation for this generic theoretical framework.

The MOF 2.0 tool in [33] allows the definition of derived features using OCL. It handles derived attributes and operations as custom code provided by the user and redirects calls using reflection. The FUJABA [34] tool suite also supports derived edges by path expressions. Both tools work in a non-incremental way.

JastEMF [35] is a semantics-integrated metamodeling approach for EMF. It uses derived features as side-effect free operations (i.e. queries) and refers to them as the static semantics of the model. Therefore, our query-based approach could be integrated with JastEMF without any problems.

In a previous paper [11], we offer an algorithm for incremental evaluation of derived features and present technical details on the integration of existing native implementations. The current paper provides details on applying incremental queries for soft interconnections by using derived features in EMF.

7 Conclusion

Interconnections between model fragments of complex EMF models are usually represented as regular associations and persisted using storage-specific URIs. This approach proves to be rigid and error-prone in some application scenarios.

We proposed to use derived features as a flexible alternative to provide soft interlinking between model fragments, and demonstrated an approach for incremental evaluation of soft links with the use of model queries on an industrial case study. Our approach supports circular dependency between models, graceful handling for unresolved links and is implemented using EMF-IncQuery, which provides efficient evaluation capabilities for incremental model queries.

As a primary direction for future work, we plan to integrate traceability queries into the EMF model layer by constructing *derived classes* whose instances behave like EObjects but their lifecycles are managed by an underlying incremental query. Such constructs could be used to create n-ary traceability models that are automatically kept in-sync, retaining the graceful handling of soft links.

References

1. The Eclipse Project: Eclipse Modeling Framework, http://www.eclipse.org/emf
2. Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A.: Incremental Evaluation of Model Queries over EMF Models. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 76–90. Springer, Heidelberg (2010)
3. Workflow Management Coalition: XML Process Definition Language, v2.1. (2008), http://www.wfmc.org/xpdl.html
4. TIBCO Developer Network: TIBCO Business Studio (2012), http://developer.tibco.com/business_studio
5. AUTOSAR Consortium: The AUTOSAR Standard, http://www.autosar.org
6. Bergmann, G., Hegedüs, Á., Horváth, Á., Ráth, I., Ujhelyi, Z., Varró, D.: Integrating Efficient Model Queries in State-of-the-Art EMF Tools. In: Furia, C.A., Nanz, S. (eds.) TOOLS 2012. LNCS, vol. 7304, pp. 1–8. Springer, Heidelberg (2012), https://www.inf.mit.bme.hu/en/research/publications/integrating-efficient-model-queries-state-art-emf-tools
7. Bergmann, G., Ráth, I., Szabó, T., Torrini, P., Varró, D.: Incremental pattern matching for the efficient computation of transitive closures. In: Sixth International Conference on Graph Transformation, Bremen, Germany (submitted, 2012)
8. Varró, D., Balogh, A.: The Model Transformation Language of the VIATRA2 Framework. Science of Computer Programming 68(3), 214–234 (2007)
9. Bergmann, G., Ujhelyi, Z., Ráth, I., Varró, D.: A Graph Query Language for EMF Models. In: Cabot, J., Visser, E. (eds.) ICMT 2011. LNCS, vol. 6707, pp. 167–182. Springer, Heidelberg (2011)
10. Ráth, I., Bergmann, G., Ökrös, A., Varró, D.: Live Model Transformations Driven by Incremental Pattern Matching. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 107–121. Springer, Heidelberg (2008)
11. Ráth, I., Hegedüs, Á., Varró, D.: Derived Features for EMF by Integrating Advanced Model Queries. In: Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., Kolovos, D. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 102–117. Springer, Heidelberg (2012), https://viatra.inf.mit.bme.hu/sites/viatra.inf.mit.bme.hu/files/attachments/ecmfa2012.pdf
12. Rose, L., Kolovos, D., Drivalos, N., Williams, J., Paige, R., Polack, F., Fernandes, K.: Concordance: A Framework for Managing Model Integrity. In: Kühne, T., Selic, B., Gervais, M.-P., Terrier, F. (eds.) ECMFA 2010. LNCS, vol. 6138, pp. 245–260. Springer, Heidelberg (2010)
13. The Eclipse Project: EMFStore (2012), http://www.eclipse.org/emfstore
14. The Eclipse Project: The CDO Model Repository (2012), http://www.eclipse.org/cdo
15. Kolovos, D.S.: Establishing Correspondences between Models with the Epsilon Comparison Language. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 146–157. Springer, Heidelberg (2009)
16. Anwar, A., Ebersold, S., Coulette, B., Nassar, M., Kriouile, A.: A rule-driven approach for composing viewpoint-oriented models. Journal of Object Technology 9(2), 89–114 (2010)
17. Clasen, C., Jouault, F., Cabot, J.: Virtual Composition of EMF Models. In: 7èmes Journées sur l'Ingénierie Dirigée par les Modèles (IDM 2011), Lille, France (2011)

18. Xiong, Y., Liu, D., Hu, Z., Zhao, H., Takeichi, M., Mei, H.: Towards automatic model synchronization from model transformations. In: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering, ASE 2007, pp. 164–173. ACM, New York (2007)
19. The Object Management Group: Object Constraint Language, v2.0 (May 2006), http://www.omg.org/spec/OCL/2.0
20. Willink, E.D.: Aligning ocl with uml. ECEASST 44 (2011)
21. Eclipsepedia: MDT/OCLinEcore (2012), http://wiki.eclipse.org/MDT/OCLinEcorel
22. Balsters, H.: Modelling Database Views with Derived Classes in the UML/OCL-framework. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 295–309. Springer, Heidelberg (2003)
23. The Eclipse Project: EMF Model Query 2, http://wiki.eclipse.org/EMF/Query2
24. The Eclipse Project: EMFT Search, http://www.eclipse.org/modeling/emft/?project=search
25. Biermann, E., Ermel, C., Taentzer, G.: Precise Semantics of EMF Model Transformations by Graph Transformation. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 53–67. Springer, Heidelberg (2008)
26. Giese, H., Hildebrandt, S., Seibel, A.: Improved flexibility and scalability by interpreting story diagrams. In: Proceedings of GT-VMT 2009, vol. 18. ECEASST (2009)
27. Cabot, J., Teniente, E.: Incremental integrity checking of UML/OCL conceptual schemas. J. Syst. Softw. 82(9), 1459–1478 (2009)
28. Groher, I., Reder, A., Egyed, A.: Incremental Consistency Checking of Dynamic Constraints. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 203–217. Springer, Heidelberg (2010)
29. Schürr, A.: Introduction to PROGRESS, an Attribute Graph Grammar Based Specification Language. In: Nagl, M. (ed.) WG 1989. LNCS, vol. 411, pp. 151–165. Springer, Heidelberg (1990)
30. Jeusfeld, M.A., Jarke, M., Mylopoulos, J.: Metamodeling for Method Engineering. The MIT Press (2009)
31. Diskin, Z.: Model Synchronization: Mappings, Tiles, and Categories. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2011. LNCS, vol. 6491, pp. 92–165. Springer, Heidelberg (2011)
32. Diskin, Z., Maibaum, T., Czarnecki, K.: Intermodeling, Queries, and Kleisli Categories. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 163–177. Springer, Heidelberg (2012)
33. Scheidgen, M.: On implementing mof 2.0—new features for modelling language abstractions (2005)
34. Nickel, U., Niere, J., Zündorf, A.: The FUJABA environment. In: Proc. ICSE 2000, pp. 742–745 (2000)
35. Bürger, C., Karol, S., Wende, C., Aßmann, U.: Reference Attribute Grammars for Metamodel Semantics. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 22–41. Springer, Heidelberg (2011)

Modeling the Linguistic Architecture of Software Products

Jean-Marie Favre[1], Ralf Lämmel[2], and Andrei Varanovich[2]

[1] Université Joseph Fourier, Grenoble, France
[2] Software Languages Team, Universität Koblenz-Landau, Germany

Abstract. Understanding modern software products is challenging along several dimensions. In the past, much attention has been focused on the logical and physical architecture of the products in terms of the relevant components, features, files, and tools. In contrast, in this paper, we focus on the linguistic architecture of software products in terms of the involved software languages and related technologies, and technological spaces with linguistic relationships such as membership, subset, or conformance. We develop a designated form of megamodeling with corresponding language and tool support. An important capability of the megamodeling approach is that entities and relationships of the megamodel are linked to illustrative software artifacts. This is particularly important during the understanding process for validation purposes. We demonstrate such megamodeling for a technology for Object/XML mapping. This work contributes to the *101companies* community project.

Keywords: Megamodel, Linguistic architecture, Software language, Software technology, Technological space, Object/XML mapping, MegaL.

1 Introduction

Understanding modern software products is challenging because they are complex along several dimensions. We are specifically interested in the complexity of software products due to the involved *software languages*, *software technologies*, and *technological spaces* [7, 17] while possibly leveraging generative, reflective, transformational, and model-driven engineering approaches.

Consider, for example, web applications. A given application may leverage several software languages simultaneously: programming languages (e.g., Java, PHP, JavaScript, or Python), reusable domain-specific languages (e.g., CSS, XSLT, or SQL), library-based languages (e.g., JQuery, DOM), implicit languages relying on particular frameworks or annotation schemas (e.g., particular configuration or mapping languages for frameworks such as Hibernate or JAXB) as well as problem-specific languages (such as the object models, schemas, or domain-specific languages specifically designed for the application). Artifacts of these languages are processed, generated, or affected by compilers, interpreters, code generators, annotation injectors, etc.

In order to facilitate the understanding of current software languages and software technologies, the *101companies* community project [11][1] aims at developing a free, structured, web-accessible knowledge resource including an open-source repository for different stakeholders with interests in software technologies, software languages, and technological spaces; notably: teachers and learners in software engineering or software languages as well as software developers, software technologists, and ontologists.

In the context of the *101companies* project, the present paper[2], introduces a *megamodeling* approach supported by a language MegaL and associated tools for editing and exploration of megamodels. According to [1]: "A megamodel is a model of which at least some elements represent and/or refer to models or metamodels." Different forms of megamodeling have been introduced and utilized elsewhere [2,3,14,21,26,27]. Existing work essentially focuses on modeling typical Model Driven Engineering entities such as models, metamodels and transformations. Instead, we focus on conceptual entities such as (software) languages and (software) technologies as well as a range of so-called digital entities including languages, language processors, programs, libraries, files, directories, source files, meanings of programs, and in-memory structures such as object graphs. For instance, only by including languages (as opposed to their description by metamodels or grammars), we are able to explain certain linguistically founded aspects of software technologies.

We also say that the developed megamodeling approach addresses the 'linguistic architecture' of software products, thereby complementing other, more established dimensions of architecture: the 'logical architecture' as it is the subject of 'classical' software architecture as well as specific paradigms such as component-, feature- or aspect-oriented software development; the 'physical architecture' which is typically concerned with building, packaging, and deploying software and hence, with entities such as files and servers.

Contributions of the Paper

- ◇ We develop a megamodeling approach that is useful for understanding the linguistic architecture of software products in terms of the involved languages, technologies, and linguistic relationships. This approach is supported by the MegaL language and an associated tool suite under development.
- ◇ We demonstrate megamodeling in the challenging context of Object/Relational/XML [19,22,29] mapping (or O/R/X mapping). In the paper, we focus on one *O/X mapping* technology with a megamodel that is highly abstract but still it includes the key characteristics of the technology in question.
- ◇ The value of our megamodels is a cognitive one: they facilitate understanding of technologies and usage thereof. We strongly improve such cognitive value by enabling a form of *linked megamodels* such that entities and

[1] `http://101companies.org`

[2] The paper's website `http://softlang.uni-koblenz.de/mega/` provides supplementary material including some megamodels of software products and technologies.

relationships are linked to resources (e.g., in the *101companies* repository) so that megamodels can be explored and validated.

Non-contributions of the Paper. Note that the goal of this paper is by no means to define the ultimate megamodeling language in this context (such as MegaL) in any exhaustive or formal way, but just on the contrary, to motivate the overall approach and to illustrate its value with a concrete example. Also, the megamodeling approach, as it stands, does not yet readily support any sort of automated analysis or verification of software products. Instead, the current focus is on enabling the description of entities and relationships in a linguistic architecture in a way that is fit for validation purposes during the human understanding process.

Road-map of the Paper. §2 motivates and illustrates the notion of linguistic architecture. §3 describes entity and relationship types needed for modeling linguistic architecture. §4 develops an initial megamodel for O/X mapping that abstracts essentially from most aspects of the concrete O/X mapping technology and the concrete software that uses O/X mapping. §5 derives a more detailed megamodel that also captures interesting .NET-specific characteristics of O/X mapping. §6 develops the notion of linked megamodels. §7 discusses related work. §8 concludes the paper.

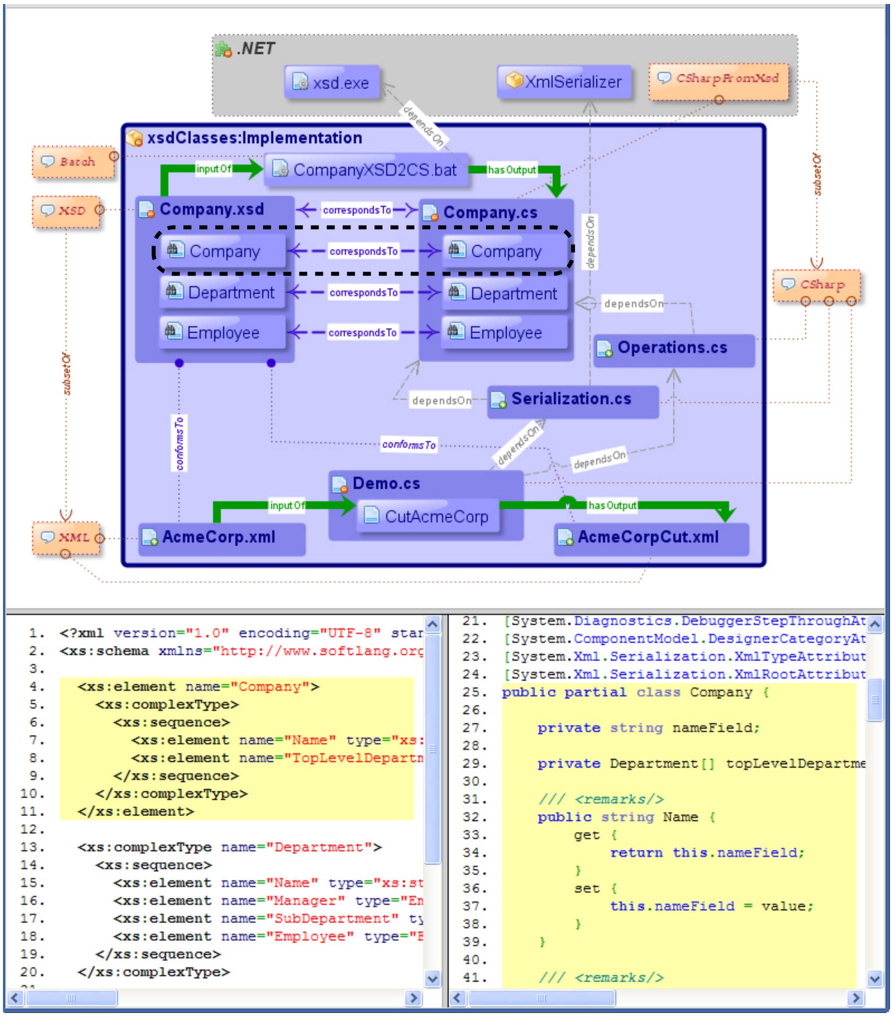
2 Illustration of Linguistic Architecture

Consider the upper frame in Figure 1. (The lower frame will be discussed in §6.) The linguistic architecture of a software product is described in the MegaL/yEd visual notation.[3] The product is a *C#*-based application which makes use of .NET's Object/XML mapping technology.[4] In fact, the product is a *101companies* implementation, which is named *xsdClasses* and available online. Hence, the application deals with companies (as in the human resources domain) including operations for totaling and cutting salaries (symbolized by the model element *Operations.cs*) as well as XML-related functionality for de-/serialization (see *Serialization.cs*). There are model elements for XML types according to the XSD language for XML schemas (see file *Company.xsd*) and *C#* classes (see file *Company.cs*) with fragments (see *Company*, *Department*, and *Employee*). There are correspondence relationships between the XML and object types to express that instances of these types can be (roughly) converted into each other (modulo the X/O impedance mismatch [18]). Class generation is automated with a batch file

[3] MegaL is currently a combination of an ontology and a set of concrete syntaxes; there exist these flavors of the language: MegaL/yEd—a visual notation, MegaL/TXT—a textual notation and MegaL/RDF—an RDF version of MegaL. The correspondance between these notations is rather straightforward and it will be introduced by means of illustration in the course of the paper.

[4] http://msdn.microsoft.com/en-us/library/x6c1kb0s(v=vs.71).aspx

The upper frame uses the MegaL/yEd visual notation for megamodeling.
The lower frame shows *linked artifacts* of the product explained later in the paper.



```
<?xml version="1.0" encoding="UTF-8" star
<xs:schema xmlns="http://www.softlang.org

  <xs:element name="Company">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Name" type="xs:
        <xs:element name="TopLevelDepartm
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:complexType name="Department">
    <xs:sequence>
      <xs:element name="Name" type="xs:st
      <xs:element name="Manager" type="Em
      <xs:element name="SubDepartment" ty
      <xs:element name="Employee" type="E
    </xs:sequence>
  </xs:complexType>
```

```
[System.Diagnostics.DebuggerStepThroughAt
[System.ComponentModel.DesignerCategoryAt
[System.Xml.Serialization.XmlTypeAttribut
[System.Xml.Serialization.XmlRootAttribut
public partial class Company {

    private string nameField;

    private Department[] topLevelDepartme

    /// <remarks/>
    public string Name {
        get {
            return this.nameField;
        }
        set {
            this.nameField = value;
        }
    }

    /// <remarks/>
```

Fig. 1. The linguistic architecture of a software product when displayed with the MegaL/Explorer tool

(see *CompanyXSD2CS.bat*), which essentially invokes the .NET tool `xsd.exe` (see *dependsOn*). Ultimately, the operation for cutting companies is invoked by demo functionality (see *Demo.cs*) and applied to a specific company—the *Acme Corporation*.[5]

[5] http://en.wikipedia.org/wiki/Acme_Corporation

The shown linguistic architecture describes artifacts as they arise during development time and runtime together with the relationships regarding dataflow, language membership, schema/type conformance, and correspondence. Characteristics of the .NET technology for Object/XML mapping are clearly identifiable. Consider, for example, the fact that the class generator is not described as generating 'arbitrary' *C#*. Instead, the subset *CSharpFromXsd* is introduced for referring to regular *C#* as produced by the generator. The identification of such 'hidden' languages is fundamental to the understanding of software technologies.

3 Entity and Relationship Types for Megamodels

The proposed form of megamodeling essentially involves the identification and classification of entities and relationships that make up the linguistic architecture of software products or the underlying software technologies. In this section, we gather a set of entity and relationship *types* that may be used in megamodels.

3.1 Background

There exist megamodel-like models in different areas of computer science. Linguistic relations have been of interest since the early days of computing as *tombstone diagrams* testify. In Figure 2, on the left, we show a tombstone diagram, as it is used in compiler construction to describe the bootstrapping process for a *C* compiler, also written in the programming language *C* and compiling to *M* (the machine language) such that initially another *C* compiler is needed—this time written (or executable) in *M*. Hence, *languages* and *compilers* serve as entities while relationships are concerned with *dataflow* or *function application* and *membership*.

On the right, we show a much more recent diagram, as it appears in the documentation of the *ATL* transformation language; the diagram shows the mechanics of a model transformation in terms of entities for the involved *models* and *metamodels* as well as relationships for *conformance* and *dataflow*.

Further inspiration, specifically regarding linguistically relevant relationships, can be drawn from fundamental research on modeling and model management. The 'conformsTo' relationship is established in modeling for relating models and metamodels [9,16]. We also rely on yet other basic modeling relationships (as in UML)—in particular 'partOf' and 'dependsOn'. The 'elementOf' and 'subsetOf' relationships are hardly used directly in regular modeling, but it appears in fundamental discussions, when *the usage of languages* is taken into account as opposed to sole restriction to metamodel-based conformance [9,16]. The 'modelOf' or 'representationOf' relationship [16,23,24] is important for capturing the roles of descriptions, definitions, specifications, programs, or more generally

[6] Source: http://en.wikipedia.org/wiki/Tombstone_diagram
[7] Source: http://wiki.eclipse.org/ATL/Concepts#Model_Transformation

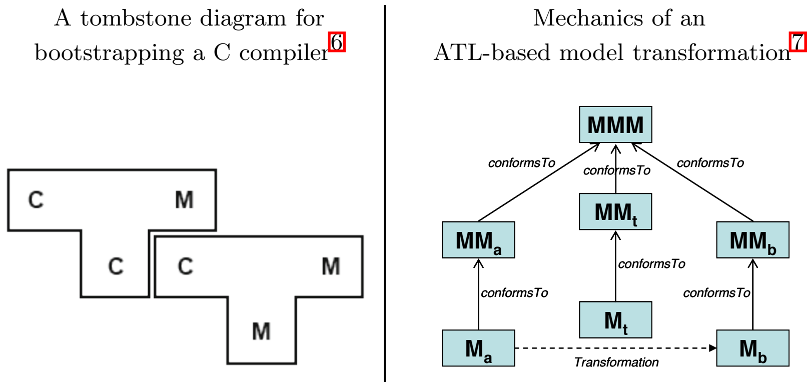


Fig. 2. Megamodels in different areas of computer science

models in megamodels. Ideally bidirectional intermodel mappings [5,6], with interpretations at both the schema (metamodel) and the instance (model) level, give rise to the 'correspondsTo' relationship in our terminology.

Based on this background, the MegaL ontology defines a set of entity and relationship types as discussed below.

3.2 Entity Types of MegaL

We distinguish three kind of entities: *abstract* entities—they appear at the mathematical level of thinking; *conceptual* entities—they are cognitive elements such as languages or technologies; *digital* entities—they correspond to artifacts that reside in and are processed by computers.

In this paper, we use these types of *abstract entities*: Entity, Set, Pair, Relation, Function, FunctionApplication (i.e., pairs pertaining to a function). For instance, functions are needed to model the meaning of tools or programs. Further, we use these types of *conceptual entities*: Language and Technology. Languages can be viewed (in a simplified manner) as sets. Technologies can be viewed as compound entities with components for tools, languages, and others. Finally, we use these types of *digital entities*: Artifact (the base type for the following types), File, Fragment (of a file), Program, Library, ObjectGraph.

The aforementioned entity types are just sufficient for the examples in this paper. The megamodel ontology can be extended to cover different domains, technological spaces, or engineering activities [8]. For instance, a megamodel in the context of model-driven engineering may benefit in clarity from additional digital entity types for models, metamodels, and model transformations.

3.3 Relationship Types of MegaL

Based on the fundamental relationships and the types of entities, as identified above, the following relationship types can be derived. Again, the list is trimmed down for the scope of this paper. We apply a UML-like convention to use ':Type' for a concrete (anonymous) entity of the given type.

- ◇ :Language **subsetOf** :Language
- ◇ :Artifact **elementOf** :Language
- ◇ :Language **domainOf** :Function
- ◇ :Function **hasRange** :Language
- ◇ :FunctionApplication **elementOf** :Function
- ◇ :Artifact **inputOf** :FunctionApplication
- ◇ :FunctionApplication **hasOutput** :Artifact
- ◇ :Artifact **conformsTo** :Artifact
- ◇ :Artifact **partOf** :Artifact
- ◇ :Artifact **correspondsTo** :Artifact
- ◇ :Artifact **dependsOn** :Artifact
- ◇ :Artifact **dependsOn** :Language
- ◇ :Artifact **realizationOf** :Function
- ◇ :Artifact **definitionOf** :Language
- ◇ :Program **partOf** :Technology
- ◇ :Library **partOf** :Technology

Megamodels initially just *declare* entities and relationships. Eventually, megamodels may be *linked* so that both entities and relationships are meaningfully demonstrated by actual artifacts of specific software products. This will be discussed in §6.

4 An Initial Megamodel for O/X Mapping

Megamodeling is demonstrated in this section for O/X mapping. In (schema-first) O/X mapping [19,25], one is concerned with marrying object-oriented programming with XML-based data representation such that an object model for data representation is generated from an XML schema and library functionality is responsible for mediating between XML documents ('files') and objects back and forth. The population of objects from XML data is also called deserialization whereas the other direction is referred to as serialization. The notion of O/X mapping is also known as XML data binding. In the context of the .NET platform, the term XML serialization is used as well.

4.1 Stepwise Development of the Megamodel

Let us develop an initial megamodel for O/X mapping, step by step. We use MegaL/TXT—this simple textual notation can express the same concepts as the visual notation MegaL/yEd that we used earlier. The textual notation comes with straightforward syntactic shorthands for recurring patterns [8] such as '$\rightarrow$' and '$\mapsto$' (instead of combinations of 'domainOf', 'hasRange', 'inputOf', 'hasOutput').

We begin with the *languages* involved in O/X mapping:

Languages *XSD, CSharp, XML, ClrObjectGraphs* .

The *C#* (or *CSharp*) language is mentioned because it is assumed here that schema-derived object models are represented in *C#*. We could make the object-oriented programming language a parameter of the megamodel, but we commit to *C#* here for concreteness' sake. *XSD* is the language of XML schemas. *XML* is the language of XML trees (or XML documents), i.e., the primary ('on file') representation format for data. Finally, *ClrObjectGraphs* is the language of object graphs. Again, we could make the in-memory representation of objects a parameter of the megamodel, but we commit to .NET's CLR representation here for concreteness' sake.

In fact, another language should be identified:

Language *CSharpFromXsd* ***subsetOf*** *CSharp .*

That is, *CSharpFromXsd* proxies for the C# subset that is used by the class generator of the O/X mapping technology. In conservative discussions of O/X mapping, this language is never articulated. However, awareness of this language and its characteristics helps understanding O/X mapping.

The characteristics of schema-derived object models vary indeed for each O/X mapping technology. In the case of .NET's O/X mapping technology, we can state the following characteristics for all $x \in$ *CSharpFromXsd*: (i) x declares classes only—as opposed to interfaces, enumerations, etc. (ii) The classes of x declare fields and properties as members, but no methods. (iii) x use attributes controlling XML serialization.

Let us now consider the major artifacts involved in O/X mapping. There are two type-level artifacts involved in such O/X mapping: an XML schema and an object model. There are also two instance-level artifacts involved: an actual XML document and an actual object graph:

File *xmlTypes* ***elementOf*** *XSD .*
File *ooTypes* ***elementOf*** *CSharpFromXsd .*
File *xmlDoc* ***elementOf*** *XML .*
ObjectGraph *clrObj* ***elementOf*** *ClrObjectGraphs .*

We also need to impose 'conformsTo' relationships as constraints on the instance-level artifacts: an arbitrary XML document would not be suitable; it must conform to the XML schema at hand; likewise for the object graph. Thus:

xmlDoc ***conformsTo*** *xmlTypes .*
clrObj ***conformsTo*** *ooTypes .*

Ultimately, we expect an O/X mapping technology to provide functionality for class generation and for deserialization (as well as serialization, which we skip here though). To this end, we introduce the following conceptual entities, in fact, functions, and we apply them in the expected manner to relate the artifacts at the type and instance levels. Thus:

Function *classgen : XSD* $\rightarrow$ *CSharpFromXsd .*
Function *deserialize : XML* $\rightarrow$ *ClrObjectGraphs .*
classgen(xmlTypes) $\mapsto$ *ooTypes .*
deserialize(xmlDoc) $\mapsto$ *clrObj .*

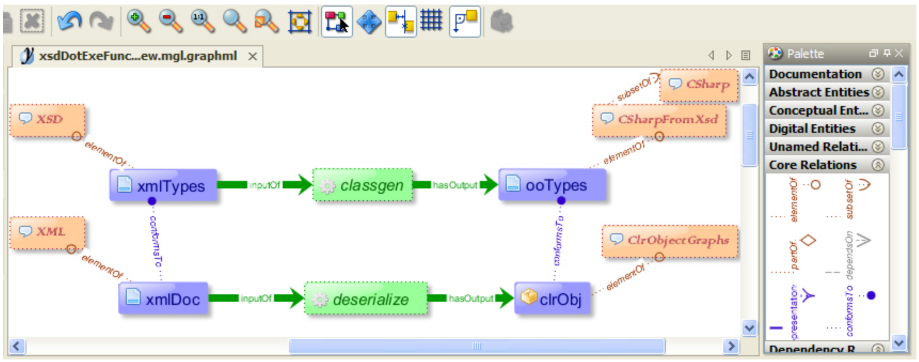


Fig. 3. An initial megamodel for O/X mapping drawn with the MegaL/yEd editor

4.2 Summary of the Megamodel

Figure 3 summarizes the megamodel in the form of a diagram drawn with the MegaL/yEd editor.[8] The visual and the textual notation convey the same information. Note that icons and colors are bound to entity types in the diagram. Some megamodel elements can be mapped to different visual elements. For instance, 'partOf' relationships are represented by node embedding in the upper frame of Figure 1, but a regular 'partOf' edge could also be used.

4.3 Discussion

The initial megamodel of this section was deliberately kept simple. This intermediate state also allows us to reflect on methodological questions of megamodeling:

- ◇ Do we model all important aspects of O/X mapping overall?
- ◇ What specifics of .NET's O/X mapping technology should be modeled?

Without focus on O/X mapping, these questions take the following form:

- ◇ Do we model all general aspects of the kind of technology at hand?
- ◇ What specifics of a concrete technology should be modeled?

It is relatively easy to observe that the megamodel could be enhanced to incorporate additional aspects of O/X mapping, overall. For instance, we did not yet model the fact that O/X mapping is carried out 'for a purpose': some OO program is meant to use the generated object model to implement data-processing functionality. As to the question of technology-specific aspects, we did not yet model the components of .NET's technology for O/X mapping. These and additional aspects are addressed in the following section.

[8] Our implementation uses yEd for megamodel editing `http://www.yworks.com/en/products_yed_about.html` with GraphML `http://graphml.graphdrawing.org/` for the representation.

5 A Megamodel for O/X Mapping with .NET

We advance the megamodel of the previous section to cover generally more aspects of O/X mapping and to also apply more directly to the situation for the .NET platform.

5.1 The Use of Schema-Derived Object Models

The value proposition of O/X mapping depends on the fact that it enables essentially OO programming on XML data. We capture this aspect in the megamodel by introducing a problem-specific program that is said to depend on the schema-derived object model. This is another placeholder for an entity that does not belong to the technology itself, but instead to the software product that uses the technology. Thus:

File *problemProgram* ***elementOf*** *CSharp .*
problemProgram ***dependsOn*** *ooTypes .*

5.2 Technology Components for .NET

The technology consists of a code-generation tool, `xsd.exe`, a library, hosted by the namespace *System.Xml.Serialization*, and custom attributes (annotations) for metadata.[9] We declare corresponding entities:

Program *xsdDotExe . —— the "xsd.exe" tool*
Library *XmlSerializer . —— namespace "System.Xml.Serialization"*
Language *XsdMetadata .*

We can model now the fact that the `xsd.exe` tool realizes the class generation functionality for O/X mapping. In fact, the tool also realizes additional functionality, e.g., related to O/R mapping. To this end, the tool can be used in different modes controlled through the command line or an API, but we do not model such variability here. Thus:

xsdDotExe ***realizationOf*** *classgen .*

Previously, we simply assumed a function, *deserialize*, for deserializing XML into objects, without though clarifying the origin of the function. It is the problem-specific program that essentially performs de-serialization. In fact, we assume that some part of the program realizes serialization by making appropriate use of .NET's library for XML serialization. Thus:

Fragment *deserialization* ***partOf*** *problemProgram .*
deserialization ***dependsOn*** *XmlSerializer .*
deserialization ***realizationOf*** *deserialize .*

We can also clarify the role of metadata in O/X mapping. We assume that, subject to an appropriate interpretation of 'partOf' for languages, the *C#* language indeed comprises a part for metadata such that metadata for O/X mapping is a subset of general metadata.

[9] `http://msdn.microsoft.com/en-us/library/ms950721.aspx`

Language *CSharpMetadata .*
CSharpMetadata ***partOf*** *CSharp .*
XsdMetadata ***subsetOf*** *CSharpMetadata .*

Also, we can capture the characteristics of schema-derived classes to depend on metadata for O/X mapping. We do not formalize other characteristics of *CSharpFromXsd*. Thus:

ooTypes ***dependsOn*** *XsdMetadata .*

5.3 Additional Linguistic Details

Let us call *problemLanguage*a problem-specific language underlying the involved type-level artifacts. We think of this language as being abstract, rather than concretely represented by XML trees or object graphs. This language can be viewed as a proxy for the domain that is covered with a Object/XML mapping effort.

Language *problemLanguage .*
xmlTypes ***definitionOf*** *problemLanguage .*
ooTypes ***definitionOf*** *problemLanguage .*

It remains to establish a correspondence relationship between XML and object types as well as the involved instances:

xmlTypes ***correspondsTo*** *ooTypes .*
xmlDoc ***correspondsTo*** *clrObj .*

At the instance level, the object graph, which is obtained by de-serialization, is expected to be a *representation of* the original XML document and *vice versa* such that the original document could be re-obtained by serialization from which we abstract here for simplicity.

At the type level, correspondence means that (ideally) XML schema and object model are related by bidirectional intermodel mappings (say, 'structure-preserving' bijections) modulo difficulties due to the O/X impedance mismatch [18]. The couple of de-serialization and serialization functionalities should be considered the concrete interpretation of these mappings at the instance level, but this view is not developed in detail here. More intuitively, we could say that there is 1:1 mapping of types driven by name equality or similarity, and for each couple of associated types there is also a correspondence at the 'member' level.

5.4 Discussion

We conclude with a discussion of potential directions for enhancing the megamodel. We have focused here on de-serialization, but serialization could also be of interest, if XML transformation or generation is to be modeled. Further, we have not modeled any variability or configurability admitted the mapping technology, as needed for advanced usage scenarios of the technology.

We claim originality for analyzing O/X mapping by megamodeling. For comparison, the arguably most comprehensive catalog of O/X mapping technologies [25] uses an informal metamodel to compare technologies (tools) on the grounds of capabilities and limitations—linguistically relevant entities and relationships are not considered.

6 Linked Megamodels

A difficulty with metamodeling and even more with megamodeling approaches resides in the high level of abstraction they involve. This difficulty is even exacerbated by megamodels that deal with technologies, as in the previous two sections, because of the gap between the abstract notation and the very concrete artifacts a software engineer deals with, e.g., some files or objects. As a result it may be hard to convince anyone that any given statement in the megamodel holds.

Linked megamodels close the gap between abstraction and concreteness by linking each entity in the megamodel to a web resource. Thus, an entity is no longer represented merely as an identifier, leaving all room for misunderstanding and misinterpretation; instead, the identifier is linked to a unique resource that can be browsed and examined at will. Relationships can also be linked. As a result, it becomes much easier to understand and to validate megamodels.

6.1 Binding Placeholder Entities

Note that in the megamodel of the previous two sections, artifact placeholders were used for some entities, e.g., *xmlDoc* and *clrObj*. When the goal is to validate or illustrate the megamodel, then it is useful to 'bind' placeholders to actual artifacts. This has been done in Figure 1 with the concrete artifacts being part of a particular software product. That is, X/O mapping is illustrated thanks to the *xsdClasses* implementation of the *101companies* project. For instance, the placeholder *xmlDoc* is bound to *Company.xsd*—an XML schema file of the *xsdClasses* implementation.

6.2 Exploring Linked Megamodels

From the end-user perspective, linked megamodels are seen as hypertext documents that can be explored. Figure 1 shows a screenshot of the MegaL/Explorer tool. The upper frame corresponds to a clickable image that is produced with MegaL/Editor. Within the context of the explorer, a click on an entity displays the corresponding resource in the lower frame. For instance, clicking on the *CSharp* node leads to a wiki page for C# according to the *101companies* project; clicking on `xsd.exe` node also leads to a page for the tool; clicking on a file, e.g., *Company.xsd*, displays the content of the file extracted from the *101companies* repository. Relationships (i.e., graph edges) are also clickable. In Figure 1, the user has selected the (circled) correspondence link between the *Company* fragments respectively in *Company.xsd* and *Company.cs*. As a result, the source fragments are shown side by side in the lower frame—clearly showing what the `xsd.exe` tool actually generates for a given example.

```
_:xmlTypes rdf:type mgl:File .
_:xmlTypes rdfs:label "xmlTypes" .
_:xmlTypes mgl:elementOf lang:XSD .
_:xmlTypes mgl:inputOf _:classgen .

_:xmlDoc rdf:type mgl:File .
_:xmlDoc rdfs:label "xmlDoc" .
_:xmlDoc mgl:elementOf lang:XML .
_:xmlDoc mgl:conformsTo _:xmlTypes .
_:xmlDoc mgl:inputOf _:classgen .

_:classgen_app_1 rdf:type mgl:FunctionApplication .
_:classgen_app_1 rdfs:label "classgen" .
_:classgen_app_1 rdf:elementOf _:classgen .
_:classgen_app_1 rdf:hasOutput _:ooTypes .

... etc. ...
```

Entities are associated with a prefix, e.g., `rdf`, corresponding to a unique URI (not shown here). This means that each entity is now associated with a URL where the corresponding resource can be found. Only 'blank nodes', i.e., those with the _ prefix, are local identifiers. The `rdf` and `rdfs` prefixes refers to RDF and RDFS definitions respectively. The prefix `mgl` refers to the MegaL ontology which contains definitions for both entity types (represented as OWL classes) and relationships types (represented as OWL properties).

Fig. 4. Figure 3 expressed in MegaL/RDF

```
_:CompanyDotXSD rdf:type mgl:File .
_:CompanyDotXSD rdfs:label "Company.xsd" .
_:CompanyDotXSD mgl:elementOf lang:XSD .
_:CompanyDotXSD mgl:inputOf _:CompanyXSD2CSDotBat .
_:CompanyElement mgl:partOf _:CompanyDotXSD .
_:CompanyElement rdf:type mgl:FileFragment .
_:CompanyElement rdfs:label "Company" .
... other fragments omitted ...

_:CompanyDotXSD mgl:partOf impl:xsdClasses .
_:CompanyDotXSD mgl:filename "./Company.xsd" .
_:CompanyElement mgl:xpathLocation
                     "//*[@name=\"Company\"]" .
... etc
```

The first block of triples shows some properties of the file `Company.xsd` including its decomposition into fragments.

The second block models links to online software artifacts. For instance the `impl` prefix refers to *101companies* implementations, `./Company.xsd` refers to a file name, and the property `mgl:xpathLocation` refers to a fragment of the schema file.

Fig. 5. RDF-based links for the megamodel of Figure 1

6.3 MegaL/RDF, Linked Megamodels and *Linked Data*

Technically, linked megamodels are represented in RDF by following *Linked Data*[10] principles. Figure 4 and Figure 5 show fragments of two megamodel expressed in MegaL/RDF as sets of triples while using RDF/turtle syntax. The first figure is concerned with the general megamodel for O/X mapping. It contains therefore placeholders with generic names, e.g. *xmlDoc* and *xmlTypes*). By contrast, the second figure is concerned with the bound megamodel for the *101companies* implementation *xsdClasses.* It contains product-specific names, e.g. , *CompanyDotXSD*, but also, and this is a very important aspect, links to concrete software artifacts, which should be considered as *resources* according to RDF principles.

[10] http://linkeddata.org/

Since all information in the *101companies* project is represented as RDF triples, links between *101companies* resources and external ones such as Wikipedia pages, i.e., dbpedia[11] resources in terms of RDF, this approach therefore enables the integration of megamodels and various other resources in the *Linked Data* global data space.

7 Related Work

Megamodeling. Megamodeling is somewhat established in the communities of modeling and model-driven engineering. Existing forms of megamodels do not cover the range of linguistic relationships of MegaL (such as 'elementOf', 'subsetOf', and 'correspondsTo'); they have not been used in a manner to understand software technologies across technological spaces. We look at representative examples. In [27], megamodeling is applied to the human-computer interaction domain. In [10], a UML/OCL-based megamodel of MDA/MDE is provided, thereby supporting reasoning about MDA/MDE. In [30], megamodeling is used for organizing and utilizing runtime models and relations in a model-driven manner while also supporting a high level of automation. In [15], megamodeling is used to coordinate "heterogeneous" models in the sense of conforming to a multiplicity of metamodels expressed in different DSLs. In [13], megamodeling is applied to model transformation with the objective of supporting the evolution of software architectures. In [14], some forms of megamodels and associated applications are surveyed.

Some model transformation approaches involve explicitly chains or compositions of transformations, perhaps even involving different model transformation languages and dealing with different 'modeling spaces'. Such compositions can be viewed as a form of executable megamodels. In [20], the authors motivate the need for a precise semantics for model-to-model transformations, thereby enabling verification of correctness for compositions, thereby, in turn, encouraging reusability.

Foundations of Modeling. Our work is substantially inspired by recent efforts on the foundation of modeling from which we derive basic idioms of megamodeling. We rely on established relationships such as 'conforms to' and 'element of' [9,16]. Further, there is the multi-faceted 'represents/models' relation [23,24]. We derive the correspondence relation from the field of model management. In [5,6], a categorical approach to intermodel mappings including heterogeneous (meta)model correspondences is developed.

Viewpoints. We consider megamodels as supporting another 'point of view' in the tradition of viewpoints in software development [12]. Viewpoints are used in practice, specifically for enterprise architecture [28], on the basis of the reference model RM-ODP[12] and the IEEE-1471 standard[13]. Each viewpoint is typically

11 http://dbpedia.org
12 http://www.rm-odp.net/
13 http://standards.ieee.org/findstds/standard/1471-2000.html

associated with one or more designated modeling languages [4] subject to different metaware [8] (i.e., metamodels and model-driven software technology). In this paper, we enable the linguistic point of view.

8 Conclusion

We have developed a form of modeling that targets the linguistic architecture of software technologies and software products. Megamodels serve as cognitive models for the benefit of software engineers, software linguists, and others.

We expect to advance the *101companies* project to provide megamodels systematically for a substantial number of software technologies and *101companies* contributions. We plan to use such megamodels in teaching software technologies to undergraduates. Without megamodels, it is very difficult to convey sufficiently abstract knowledge about, for example, technologies for Object/Relational/XML mapping in university courses.

Future research should advance the megamodeling approach in several dimensions. Megamodels should be extended to incorporate declarative descriptions of relationships for conformance, correspondence, membership, and others so that the cognitive value of such extended megamodels is improved. For instance, newly identified languages, such as the *C#* subset used by the generator in our example, could be properly described in this manner. Also, our understanding of intermodel mappings, such as the mapping between XML and object types, could be properly explained in this manner. Such descriptions could leverage language support for code queries.

The notion of linked megamodels will be advanced so that some links can be recovered semi-automatically from products that adhere to some tagging and naming rules. Also, static and dynamic program analysis will be leveraged so that the applicability of a generic megamodel to a specific product can be verified and eventually inferred.

Acknowledgement. The authors are grateful to a number of important discussions and exchanges with D. Gasevic and Z. Diskin that helped developing the present notion of megamodeling. An emerging notion was presented by the authors on different occasions at SLE/GPC 2011 (tutorial by J.M. Favre, D. Gasevic, and R. Lämmel), GTTSE 2011 (short participant presentation by A. Varanovich without publication), AOSD 2011 (tutorial by R. Lämmel), KPS 2011 (informal workshop presentation by R. Lämmel). The authors are grateful to several students—Martin Leinberger, Marius Rackwitz, and Thomas Schmorleiz—for their participation in this research.

References

1. Bézivin, J., Jouault, F., Valduriez, P.: On the need for Megamodels. In: Proceedings of the OOPSLA/GPCE: Best Practices for Model-Driven Software Development Workshop (2004)

2. Bézivin, J., Jouault, F., Rosenthal, P., Valduriez, P.: Modeling in the Large and Modeling in the Small. In: Aßmann, U., Aksit, M., Rensink, A. (eds.) MDAFA 2003. LNCS, vol. 3599, pp. 33–46. Springer, Heidelberg (2005)
3. Bräuer, M., Lochmann, H.: An Ontology for Software Models and Its Practical Implications for Semantic Web Reasoning. In: Bechhofer, S., Hauswirth, M., Hoffmann, J., Koubarakis, M. (eds.) ESWC 2008. LNCS, vol. 5021, pp. 34–48. Springer, Heidelberg (2008)
4. Dijkman, R.M., Quartel, D.A.C., Pires, L.F., van Sinderen, M.: An Approach to Relate Viewpoints and Modeling Languages. In: Proceedings of 7th International Enterprise Distributed Object Computing Conference (EDOC 2003), pp. 14–27. IEEE (2003)
5. Diskin, Z., Maibaum, T., Czarnecki, K.: Intermodeling, Queries, and Kleisli Categories. In: de Lara, J., Zisman, A. (eds.) Fundamental Approaches to Software Engineering. LNCS, vol. 7212, pp. 163–177. Springer, Heidelberg (2012)
6. Diskin, Z., Xiong, Y., Czarnecki, K.: Specifying Overlaps of Heterogeneous Models for Global Consistency Checking. In: Dingel, J., Solberg, A. (eds.) MODELS 2010. LNCS, vol. 6627, pp. 165–179. Springer, Heidelberg (2011)
7. Djuric, D., Gasevic, D., Devedzic, V.: The Tao of Modeling Spaces. Journal of Object Technology 5(8) (2006)
8. Favre, J.M.: CacOphoNy: Metamodel-Driven Architecture Recovery. In: Proceedings of 11th Working Conference on Reverse Engineering (WCRE 2004), pp. 204–213. IEEE (2004)
9. Favre, J.M.: Foundations of meta-pyramids: Languages vs. metamodels – Episode II: Story of thotus the baboon. In: Language Engineering for Model-Driven Software Development. No. 04101 in Dagstuhl Seminar Proceedings (2005)
10. Favre, J.M.: Foundations of Model (Driven) (Reverse) Engineering: Models – Episode I: Stories of The Fidus Papyrus and of The Solarus. In: Language Engineering for Model-Driven Software Development. No. 04101 in Dagstuhl Seminar Proceedings (2005)
11. Favre, J.M., Lämmel, R., Schmorleiz, T., Varanovich, A.: *101companies*: A Community Project on Software Technologies and Software Languages. In: Furia, C.A., Nanz, S. (eds.) TOOLS 2012. LNCS, vol. 7304, pp. 58–74. Springer, Heidelberg (2012)
12. Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., Goedicke, M.: Viewpoints: A framework for integrating multiple perspectives in system development. International Journal of Software Engineering and Knowledge Engineering 2(1), 31–57 (1992)
13. Graaf, B.: Model-Driven Evolution of Software Architectures. Dissertation, Delft University of Technology (2007)
14. Hebig, R., Seibel, A., Giese, H.: On the Unification of Megamodels. In: Proceedings of the 4th International Workshop on Multi Paradigm Modeling (MPM 2010) at the 13th IEEE/ACM International Conference on Model Driven Engineering Languages and Systems, MoDELS 2010 (2010)
15. Jouault, F., Vanhooff, B., Bruneliere, H., Doux, G., Berbers, Y., Bézivin, J.: Inter-dsl coordination support by combining megamodeling and model weaving. In: SAC, pp. 2011–2018 (2010)
16. Kühne, T.: Matters of (Meta-) Modeling. Software and Systems Modeling 5, 369–385 (2006)
17. Kurtev, I., Bézivin, J., Aksit, M.: Technological spaces: An initial appraisal. In: CoopIS, DOA 2002 Federated Conferences, Industrial Track (2002)

18. Lämmel, R., Meijer, E.: Revealing the X/O impedance mismatch (Changing lead into gold). In: Backhouse, R., Gibbons, J., Hinze, R., Jeuring, J. (eds.) SSDGP 2006. LNCS, vol. 4719, pp. 285–367. Springer, Heidelberg (2007)
19. Lämmel, R., Meijer, E.: Mappings Make Data Processing Go 'Round. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 169–218. Springer, Heidelberg (2006)
20. Lano, K., Rahimi, S.K.: Model-Driven Development of Model Transformations. In: Cabot, J., Visser, E. (eds.) ICMT 2011. LNCS, vol. 6707, pp. 47–61. Springer, Heidelberg (2011)
21. Mah/'e, V., Perez, S.M., Brunelière, H., Doux, G., Cabot, J.: PORTOLAN: a Model-Driven Cartography Framework. Tech. rep., INRIA (2011), # 7542
22. Melnik, S., Adya, A., Bernstein, P.: Compiling mappings to bridge applications and databases. In: SIGMOD 2007: Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, pp. 461–472. ACM (2007)
23. Muller, P.A., Fondement, F., Baudry, B.: Modeling Modeling. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 2–16. Springer, Heidelberg (2009)
24. Muller, P.A., Fondement, F., Baudry, B., Combemale, B.: Modeling modeling modeling. Software and Systems Modeling, 1–13 (2011)
25. Ronald Bourret: Xml data binding resources (2001–2012), http://www.rpbourret.com/xml/XMLDataBinding.htm
26. Salay, R., Mylopoulos, J., Easterbrook, S.M.: Using Macromodels to Manage Collections of Related Models. In: van Eck, P., Gordijn, J., Wieringa, R. (eds.) CAiSE 2009. LNCS, vol. 5565, pp. 141–155. Springer, Heidelberg (2009)
27. Sottet, J.S., Calvary, G., Favre, J.M., Coutaz, J.: Megamodeling and Metamodel-Driven Engineering for Plastic User Interfaces: MEGA-UI. In: Human-Centered Software Engineering. Springer Human-Computer Interaction Series, pp. 173–200 (2009)
28. Steen, M.W.A., Akehurst, D.H., ter Doest, H.W.L., Lankhorst, M.M.: Supporting Viewpoint-Oriented Enterprise Architecture. In: Proceedings of 8th International Enterprise Distributed Object Computing Conference (EDOC 2004), pp. 201–211. IEEE (2004)
29. Thomas, D.: The Impedance Imperative: Tuples + Objects + Infosets = Too Much Stuff! Journal of Object Technology 2(5), 7–12 (2003)
30. Vogel, T., Seibel, A., Giese, H.: The Role of Models and Megamodels at Runtime. In: Dingel, J., Solberg, A. (eds.) MODELS 2010. LNCS, vol. 6627, pp. 224–238. Springer, Heidelberg (2011)

Cross-Language Support Mechanisms Significantly Aid Software Development

Rolf-Helge Pfeiffer and Andrzej Wąsowski

IT University of Copenhagen, Denmark
{ropf,wasowski}@itu.dk

Abstract. Contemporary software systems combine many artifacts specified in various modeling and programming languages, domain-specific and general purpose as well. Since multi-language systems are so widespread, working on them calls for tools with cross-language support mechanisms such as (1) visualization, (2) static checking, (3) navigation, and (4) refactoring of cross-language relations. We investigate whether these four mechanisms indeed improve efficiency and quality of development of multi-language systems. We run a controlled experiment in which 22 participants perform typical software evolution tasks on the JTrac web application using a prototype tool implementing these mechanisms. The results speak clearly for integration of cross-language support mechanisms into software development tools, and justify research on automatic inference, manipulation and handling of cross-language relations.

1 Introduction

Developers building contemporary software systems constantly deal with multiple languages at the same time. For example, around one third of developers using the Eclipse IDE work with C/C++, JavaScript, and PHP, and a fifth of them use Python besides Java [1]. PHP developers regularly use one to two languages besides PHP [2]. Developers of large enterprise systems face a particularly complex challenge. For instance, OFBiz, an industrial quality open-source ERP system combines more than 30 languages including General Purpose Languages (GPLs), several XML-based Domain-Specific Languages (DSLs), along with configuration files, property files, and build scripts. ADempiere, another industrial quality ERP system, uses 19 languages. The eCommerce systems Magento and X-Cart utilize more than 10 languages each.[1]

We call systems using multiple languages, *Multi-Language Software Systems (MLSSs)*. Obviously, the majority of modern software systems are MLSSs.

To demonstrate how disturbing development of MLSSs is, lets consider an example extracted from JTrac, an open-source, web-based bug-tracking system. JTrac's login page (Fig. 1) is implemented in three source code files in three different languages. The login page itself is described in HTML (Lst. 3), displayed messages are given in a properties file (Lst. 1), and the logic evaluating a login

[1] See ofbiz.apache.org, adempiere.com, magentocommerce.com, x-cart.com

```
1 login . title  = JTrac Login
2 login .home = Home
3 login .loginName = Login Name / email ID
4 login .password = Password
5 login .rememberMe = remember me
6 login .submit = Submit
7 login . error  = Bad Credentials
```

Listing 1. A fragment of a properties file

```
1 private  class  LoginForm extends StatelessForm {
2   private  String loginName;
3   private  String password;
4   public  String getLoginName() {
5          return  loginName;
6   }
7   public  String getPassword() {
8          return  password;
9   }
```

Listing 2. A fragment of Java login logic

```
 1 <table class ="jtrac">
 2  <tr>
 3   <td class ="label"><wicket:message key="login.loginName"/></td>
 4   <td colspan="2"><input wicket:id="loginName" size="35"/></td>
 5  </tr>
 6  <tr>
 7   <td class ="label"><wicket:message key="login.password"/></td>
 8   <td><input type="password" wicket:id="password" size="20"/></td>
 9   <td align ="right">
10    <input type="submit" wicket:message="value:login.submit"/>
11   </td>
12  </tr>
```

Listing 3. A fragment of the HTML code describing JTrac's login page

is described in Java (Lst. 2). The HTML code describes the structure of the login page and its contents—how the input fields for login and password insertion are laid out and how they are ordered. Since JTrac is built using the web-development framework *Wicket*, the HTML code contains wicket identifiers, which serve as anchors for string generation or behavior triggering, see lines 3, 4, 7, 8, and 10 in Lst. 3. The properties provide certain messages for the login page. For instance, the property on line 3 in Lst. 1 provides the message string for line 3 of the HTML code. The Java code (Lst. 2) provides authentication logic. Most of this code is not shown here, to conserve space. In order, to correctly invoke the Java code, the field names (lines 2–3), the corresponding get methods (lines 4 and 7), and the set methods (not shown), must use the same name as the wicket identifiers on lines 4 and 8 in Lst. 3.

Now, imagine that a developer renames the string literal login.loginName on line 3 in the HTML code to login.loginID. Obviously, the relation between the properties file (line 3) and the HTML file is now broken. In effect, the message asking for a login name is not displayed correctly anymore, see Fig. 2. The mistake is only visible at runtime. Observe that such small quiet changes of behavior can easily be missed by testers. Similarly, renaming the string literal loginName on line 4 in Lst. 3 to loginID breaks a relation to the field loginName in the Java file (affects lines 2, 4, and 5 in Lst. 2). The effect of this change is even more serious since JTrac crashes with an error page, see Fig. 3.

We believe that development of MLSSs could be significantly improved if Integrated Development Environments (IDEs) included support for multi-language development, known from single languages, such as (i) visualization (ii) static checking for consistency, (iii) navigation and (iv) refactoring of cross-language

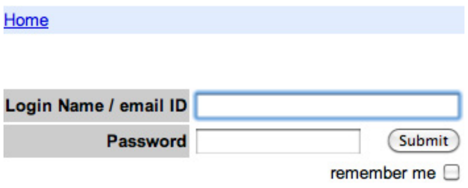


Fig. 1. Error-free login page

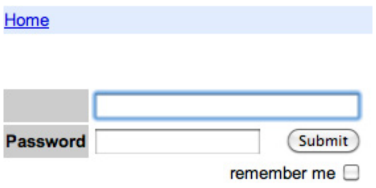
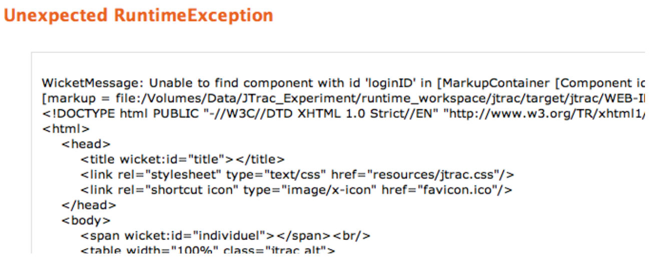


Fig. 2. Login page with a broken relation between HTML and property code



Unexpected RuntimeException

```
WicketMessage: Unable to find component with id 'loginID' in [MarkupContainer [Component ic
[markup = file:/Volumes/Data/JTrac_Experiment/runtime_workspace/jtrac/target/jtrac/WEB-II
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/
<html>
   <head>
      <title wicket:id="title"></title>
      <link rel="stylesheet" type="text/css" href="resources/jtrac.css"/>
      <link rel="shortcut icon" type="image/x-icon" href="favicon.ico"/>
   </head>
   <body>
      <span wicket:id="individuel"></span><br/>
      <table width="100%" class="jtrac alt">
```

Fig. 3. Login page with a broken relation between HTML and Java code

relations. In the remainder we refer to these four mechanisms as *Cross-Language Support (CLS)* mechanisms. In this paper, we address the following research question on CLS:

> Do Cross-Language Support mechanisms improve developer's understanding of the system and reduce the number of errors made at development time?

To investigate this question we run a controlled experiment in which 22 participants perform typical development and customization tasks on JTrac, a representative MLSS.

It is well known that maintenance and customization of software systems is expensive and time consuming. Between 85% to 90% of project budgets go to legacy system operation and maintenance [5]. Lientz et al. [12] state that 75% to 80% of system and programming resources are used for extensions and maintenance, where alone understanding of the system stands for 50% to 90% percent of these costs [18]. The results of our experiment demonstrate (i) that developers using CLS mechanisms find and fix more errors in a shorter time than those in the control group, (ii) that they perform development tasks on language boundaries more efficiently, and (iii) that even unexperienced developers provided with CLS perform similarly or better than experienced developers in developing MLSSs. Clearly, the integration of CLS into IDEs and development tools would contribute to reducing the high cost of software maintenance and evolution. These results confirm the importance of research on interrelating models and modeling languages, such as trace models [7,14], multi-modeling [8], mega-models [9], macromodels [17], and relation models [15,16]. Additionally, the results motivate research on automatic inference of cross-language relations.

The JTrac system plays a role of the experimental unit in our setup. We use a prototype development editor, TexMo, as the experimental variable, by enabling and disabling its cross-language support. JTrac and TexMo are presented in Sect. 2. Section 3 describes our methodology and the setup of the experiment. We analyze the results in Sect. 4, discuss threats to validity in Sect. 5 and related work in Sect. 6. We conclude in Sect. 7.

Experiment artifacts referred in this paper are available online at www.itu.dk/people/ropf/download/Experiment.zip. The archive contains TexMo's source code, the JTrac instance used for the experiment, all documents, questionnaires, answers, and statistics. Screen captures are available on request, as they take up a lot of space.

2 Technical Background

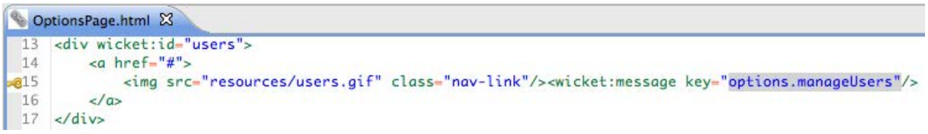
2.1 JTrac: An MLSS Representative

We use the open-source web-based bug-tracking system JTrac as an experimental unit in our experiment. JTrac's code base contains 374 files of which the majority (291) contain code: Java (141), HTML (65), property files (32), XML (16), JavaScript (8), and 29 other source code files such as Shell scripts, XSLT transformations, etc. The remaining 83 files are images such as ".png", ".gif" and a single jar file. Most of the property files are used for localization of system messages. The XML files are used for various purposes, for example to give an object-relational mapping describing how to persist business objects. As many other web-applications, JTrac implements the model-view-controller pattern. This is achieved using popular frameworks: Hibernate (hibernate.org) for object-relational mapping and Wicket (wicket.apache.org) to couple views and controller code. Clearly, JTrac is a MLSS.

2.2 TexMo: A Multi-language Programming Environment

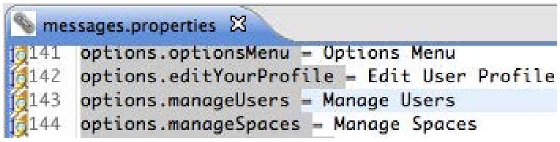
TexMo is a prototype of a Multi-Language Development Environment [16] developed by Pfeiffer. It is an editor that allows to interrelate source code in multiple languages. TexMo uses a *relation* metaphor. Relations are defined between *references* and *keys*. A key is a fragment of code that introduces an identifiable object, a concept, etc. A reference is a location in code that relates to a key. Relations are always many-to-one between *references* and *keys*. TexMo addresses MLSSs development by implementing the CLS mechanisms as follows:

1. *Visualization.* TexMo highlights keys and references in gray. See reference from l. 143 in Fig. 4b to l. 15 in Fig. 4a. Keys are labeled with a key icon and references are labeled by a book icon; see Fig. 4, left to line numbers. Inspecting markers reveals further details, such as how many references and in which files refer to a key.



```
<div wicket:id="users">
    <a href="#">
        <img src="resources/users.gif" class="nav-link"/><wicket:message key="options.manageUsers"/>
    </a>
</div>
```

(a) HTML code, which fills a message given by a property name.



```
options.optionsMenu = Options Menu
options.editYourProfile = Edit User Profile
options.manageUsers = Manage Users
options.manageSpaces = Manage Spaces
```

(b) A properties key options.manageUsers.

Fig. 4. Declaration of a Wicket id and its use

2. *Navigation.* Users can access the key from any of its reference and navigate from a key to any of its references. Navigation is activated via a context menu.
3. *Static checking.* TexMo statically checks cross-language relations. Broken relations are underlined red and labeled by a standard error indicator, see Fig. 5.
4. *Refactoring.* Broken relations can be fixed automatically by applying quick fixes. TexMo's quick fixes are key centric rename refactorings. Applying a fix to a key renames all references to the content of the key. Dually, applying a quick fix to a reference renames this single reference to point to its key.

TexMo is an Eclipse plugin. It uses a universal model for representation of any textual language. That is, any source code file is an instance of an EMF-based DSL, which relies on the physical structure of its text. Code is represented as paragraphs, words, parts of words, characters, and special characters like dots or semicolons. This universal representation of source code permits the use of a universal *relation model*, to track relations across different programming artifacts, to link arbitrary information across language boundaries and to synchronize these relations whenever programming artifacts are modified by developers. Further information about TexMo is available in [16].

3 The Experiment

We run a controlled experiment with 22 participants divided into two groups. The control group A performs the tasks using TexMo with CLS disabled. The treatment group B uses TexMo with all four CLS mechanisms enabled.

3.1 Hypotheses

We refine the initial research question into five specific hypotheses:

H1. *Developers using CLS find and fix more errors than the developers in the control group.* This hypothesis aims at capturing the effectiveness of CLS.

Since developers get more support by the IDE guiding to problems and offering possible solutions, we expect them to find and fix more errors.

H2. *Using CLS does not have negative impact on speed of work.* Since CLS provides more information that need to be processed by developers, it could take longer working with CLS than without it (due to information flooding).

H3. *The least experienced developers using CLS perform better than the most experienced developers in the control group.* Since we expect experienced software developers to perform better than non-experienced developers, it is interesting to investigate how close non-experienced developers can be brought to the quality and performance of experienced ones by just offering CLS. Note, that we refer to general experience in software engineering not experience related to the experimental unit JTrac.

H4. *Developers using CLS locate errors in source code, whereas developers in the control group identify effects of errors.* We expect developers in the treatment group, those using CLS, to describe errors on a different level of abstraction. They will locate errors, i.e., which code constructs in relation with others are responsible for erroneous behavior, whereas developers in the control group will identify effects of errors, i.e., the erroneous behavior of the system. This would mean that developers using CLS have a deeper understanding of the implementation of the system under development.

H5. *Developers use CLS mechanisms.* We expect developers offered CLS mechanisms to actually use them voluntarily.

3.2 Experiment Design

We use the terminology of Juristo and Moreno [10, Chpt. 4.2] in our description.

The Experimental Unit. JTrac is a representative of a MLSS. It uses more than 5 languages and with its size of nearly 300 source code files it is sufficiently large to not be easily understandable by the experiment subjects within the given time.

The Experimental Variable. We used TexMo as an IDE with CLS. We are not aware of any other tool supporting the four CLS mechanisms simultaneously. Other existing tools either only provide CLS for particular pairs of languages like IntelliJ IDEA (jetbrains.com/idea), are no longer available, like X-Develop [20], or they do not implement all four mechanisms simultaneously. Also, since TexMo is an Eclipse extension it allows the participants to work in a familiar environment.

Factors. We follow a single-factor with two alternatives experiment design. The factor alternatives are TexMo with visualization, navigation, static checking and refactoring of cross-language relations disabled and the full-featured TexMo as described in Sect. 2. Group B uses the full-featured TexMo and the control group, Group A, uses the restricted TexMo. The latter simulates using a modern IDE.

The Response Variables. We have four response variables representing all quantitative outcomes: number of found errors, number of fixed errors, and the times for finding and fixing errors.

The Pre-Experiment. Before the actual experiment we ran a pre-experiment with five participants, three using the full-featured TexMo editor and two using the control group version. The purpose of the pre-experiment was to check if the experiment tutorials, task descriptions, and objects are consistent, correct and can be understood. In response to the results of the pre-experiment we have fixed incorrect file paths, typos, and wrong line numbers in the task document, and we improved error markers in the TexMo editor. The participants of the pre-experiment have not been used in the main experiment to avoid learning effects. The results of the pre-experiment are not included in the statistics below.

The Pre-Questionnaire. To avoid bias in the distribution of participants in two groups with similar technical experience, we let everyone answer a short questionnaire prior to the actual experiment. We asked 13 yes-no questions about the technical experience of participants: did they develop web-applications before and whether they know and used the web-application frameworks Wicket (wicket.apache.org) or Spring (springsource.org), the object relational mappers Cayenne (cayenne.apache.org) or Hibernate (hibernate.org), the IDEs VisualStudio (microsoft.com/visualstudio) or Eclipse (eclipse.org).

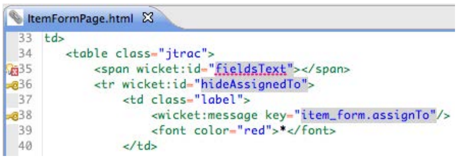
Only Wicket, Hibernate, and Eclipse are used in the experiment but we asked for alternative technologies to minimize the risk that a participant tries to learn about an important technology before the actual experiment.

The Experimental Subject. This experiment is conducted with 22 experimental subjects falling into four major categories: software professionals along with PhD, MSc, and undergraduate students at The IT University of Copenhagen.

The youngest participant is 18 and the oldest is 48, average age is around 29 years, median 28. Nineteen participants report that they have been working as professional software engineers for at least half a year, with maximum of 13 years (average work experience: around 3 years, median 3 years). Two PhD and one graduate student have no experience as professional software engineers.

We distributed the subjects in two groups, one per factor alternative. The distribution was solely based on technological experience reported in the pre-questionnaire, described above.

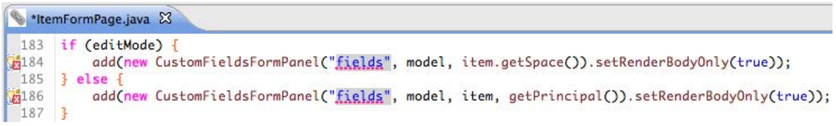
From the 22 participants, 19 reported to have experience with web-application development, 1 already used Wicket, 5 used Hibernate, and 20 have experiences using the Eclipse IDE. Participants were assigned randomly to distribute them equally according to their experience. In Group A, 10 persons have experience developing web-applications, none of them used Wicket before, 2 of them used Hibernate, and 9 used the Eclipse IDE. Similarly, in Group B, 9 persons have developed web-applications before, 1 of them used Wicket, 3 of them Hibernate, and 9 of them Eclipse.



ItemFormPage.html

```
33 td>
34     <table class="jtrac">
35         <span wicket:id="fieldsText"></span>
36         <tr wicket:id="hideAssignedTo">
37             <td class="label">
38                 <wicket:message key="item_form.assignTo"/>
39                 <font color="red">*</font>
40             </td>
```

(a) Declaration of the fieldsText id attached to a span tag.



*ItemFormPage.java

```
183 if (editMode) {
184     add(new CustomFieldsFormPanel("fields", model, item.getSpace()).setRenderBodyOnly(true));
185 } else {
186     add(new CustomFieldsFormPanel("fields", model, item, getPrincipal()).setRenderBodyOnly(true));
187 }
```

(b) Java code that fills a panel to the span HTML element

Fig. 5. Declaration of a Wicket id and its use

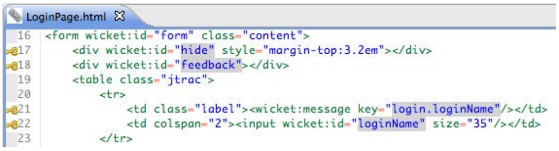
The demographic characteristics of the sample were established using a post-questionnaire (see below). Group A's average age is 29 years, with a median of 28, average work experience is 3.67 years with a median of 3 years. For Group B the age average is 28.64 years with a median of 30, and average work experience is 3.22 years with a median of 3 years.

The Tutorials. At the beginning of the experiment each participant received a tutorial explaining how to compile, start, and stop JTrac. Group B received an extended version explaining the CLS mechanisms of TexMo. To reduce bias, all features are described using an example in a different domain than the one used for the experiment—the development of a Safari browser extension.

The Tasks. The subjects were asked to perform three tasks representing typical development and customizations tasks. Each task had to be completed, including a brief per task questionnaire, within the 10 minutes. After 8 minutes the participant was reminded that only two minutes were left. After 10 minutes the participants were asked to proceed to the next task. We recorded screen contents of subjects solving the tasks.

Task 1. The participants received an instance of JTrac in which a cross-language relation was broken. Figure 5 shows the error: we renamed fields to fieldsText, the wicket:id attribute in a span tag of the HTML code in line 35. This string literal serves as a key for two references in the corresponding Java code. The renaming leads to a runtime error whenever a new issue report is added to the system.

The participants where asked to locate the error in the source code, name all files which contribute to the error, and to fix the error. The error can be fixed by renaming the key fieldsText to fields or conversely by renaming the references from fields to fieldsText. We considered both solutions as valid fixes.



LoginPage.html

```
16 <form wicket:id="form" class="content">
17     <div wicket:id="hide" style="margin-top:3.2em"></div>
18     <div wicket:id="feedback"></div>
19     <table class="jtrac">
20         <tr>
21             <td class="label"><wicket:message key="login.loginName"/></td>
22             <td colspan="2"><input wicket:id="loginName" size="35"/></td>
23         </tr>
```

Fig. 6. Declaration of a Wicket id and its use

```
<tr>
    <td class="label"><wicket:message key="logon.logonID"/></td
        >
    <td colspan="2"><input wicket:id="logonID" size="35"/></td>
</tr>
```

Listing 4. HTML code replacing lines 20 to 23 in Fig. 6

Task 2. We asked to rename the property options.manageUsers in line 143 Fig. 4b to options.manageAllUsers. This renaming breaks a cross-language relation between a properties file and HTML code. The system will still run error free but a message next to an icon on JTrac's administration page is not displayed anymore.

The participants were asked to name all files contributing to the newly introduced error and to fix the error. We recognize both renaming options.manageUsers to options.manageAllUsers in the HTML code and reverting the change applied to the properties file as valid solutions.

Task 3. The participants were asked to replace a block of code. Figure 6 shows the HTML code of JTrac's login page. Lines 20–23 implement a table row displaying labeled input fields. Line 21 contains a key login.loginName and line 22 contains another key loginName. A property file providing the text labels refers the login.loginName. The loginName key is referred from a Java class that evaluates user's input.

The participants were asked to replace the code block in lines 20–23 with the HTML code given in Fig. 4. Replacing this block removes two keys and breaks several references across three files in different languages. We asked the participants to name all files containing dangling references and to explain how to fix the problem.

The Post-Questionnaire. The post-questionnaire gathered both qualitative and quantitative data, mostly about the demographics: age, length of professional experience, size of developed systems, experience in web-development, familiarity with IDEs, whether they tried to learn technologies mentioned in the pre-questionnaire. Some of the questions overlapped with the pre-questionnaire, to verify consistency, or to check for temporal changes. We also asked if a participant experienced problems working with TexMo, and whether TexMo could be beneficial for software development, to collect feedback about our tooling.

Table 1. Success rate per task (n/a=not applicable). Each group has 11 members

	Task 1		Task 2		Task 3		Average	
	A	B	A	B	A	B	A	B
error located	9.09%	100%	45.45%	100%	0%	100%	18.18%	100%
error effect located	45.45%	n/a	36.36%	n/a	90.9%	n/a	57.57%	n/a
error fixed	0%	100%	45.45%	100%	qualitative		22.72%	100%

4 Results

H1. Developers using CLS find and fix more errors than the developers in the control group. We distinguish between *locating* an error and observing its *effect*. A participant locates an error if she properly names all files contributing to an error and navigates to corresponding lines within the code. She only observes the effect of an error if she runs the application and identifies erroneous behavior.

The results are summarized in Tab. 1. All developers in Group B successfully locate errors in all tasks. Only one developer in Group A locates the error in Task 1, five locate the error introduced in Task 2, and none is able to locate the errors in Task 3. Four developers in Task 1 and five developers in Task 3 managed to partly locate errors, indicating some files contributing to an error but not all.

Tasks 1 and 2 ask the participants to fix the errors. In Task 3 the participants explain how to fix the problem. This is why Tab. 1 contains no success rates for fixing errors for Task 3. All members of Group B fix the error in Task 1, compared to none in Group A. In Tasks 2 and 3, a substantially larger fraction of participants fixes the errors in Group B than in Group A. On average Group B is around five times more effective in locating errors than Group A and nearly four times better in fixing errors than Group A. We conclude that CLS significantly improves effectiveness of locating and fixing errors in the presented case.

H2. Using CLS does not have negative impact on speed of work. We measure the time to locate errors (Group B) or observe effects of errors (Group A) and the subsequent times to fix identified errors (both groups). The results per task are illustrated in Fig. 7 ($\triangle$ and $+$ symbolize outliers outside 3 times, or between 1.5-3 times the interquartile range). We only report the time for participants completing a task, at least partly within the given time.

Group B finds and fixes errors faster than Group A, in Tasks 1 and 2. For Task 3 Group A is slightly faster than Group B. But remember that we give the time to observe an error's effect for Group A and the time to locate an error for Group B. To fix the error the members of the control group would still need to locate it.

In Task 1 (column 1 in Fig. 7) only six participants in the control group locate the error and none of them is able to fix it. Consequently, there is no corresponding box-plot in the second column of Fig. 7. In Task 2, only five participants in

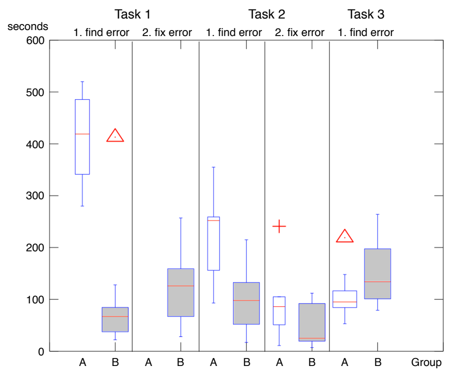


Fig. 7. Time to find and fix errors per group and task in seconds

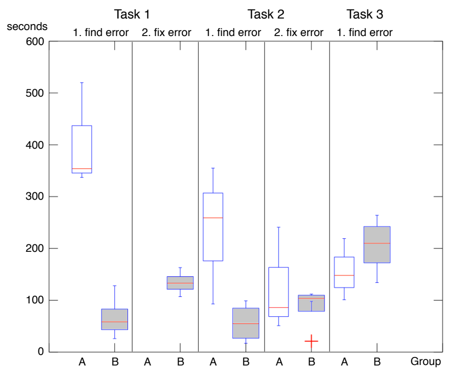


Fig. 8. Time to find and fix errors for the most experienced third of Group A and the least experienced in Group B

Group A suceed to locate and fix the error. For Task 3 ten Group A participants locate the error. All eleven participants in Group B locate and fix all the errors in all tasks (100% success rate). For Task 3, Group A members are slightly faster. This is because the observable error effect appears directly on the login page and is easy to find. Still, members of Group A are not able to find all files contributing to the error, see Tab. 1.

Since Group B is always similarly fast (Task 3) or faster (Tasks 1 and 2) than Group A, we conclude that CLS does not have negative impact on effectiveness in the presented case.

H3. The least experienced developers using CLS perform better than the most experienced developers in the control group. We ordered participants in both groups based on age, professional experience, experience in engineering of large software systems and web-applications, and the size of developed systems. In our sample, high experience correlates with age, work experience, experience in development of large systems, and with the sizes of systems developed. We compare the four most experienced developers in Group A with the four least experienced in Group B. Figure 8 illustrates the time used per task. We give the time until a participant observed the effect of an error for Group A and the time to locate an error for Group B. Only three of the selected four members in Group A contribute data to the analysis, since one of the participants did not finish the tasks within the allotted ten minutes.

Clearly, the least experienced members of Group B are faster in locating errors than the most experienced members of the control group, in Tasks 1 and 2. Again, in Task 3 the error is easily observable directly on the login page. Group A members are slighty faster in finding the effect but do not find all files contributing to the error.

Table 2. Rate of Group B participants using CLS per task

	Task 1	Task 2	Task 3
read markers	100%	100%	100%
used navigation	63.63%	72.72%	18.18%
used refactoring	63.63%	45.45%	qualitative

For errors which are not easily observable, developers exploiting CLS are faster in finding and fixing errors than developers without them, despite disadvantageous difference in reported experience.

H4. Developers using CLS locate errors in source code, whereas developers in the control group identify effects of errors. Members of Group B always locate errors successfully, see Tab. 1. Only one participant in Group B decided to start JTrac but did not even look at it. Significantly less members of Group A locate errors. Usually, they do observe the effects, and only subsequently they search for error locations if time is left. They rely on text search within the code base for locating the errors.

Members of Group B reason right from the beginning about abstract structures of the implementation, rather than merely observing effects of errors. This increases their effectiveness as indicated by the following quote from a post-questionnaire: I *liked the references part and the checking. Usually, if you change the keys/references you get errors at runtime which is kind of late in the process.* At the same time members of Group A are often not aware, that errors are caused by broken cross-language relations, as seen from their task notes. They either do not locate the error, or admit that they do not know the reason, or simply repeat the error message from the running system. Clearly, members of Group B work on a higher cognitive level than members of Group A.

H5. Developers use CLS mechanisms. All participants in Group B actually use visualizations. They actively investigate error markers by hovering the mouse cursor over them to get more detailed error descriptions. Over 45% of participants for Tasks 1–2 use navigation and automatic refactoring, see Tab. 2. Most do not use navigation, when replacing a code block, since they just deleted the keys, which they could use as navigation start points. Those participants who used navigation did so by undoing the changes and calling navigation from the old keys. This indicates need for new user interface design that would allow accessing deleted relations in a natural manner. No participants used automatic refactoring for Task 3, since TexMo does not implement automatic inference of possible keys out of the newly inserted code.

Furthermore, members of Group A complain that there is no static checking for the errors created when breaking cross-language relations. They expect this feature from an IDE searching for error markers or warnings. It is *difficult to* *identify the errors [and]...to navigate through the source code structure.* Contrary, Group B members not only do use CLS mechanisms, but also admit that *[TexMo] solves [a] commonly experienced problem when software project involves multiple languages.* These results strengthen our believe that higher speed and success rate in Group B is not accidental, but indeed caused by the availability of CLS mechanisms in their version of TexMo.

5 Threats to Validity

To ensure that the results and conclusions in Sect. 4 are statistically sound, we test hypotheses H1 to H3 statistically. Hypothesis H4 relies on qualitative data and hypothesis H5 only observes behavior of Group B, the treatment group. We apply a χ-test to the sample data for hypothesis H1 and Student's t-test to the sample data of hypotheses H2 and H3. The effective null-hypothesis for every test is that there is no difference between the experimental factor's alternatives ($\mu_A = \mu_B$), so CLS mechanisms do not aid software developers measurably.

We reject the null-hypothesis for H1, as all p-values are below significance level (0.05), meaning that for all tasks the alternative providing CLS has a significant impact on developers. For Tasks 1–3 developers in the treatment group perform significantly better than in the control group.

Testing H2 and H3, results in a statistically significant performance gain for the treatment group to locate errors, except if the errors are easily observable. However, performance is not statistically significantly better for fixing errors if we apply the test to the part of the control group that suceeded (no time to fix the error is available for the subjects who failed). Applying the t-test assuming time larger than 10 minutes for those participants who did not complete the tasks, confirms a significant performance improvement when fixing errors using CLS in Tasks 1–2. All statistical test data is available in the online appendix.

Internal Threats to Validity. The extended tutorial, explaining TexMo's features, might have caused a learning effect on members of Group B. They might have been more aware of cross-language relations. We believe that these effects are sufficiently minimized by choice of an example from a completely different domain. Also we assumed that in a standard development scenario, the developers would be aware of CLS support, either through reading manuals or by observing user interface visualizations. A tutorial might have helped them to use them faster in the beginning, which is justified within a frame of a short experiment task. Undoubtedly, they would be able to use the CLS mechanisms even more fluently, if they applied them in a daily work.

Arguably, the sample sizes for H1 tests are very small, while the χ-test is best applied for larger frequencies [10]. We used it, mostly to get a feeling for the data and to give an indication for a trend. Extending the experiments with more participants will have to prove this trend. Similarly, sample sizes pose a threat to validity when testing H2 and H3 with t-tests; in particular, testing H3 where three data points of Group A are compared to four data points of

Group B is questionable. Note though, that comparing to similar experiments in related work [21,19] our sample size is large. Indeed this is the largest controlled experiment about CLS mechanisms, that we are aware of.

It can be questioned if times for locating errors (Group B) are at all comparable with times for just observing their effects (Group A). We believe that this is not a problem since for the control group the time to observe the effect is a lower bound for the time to locate an error. So we compare an optimistic under approximation with complete time, and Group B still performs favorably.

External Threats to Validity. We ran a blind experiment. We tried to minimize bias of the participants by relying on written questionnaires and provided only minimal help on request. Typical help was to point the participants to the appropriate Ant task to compile and run JTrac.

There is a risk that participants could have learned about technologies after answering the pre-questionnaire. In the post-questionnaire we re-evaluate the known technologies and note that only four participants learned about a previously unknown technology. Two of them studied Cayenne and Spring respectively, which poses no threat as they are not used in the experiment. Another two learned about Wicket and Hibernate. Since they fall in two separate groups we do not think that this poses a threat to our grouping.

If our subjects were JTrac experts, they would be able to apply the fixes faster and the disparity would likely be smaller. However, the task of changing unknown code is a common scenario, so the results are valuable.

The factor alternative for control group, with disabled CLS, is not a plain Eclipse. TexMo does not implement all features of Eclipse editors. In particular it does not implement all the keyboard shortcuts. To allow for comparability of results we decided to use the restricted TexMo in the control group, so that the same functionality is available to both groups (besides CLS). We do not think that this has a significant impact on the results. TexMo does provide syntax highlighting and redo/undo support. We believe that industrial strength implementation of CLS mechanisms would only improve the already promising results of this experiment.

We established the cross-language relation model for JTrac manually. It relates 9 artifacts containing 51 keys, 87 references, via 87 relations with each other. Our model does not contain false positives, which could have been the case, if it was established automatically.

6 Related Work

Mens et al. [13] identify support for multi-language systems as a major challenge for software evolution. They postulate investigating techniques that are as language independent as possible and providing real-world validation and case studies on industrial software systems as valuable.

Chimera [3] provides hypertext functionality for heterogeneous Software Development Environments (SDE). It allows for the definition of anchors that can be interrelated via links into a hyperweb. Chimera supports navigation along the links. The authors claim that developers in an industrial context appreciate using such links while working. Our paper confirms this belief through a controlled experiment, not provided in [3].

Others agree [11], that multi-language systems pose a real problem in maintenance and evolution. The authors of [11] focus on the process of understanding of such systems, trying to improve it with a graph-based query mechanism to find and understand cross-language relations. Their tool is used in industry, but no empirical data on its effectiveness is available. Our experiment results indicate that these techniques are likely very effective, too.

SourceMiner [6] is an IDE providing advanced software visualizations such as tree maps to aid program understanding. The paper does not present any empirical data. It would be interesting to combine SourceMiner with TexMo to measure if these visualizations improve development of MLSSs beyound the CLS mechanisms studied here.

Since the experimental unit JTrac is based on Wicket, we could have chosen *QWickie* (code.google.com/p/qwickie) as a factor alternative. QWickie is an Eclipse plugin, implementing navigation and renaming support between interrelated HTML and Java files containing Wicket code. We favored TexMo, since we wanted to allow for rerunning the experiment on other experimental units. TexMo is not bound to a particular framework like Wicket.

Visualization mechanisms for relations across heterogeneous concrete syntaxes are studied in the Human-Computer Interaction community. In [21,19] relations across documents in different applications are visualized by links on user request. Visual links are lines crossing application windows. Waldner et al. [21] study if visualization of links between related information in several browser windows is beneficial for understanding scattered information. They run an informal user evaluation with seven participants concluding that *Visual links prevent the user from having to search information manually ... thereby limiting the error probability induced by overseeing information and the effort for the user.* A similar but more formal experiment with 18 participants on visual links is reported by Steinberger et al. [19]. They argue that visual search across different views is a typical task of knowledge workers, which has to be supported by tools. They demonstrate that context preserving visual links are beneficial when searching for interrelated information. Our experiment confirms usefulness of explicit visualization, even though TexMo uses a different visualization scheme.

Chen and coauthors [4] name modern MLSSs, such as Hibernate and Spring "polyglot frameworks". They implement rename refactorings between Java source code and XML configuration files. Unfortunately, they do not provide any experimental data confirming usefulness of such refactorings. Our experiment shows that a substantial amount of developers use such refactorings when provided.

7 Concluding Remarks and Future Work

In this paper we report a controlled experiment evaluating cross-language support mechanisms. The result is, that visualization, static checking, navigation, and refactoring when offered across language boundaries are highly beneficial. CLS mechanisms perceptibly improve effectiveness of developers working on JTrac, a representative MLSS. In the experiment scenario, users of CLS are more effective than the control group with respect to both error rate and productivity (working speed). Furthermore, we show that, within the experiment, CLS mechanisms are actually used by developers and that they improve understanding of complex, unknown multi-language source code.

In future, we plan to replicate our experiment on larger samples to increase confidence in the presented results. Furthermore, we plan to enhance TexMo with more CLS mechanisms, in particular with more elaborate cross-language refactorings, in order to be able to evaluate a broader range of support functions. Ultimately, the present and future experiments will direct our efforts on developing a new generation of development environments.

References

1. The Open Source Developer Report – 2010 Eclipse Community Survey, `eclipse.org/org/press-release/20100604_survey2010.php` (March 2012)
2. Zend Technologies Ltd.: Taking the Pulse of the Developer Community, `static.zend.com/topics/zend-developer-pulse-survey-report-0112-EN.pdf` (February 2012)
3. Anderson, K.M., Taylor, R.N., Whitehead Jr., E.J.: Chimera: Hypermedia for Heterogeneous Software Development Enviroments. ACM Trans. Inf. Syst. 18 (July 2000)
4. Chen, N., Johnson, R.: Toward Refactoring in a Polyglot World: Extending Automated Refactoring Support across Java and XML. In: Proceedings of the 2nd Workshop on Refactoring Tools (2008)
5. Erlikh, L.: Leveraging Legacy System Dollars for E-Business. IT Professional 2 (May 2000)
6. de Figueiredo Carneiro, G., Mendonça, M.G., Magnavita, R.C.: An experimental platform to characterize software comprehension activities supported by visualization. In: ICSE Companion (2009)
7. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F.: Inter-modelling: From Theory to Practice. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 376–391. Springer, Heidelberg (2010)
8. Hessellund, A.: Domain-Specific Multimodeling. Ph.D. thesis, IT University of Copenhagen (2009)
9. Jouault, F., Vanhooff, B., Bruneliere, H., Doux, G., Berbers, Y., Bezivin, J.: Inter-DSL Coordination Support by Combining Megamodeling and Model Weaving. In: Proceedings of the 2010 ACM Symposium on Applied Computing (2010)
10. Juzgado, N.J., Moreno, A.M.: Basics of software engineering experimentation. Kluwer (2001)
11. Kullbach, B., Winter, A., Dahm, P., Ebert, J.: Program Comprehension in Multi-Language Systems. In: Proceedings of the Working Conference on Reverse Engineering, WCRE 1998 (1998)

12. Lientz, B.P., Swanson, E.B., Tompkins, G.E.: Characteristics of Application Software Maintenance. Commun. ACM 21 (June 1978)
13. Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., Jazayeri, M.: Challenges in software evolution. In: Proceedings of the Eighth International Workshop on Principles of Software Evolution, IWPSE 2005 (2005)
14. Paige, R.F., Drivalos, N., Kolovos, D.S., Fernandes, K.J., Power, C., Olsen, G.K., Zschaler, S.: Rigorous Identification and Encoding of Trace-Links in Model-Driven Engineering. Softw. Syst. Model. 10 (October 2011)
15. Pfeiffer, R.H., Wasowski, A.: Taming the Confusion of Languages. In: France, R.B., Kuester, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 312–328. Springer, Heidelberg (2011)
16. Pfeiffer, R.H., Wasowski, A.: TexMo: A Multi-language Development Environment. In: Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., Kolovos, D. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 178–193. Springer, Heidelberg (2012)
17. Salay, R., Mylopoulos, J., Easterbrook, S.: Using Macromodels to Manage Collections of Related Models. In: van Eck, P., Gordijn, J., Wieringa, R. (eds.) CAiSE 2009. LNCS, vol. 5565, pp. 141–155. Springer, Heidelberg (2009)
18. Standish, T.A.: An Essay on Software Reuse. IEEE Trans. Software Eng. (1984)
19. Steinberger, M., Waldner, M., Streit, M., Lex, A., Schmalstieg, D.: Context-Preserving Visual Links. IEEE Transactions on Visualization and Computer Graphics (InfoVis 2011) 17(12) (2011)
20. Strein, D., Lincke, R., Lundberg, J., Löwe, W.: An Extensible Meta-Model for Program Analysis. IEEE Trans. Softw. Eng. 33 (September 2007)
21. Waldner, M., Puff, W., Lex, A., Streit, M., Schmalstieg, D.: Visual Links Across Applications. In: Proc. of Graphics Interface (2010)

Do Professional Developers Benefit from Design Pattern Documentation? A Replication in the Context of Source Code Comprehension

Carmine Gravino[1], Michele Risi[1], Giuseppe Scanniello[2], and Genoveffa Tortora[1]

[1] Facoltá di Scienze MM.FF.NN., Università Degli Studi di Salerno, Italy
{gravino,mrisi,tortora}@unisa.it
[2] Dipartimento di Matematica e Informatica, Università della Basilicata, Italy
giuseppe.scanniello@unibas.it

Abstract. We present the results of a differentiated replication conducted with professional developers to assess whether the presence and the kind of documentation for the solutions or instances of design patterns affect source code comprehension. The participants were divided into three groups and asked to comprehend a chunk of the JHotDraw source code. Depending on the group, each participant was or not provided with the graphical and textual representations of the design pattern instances implemented within that source code. In the case of graphically documented instances, we used UML class diagrams, while textually documented instances are reported as comment in the source code. The results revealed that participants provided with the documentation of the instances achieved a significantly better comprehension than the participants with source code alone. The effect of the kind of documentation is not statistically significant.

Keywords: Design Patterns, Controlled Experiment, Maintenance, Replications, Software Models, Source Code Comprehension.

1 Introduction

Software maintenance is essential in the evolution of software systems and represents one of the most expensive, time consuming, and challenging phases of the whole development process. Maintenance starts after the delivery of the first version of the system and lasts much longer than the initial development process [5], [32]. As shown in the survey by Erlikh [10], the cost needed to perform maintenance operations ranges from 85% to 90% of the total cost of a software project. Whatever is the maintenance operation, the greater part of the cost and effort are due to the comprehension of source code [20]. In particular, Pfleeger and Atlee [23] estimated that up to 60% of software maintenance is spent on comprehension. There are several reasons that make comprehension even more costly and complex, namely the size of a subject software and the available documentation [28].

The availability of software documentation and software models should provide a better support to comprehend source code, so reducing the needed effort and positively affecting the efficiency with which developers perform maintenance operations [2]. For example, Gamma *et al.* [11] assert that developers would benefit from the documentation of design patterns to comprehend source code, so easing its modification. Although there are a number of empirical investigations on design patterns (e.g., [6], [7], [15], [19], [22], [24], [29], [30]), only few evaluations have been conducted on the practical benefits of explicitly reporting design pattern instances[1] in the comprehension of source code [12], [25]. Furthermore, there are no empirical investigations using professional software developers as the participants.

In this paper, we present the results of a differentiated replication[2] conducted with 25 professional software developers to assess whether the presence and the kind of design pattern instances affect source code comprehension. The participants have been working for software companies of the contact network of the authors' research groups. This network was created from research projects and also included companies that: *(i)* host students from the universities of Basilicata and Salerno for external interships or *(ii)* employ people who took a Master or a Bachelor degree at these universities. The participants were divided into three groups and were asked to perform a comprehension task on the source code of JHotDraw. Depending on the group, the participants were provided with source code added or not with design pattern instances either graphically or textually documented. To explicitly and graphically show these instances, we used UML class diagrams [21], while textually documented instances are reported as comment in the source code according to a template.

The work presented here is based on [12] and with respect to it the following new contributions are provided: (1) a differentiated replication with professional developers; (2) a different analysis on the effect of graphically and textually documented design pattern instances; and (3) a deeper discussion on the achieved results and on the possible future directions for this research.

The paper is organized as follows. In Section 2, we highlight the previously conducted controlled experiments and how design pattern instances are documented in these experiments and in the replication presented here. In Section 3, we show the design of this replication, while in Section 4 we show and discuss the results achieved. Related work, remarks, and future work conclude the paper.

2 Documenting Design Pattern Instances

In the design of buildings and towns a design pattern describes a problem which occurs over and over again in our environment, and then describes the core of the

[1] A design pattern includes a name, an intent, a problem, its solution, some example, and so on [11]. In the paper, we focus on the solutions and we will refer to them as design pattern instances.

[2] In this kind of replication, variations in essential aspects (e.g., different kinds of participants) of the original experimental conditions are introduced [3].

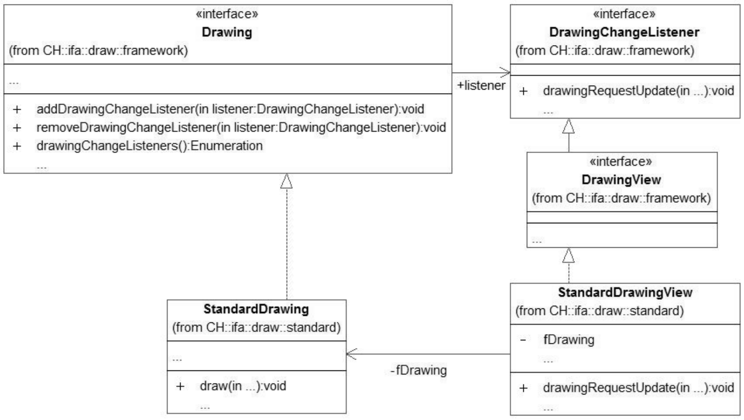
solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice [1]. This definition also holds for design patterns in object-oriented software [11]. The core of both kinds of design patterns is a solution to a problem in a given context. In object-oriented software development a solution is named design pattern instance [13].

There is no a single standard format for documenting design patterns and their instances. Rather, a variety of different formats have been proposed and many of them are based on the UML (e.g., [11], [14]). However, only few studies have been conducted to assess the support provided by explicitly reporting design pattern instances in the execution of maintenance operations and in the comprehension of source code [12], [25]. In [12], for example, we presented a controlled experiment and a replication to assess the benefit of documenting instances with respect to not documenting instances at all. In the first experiment, we considered graphically documented instances by using UML class diagrams, while we considered textually documented ones in the replication.

Each graphical representation of design pattern instances showed a superset of the information provided by the corresponding textual representation. For example, in both the representations the roles each class played within the pattern instances were indicated, while in the textually documented instances the relations among classes (both abstract and concrete) and interfaces were not shown. Figure 1(a) shows an example of graphically documented instance of the Observer design pattern [11] within the source code of JHotDraw. Figure 1(b) shows how the same instance is explicitly reported within the source code as comment. The graphical instance shows more information and then should improve the comprehension of source code. For example, from the class diagram, we can understand that when a drawing (e..g, a container of figures) is changed all the views are updated. More experienced professional software developers could find unnecessary the further information that graphically documented instance provides, because they can directly deduce it from the name of the design pattern and the role of each class and interface.

2.1 Previously Conducted Experiments

The first experiment (UNIBAS in the following) was conducted with 17 Master Students in Computer Science at the University of Basilicata. The participants to the second experiment (a differentiated replication, UNISA in the following) were 24 Master Students in Computer Science at the University of Salerno. The main differences between UNIBAS and UNISA concerned: *(i)* the participants involved and *(ii)* the design used. The participants had a level of experience comparable, but they came from different universities located in different regions. All the involved participants had basic software engineering knowledge. In particular, they knew the basics of requirements engineering, high- and low-level design of object-oriented software systems based on the UML, software development, and software maintenance. They had, however, a limited experience in developing and maintaining nontrivial software systems.


«interface»
Drawing
(from CH::ifa::draw::framework)
...
+ addDrawingChangeListener(in listener:DrawingChangeListener):void
+ removeDrawingChangeListener(in listener:DrawingChangeListener):void
+ drawingChangeListeners():Enumeration
...
+listener
«interface»
DrawingChangeListener
(from CH::ifa::draw::framework)
+ drawingRequestUpdate(in ...):void
...
«interface»
DrawingView
(from CH::ifa::draw::framework)
...
StandardDrawing
(from CH::ifa::draw::standard)
...
+ draw(in ...):void
...
-fDrawing
StandardDrawingView
(from CH::ifa::draw::standard)
- fDrawing
...
+ drawingRequestUpdate(in ...):void
...

(a)

```
/*
 * @(#)DrawingChangeListener.java 5.1
 *
 * Observer Pattern
 * Subject: Drawing
 * ConcreteSubject: StandardDrawing
 * Observer: DrawingChangeListener
 * ConcreteObserver: StandardDrawingView
 *
 */

package CH.ifa.draw.framework;
...
public interface DrawingChangeListener
                        extends EventListener {
...
    /**
     ...
     */
    public void drawingRequestUpdate(...);
}

```

(b)

Fig. 1. A sample of instance for the Observer design pattern: graphically documented (a) and textually documented (b)

Regarding the differences in the design, in UNIBAS the instances were graphically documented by UML class diagrams. For UNISA, the instances were textually documented within the source code in terms of comment. In both the experiments, the participants were asked to comprehend a chunk of JHotDraw v5.1. A single factor experimental design was used in both the experiments. The main factor was represented by the kind of documentation used to explicitly report the instances. This factor was denoted as *Method* and could assume two values: DP (Design Pattern instance documentation) and SC (Source Code alone). For UNIBAS, DP assumed the meaning of graphically documented instances (from here on, GD), while in the replication textually documented (TD). For SC, the source code did not contain any reference to the included instances.

To assess the effect of Method on a comprehension task, we considered three dependent variables that measure: *(i)* source code comprehension (*Comprehension*);

(ii) the time to comprehend source code (*Effort*); and *(iii)* comprehension task efficiency (*Efficiency*). All the three variables are ratio scale measures.

Results. The results of the data analysis on both the experiments provided evidence that participants achieved better Comprehension values when they used the documentation of design pattern instances as complementary information to the source code. The results also indicated that the capability of the participants to correctly recognize design pattern instances impacted more than the type of representation employed to document them. Furthermore, the participants in both the experiments indicated that they trusted the explicitly reported design pattern instances and found them useful.

As far as Efficiency is concerned, we observed that the participants were more efficiently supported in the execution of comprehension tasks when source code was added with the documentation of the design pattern instances. For Effort, the participants to UNIBAS significantly spent less time when using design pattern instances with respect to source code alone. We did not observe any significant difference for UNISA.

3 The Replication with Professionals

The replication was carried out by following the recommendations provided in [17], [31]. The presentation of the replication is based on the guidelines suggested in [16]. For replication purposes, we made available on the Web[3] an experimental package, the raw data, and a technical report with some analyses not reported here for space reasons.

3.1 Goal

Applying the Goal Question Metric (GQM) paradigm [4], the goal of the replication can be defined as: *Analyse* the use of graphical and textual documentation for design pattern instances *for the purpose* of evaluating them *with respect to* the source code comprehension *from the point of view* of project manager, *in the context of* professional software developers.

3.2 Context Selection

We conducted the experiment with 25 Italian software professionals. For each company, we organized a laboratory session. This was the only possible strategy because it is practically impossible to conduct a single experimental session with professionals from different companies. All the laboratory sessions were carried out under controlled conditions to avoid biasing the results, the experiment supervisors were the same in each session.

[3] www.dmi.unisa.it/people/risi/www/DesignPatternInstancesComprehension/

Before the controlled experiment, each professional was asked to fill in a pre-questionnaire. This questionnaire was sent and returned by email. The information gathered was used to classify the participants as junior (with working experience from 1 to 3 years) and senior (with an experience more than 3 years) professional software developers. The junior developers were 10, while 15 were classified as senior. The participants stated that their experience on design pattern development was from *low* to *medium*.

3.3 Selection of the Variables

The dependent variables are: Comprehension, Effort, and Efficiency. To compute the values of Comprehension, we asked each participant to answer a comprehension questionnaire composed of 14 open questions. To quantify the quality of the answers provided and then the source code comprehension achieved, we used an approach based on the information retrieval theory [27]. Therefore, we defined: *(1)* $A_{s,i}$ as the set of string items provided as answer to the question i by the participant s; *(2)* C_i as the correct set of items expected for the question i (i.e., the oracle[4]). For each answer, we can compute:

$$precision_{s,i} = \frac{|A_{s,i} \cap C_i|}{|A_{s,i}|} \quad recall_{s,i} = \frac{|A_{s,i} \cap C_i|}{|C_i|}$$

Precision (i.e., the fraction of items in the answer that are correct) and recall (i.e., the fraction of correct items in the answer) measure the correctness of the answers to a given question and the completeness of the answers, respectively. To get a balance between correctness and completeness, we used a standard aggregated measure based on the combination of precision and recall:

$$F{-}Measure_{s,i} = \frac{2 \cdot precision_{s,i} \cdot recall_{s,i}}{precision_{s,i} + recall_{s,i}}$$

For each participant, the Comprehension value is computed by performing the overall average of the F-Measure values of all the questions. Comprehension assumes values in the interval $[0, 1]$. A value close to 1 means that a participant got a very good comprehension of the source code since he/she answered very well to the questions of the comprehension questionnaire. Conversely, a value close to 0 means that a participant obtained a very bad comprehension.

To determine Effort, we used the time (expressed in minutes) to accomplish the task, which was directly recorded by each participant, while Efficiency was computed dividing Comprehension by Effort. Efficiency is a derived measure that we considered to get a deeper understanding of the contribution provided by the documentation of design pattern instances in the comprehension of source

[4] The names of the classes and methods in the oracle might be different between TD and SC as well as between TD and GD. This is because we removed any possible reference to the design pattern instances from the comment and from the identifiers when the participants used SC and GD.

code. The higher the value of Efficiency, the more efficiently the participant is supported in the accomplishment of the task.

Method is the only independent variable used. It is a nominal variable and assumes values in {SC, TD, GD}. We also grouped the professionals (of TD and GD) into participants, who correctly or incorrectly identified the needed design pattern instances to answer the questions of the comprehension questionnaire: DPCI (i.e., Design Patterns Correctly Identified) and DPnCI (i.e., Design Patterns not Correctly Identified). The data analysis was conducted considering the Comprehension values for the participants in these groups. We performed this further analysis to understand whether different design pattern instances affect source code comprehension. It was possible because we asked the participants to indicate the instances exploited to answer each question.

3.4 Hypotheses Formulation and Experiment Design

We have defined and investigated the following null hypotheses:

Hn0_D_X. The participants who used D design pattern instances (where D can be GD or TD) did not achieve significantly better results in terms of X (where X can be Effort, Comprehension, or Efficiency) than the participants who used source code alone (SC).

Hn1_X. There was not a significant difference with respect to X when participants used GD or TD.

Hn0_D_X is one-tailed because we expected a positive effect of explicitly reporting design pattern instances on the selected dependent variables. Hn1_X is two-tailed because we could not postulate any effect of GD or TD on these variables. The goal of the statistical analysis is to reject the defined null hypotheses and to accept the alternative ones (i.e., Ha0_D_X and Ha1_X), which can be easily derived (e.g., Ha1_X: There was a significant difference with respect to X when participants used GD or TD).

We used the one factor with three treatments design [31]. The participant working experience (i.e., the amount of years as professional developers) was the blocking factor. Then, we equally distributed junior and senior experienced professionals among the three groups: GD, TD, and SC. We assigned 9 participants (4 juniors and 5 seniors) to GD and 8 (3 juniors and 5 seniors) to TD and SC, respectively. The use of a different experiment design (such as the within-participant counterbalanced design) with non-trivial experimental objects (as in this experiment) may bias the results introducing a factor difficult to be controlled, i.e., the mental fatigue.

3.5 Experimental Tasks

We asked the participants to perform the following tasks:

Comprehension Task. The participants were asked to fill in the comprehension questionnaire, whose questions were divided into three groups to let

Q2. Indicating the class/es and the method/s in charge of creating, drawing, and updating the instances of the class Figure?				
How much do you trust your answer$^+$?				
☐ Unsure	☐ Not sure enough	☐ Sure Enough	☐ Sure	☐ Very Sure
How do you assess the question$^+$?				
☐ Very difficult	☐ Difficult	☐ On average	☐ Simple	☐ Very Simple
What is the source of information used to answer the question$^+$?				
☐ Previous Knowledge (PK)		☐ Internet (I)	☐ Source Code (SC)	
$^+$ Mark only one answer				

Fig. 2. A question example from the comprehension questionnaire

participants take a break if needed when passing from a group of questions to the next one. This choice was taken for reducing fatigue effect biases. We defined the questions to assess several aspects related to the comprehension of the source code. All the questions (except Q11) were formulated using a similar form/schema. Figure 2 shows a sample question for SC.

We also collected data on the source of information the participants used to answer each question. In particular, we asked the participants who accomplished the task with source code added with documented design pattern instances (i.e., GD and TD) to specify for each question whether the answer was derived using: (DPI) design pattern instances, (PK) previous knowledge, (I) Internet, or source code (SC). If the participants specified DPI, they were also asked to indicate the instances used. The participants who accomplished the task using the source code alone chose among: previous knowledge, Internet, and source code. This was the only difference introduced in the comprehension questionnaires used in the three treatments. Whatever was the treatment, we asked the participant to indicate also the confidence level (e.g., *Sure*) and the degree of complexity (e.g., *Difficult*) for each question answered (see Figure 2). The analysis on this further information is not reported for space constraint, but it is available in the technical report.

The question in Figure 2 expected as the correct answer the following set of items: *CreationTool*, *ArrayFigure*, *StandardDrawingView*, *createFigure()*, *draw()*, and *drawingRequestUpdate()*. The correct answer could be derived by the following instances of design patterns: Prototype, Composite, and Observer (see Figure 1). In particular, the Prototype instance was useful because it was in charge of managing the creation of a template figure, while the Composite drew each base element of an object Figure. The Observer instance was in charge of managing the paint and/or the repaint of an object Figure. If a participant provided CreationTool, ArrayFigure, and *drawingChangeListeners()* as the answer, the value for Comprehension is 0.44. It results from 0.66 and 0.33 as the precision and recall values, respectively. In fact, the number of correct items provided is 2 (CreationTool and ArrayFigure), while 3 is the total number of items provided and 6 is the number of correct items expected.

Post-experiment Task. We asked the participants to fill in a post-experiment survey questionnaire. The goal of this questionnaire was to obtain feedback about the participants' perceptions of the experiment execution. For space reasons the results of the survey are not reported in the paper. Details can be found in the technical report.

3.6 Experimental Procedure

The participants first attended an introductory lesson in which the supervisors presented detailed instructions on the experiment. The supervisors highlighted the goal of the experiment without providing details on the experimental hypotheses. No time limit to perform the task was imposed. We organized individual experimental sessions for professional developers working in the same business unit. The participants were not allowed to communicate each other.

To perform the comprehension task, the participants were provided with laptops having the same hardware configuration (i.e., equipped with a 1.5 GHz Intel Centrino with 1.5 GB of RAM, a 60GB Hard Disk and Windows XP Professional SP3 as operating system). To surf source code, we installed on each laptop a general purpose and well known text editor (i.e., UltraEdit[5]). We also provided the participant with an Internet connection to be used while performing the comprehension task.

We asked the participants to use the following experimental procedure for each group of questions within the comprehension questionnaire: *(i)* specifying name and start-time; *(ii)* answering the questions using the source code (without executing it) and the explicitly reported design pattern instances if present; and *(iii)* marking the end-time. We did not suggest any approach to comprehend source code. We only discouraged to read all the code.

We provided the participants with a paper copy of the following experimental material: *(i)* the comprehension questionnaire and *(ii)* a post-experiment survey questionnaire. The participants in GD were also provided with the source code (without any references to the design pattern instances) and the paper copy of a document where each design pattern instance was graphically reported (see Figure 1(a)). The participants that used TD were provided with source code that included the references to the design pattern instances in the comment (see Figure 1(b)). For SC, the participants were provided with source code without any kind of documentation to the instances implemented.

3.7 Analysis Procedure

To perform the data analysis, we carried out the following steps and used the R environment[6] for statistical computing:

1. We undertook the descriptive statistics of the measures of the dependent variables, i.e., Effort, Comprehension, and Efficiency (see Section 4.1).

[5] www.ultraedit.com

[6] www.r-project.org

2. To test the null hypotheses, we adopted non-parametric tests due to the sample size and mostly the non-normality of the data. In particular, we used the Mann-Whitney test [9] due to the design of the experiments (only unpaired analyses were possible) and to its robustness [31] (see Section 4.2). In all the statistical tests, we decided (as custom) to accept a probability of 5% of committing Type-I-error [31]. The chosen statistical test allows the presence of a significant difference between independent groups to be verified, but it does not provide any information about this difference [18]. Therefore, we used the Cohen's d [8] effect size to obtain the standardized difference between two groups that can be considered negligible for $|d| < 0.2$, small for $0.2 \leq |d| < 0.5$, medium for $0.5 \leq |d| < 0.8$, and large for $|d| \geq 0.8$. We also analyzed the statistical power for each test performed. Statistical power is the probability that the test will reject a null hypothesis when it is actually false (i.e., the probability of not committing a Type II error, or making a false negative decision). The highest value is 1, while 0 is the lowest. The higher the statistical power value, the higher is the probability to reject a null hypothesis when it is actually false.

3.8 Differences and Similarities

The experience gained in the previously executed experiments [12] suggested some variations in the experiment presented here. The variations have been introduced to mitigate as many threats to validity as possible and to improve the material and the data analysis:

Participants. They are professional developers and are more experienced than the participants to UNIBAS and UNISA. This variation allowed reducing external validity threats.

Experiment Design. We used the one factor with three treatments design. The participants were divided into three groups. The control group was the group of participants in SC. Differently, we have here two treatment groups: GD and TD. As for UNIBAS and UNISA, the independent variable is Method (i.e., the main factor), which is a nominal variable that assumes three possible values: SC, TD, and GD.

Group Composition. We used the information gathered in a pre-questionnaire to equally distribute high and low experienced professionals among the three groups. The professional experience is the blocked factor for the experiment.

Data Analysis. Bearing in mind the new adopted design, we were able to better analyze the effect of the documentation type on source code comprehension.

Training Session. The professionals did not carried out a training session on tasks similar to the one used in the experiment. Two were the reasons: (1) they had an adequate experience in performing maintenance operations on source code implemented by others; (2) time and logistic constraints did not make possible the execution of a training session (the use of professionals might cause this kind of concern).

Experimental Procedure. We allowed the participants to find information on the Web useful to accomplish that task. Professional developers usually exploit this medium as support for their daily work activities.

Comprehension Questionnaire. We removed mistakes and some sources of possible confusion.

We preserved some design choices in the replication presented here:

Dependent Variables. They are well known and widely employed in the Empirical Software Engineering community (e.g., [26]). These variables well summarize the aspects we were interested in investigating. Another by-product of this choice was in the evaluation of source code comprehension that could be computed in a repeatable manner, so reducing construct validity threats.

Experimental Object. We used a chunk (i.e., vertical slice) of JHotDraw v5.1 that included: *(i)* a nontrivial number of design pattern instances and *(ii)* well-known and widely adopted design patterns. In the selection process, we have also taken into account a trade-off between the complexity of the implemented functionality and the effort to comprehend it (about 3 hours for low experienced participants). To mitigate external validity threats, we tried as much as possible to define a realistic comprehension task.
We translated the comments from English into Italian to avoid biasing the results because different participants may have different familiarity with English. Further, we removed any possible reference to the design pattern instances from the comment and from the identifiers (e.g. *CompositeFigure* was named as ArrayFigure) when the participants performed the comprehension task with the source code alone and the graphically documented instances. The source code was constituted of 1326 Lines of Code, 26 Classes, and 823 Lines of Comments. One of the authors manually detected the design pattern instances in source code. To this end, he also used the documentation of JHotDraw and the public dataset PMARt[7]. The following instances of design patterns were present in the source code used: State, Adapter, Strategy, Decorator, Composite, Observer, Command, Template Method, and Prototype. For the State design pattern were two instances. These instances are graphically represented (as much as possible) as in [11] and textually represented as shown in Section 2. We used JHotDraw because it is intentionally designed to have very clear implementations of well-known design patterns. Therefore, it can be considered a good experimental object.

4 Results

4.1 Descriptive Statistics and Exploratory Analysis

Table 1 shows some descriptive statistics (i.e., median, mean, and standard deviation) of Effort, Comprehension, and Efficiency grouped by Method. These

[7] `www.ptidej.net/downloads/pmart/`

Table 1. Descriptive statistics for GD, TD, and SC

Dependent Variable	GD			TD			SC		
	Mean	Median	St. Dev.	Mean	Median	St. Dev.	Mean	Median	St. Dev.
Effort	132	142.2	31.17	139	136.4	37.00	151	147.4	42.81
Comprehension	50.91	51.08	10.32	53.43	53.49	7.26	40.56	39.97	9.63
Efficiency	0.37	0.37	0.11	0.37	0.42	0.14	0.29	0.31	0.14

Table 2. Descriptive statistics for GD grouped by DPCI and DPnCI

Dependent Variable	DPCI			DPnCI		
	Mean	Median	St. Dev.	Mean	Median	St. Dev.
Effort	9	10.36	3.93	8.00	9.13	3.67
Comprehension	80	73	27.70	47	39.13	36.88
Efficiency	8.17	8.24	4.60	4.4	5.28	6.18

statistics show that the participants using source code alone (SC) spent on average more time (151 minutes) than the participants using documented design pattern instances (132 and 139 minutes for GD and TD, respectively). On average the participants who used GD and TD achieved a better comprehension of source code (50.91 and 53.43, respectively) than those who used SC (40.56). We achieved similar results for Efficiency.

Table 2 shows descriptive statistics (i.e., median, mean, and standard deviation) of Effort, Comprehension, and Efficiency for GD grouping observations by DPCI and DPnCI. Similarly, Table 3 reports descriptive statistics for TD. These descriptive statistics suggest that the participants who correctly recognized the design pattern instances (both in TD and GD), to answer a given question, achieved on average better Comprehension and Efficiency values than the participants who did not correctly recognized them.

4.2 Hypotheses Testing

The results of the Mann-Whitney test are summarized in Table 4, together with the Cohens' d effect size and the statistical power values. The results show that Hn0_GD_Comprehension and Hn0_TD_Comprehension can be rejected (p-values are 0.033 and 0.009, respectively) with a large effect size and high statistical power. Thus, the participants who used the documentation of design pattern instances significantly better comprehended source code than those provided with source code alone. Hn0_D_Effort and Hn0_D_Efficiency cannot be rejected.

Table 3. Descriptive statistics for TD grouped by DPCI and DPnCI

Dependent Variable	DPCI			DPnCI		
	Mean	Median	St. Dev.	Mean	Median	St. Dev.
Effort	10	9.61	3.41	10	9.97	4.93
Comprehension	73	67.06	30.88	45	38.50	36.13
Efficiency	7.69	8.46	5.91	3.94	5.76	8.08

Table 4. Results for Hn0_D_X

Documentation	Hypothesis	Influence (p-value)	Effect Size	Statistical Power
GD	_Effort	No (0.337)	-0.137 (negligible)	0.075
	_Comprehension	**Yes** (0.033)	1.113 (large)	0.966
	_Efficiency	No (0.135)	0.527 (medium)	0.287
TD	_Effort	No (0.282)	-0.273 (small)	0.115
	_Comprehension	**Yes** (0.009)	1.586 (large)	0.875
	_Efficiency	No (0.056)	0.806 (large)	0.354

Table 5. Results for Hn1_X

Hypothesis	Influence (p-value)	Effect Size	Statistical Power
_Effort	No (0.810)	0.170 (negligible)	0.045
_Comprehension	No (0.665)	-0.271 (small)	0.065
_Efficiency	No (0.664)	-0.375 (small)	0.075

Table 5 shows the data analysis results for the null hypotheses Hn1_X. In particular, the results of the Mann-Whitney test indicated that the null hypotheses cannot be rejected. Therefore, for all the dependent variables the difference between the participants who used graphically documented and textually documented design pattern instances is not statistical significant.

4.3 Further Analyses - Analysis by Question

The DPCI participants in GD achieved significant better results in terms of Comprehension than the DPnCI participants on the questions Q1 (p-value = 0.043) and Q5 (p-value 0.048). This result suggested that the design patterns that better supported the participants in the execution of comprehension tasks were: Prototype, Composite, Observer, and Template Method. Regarding TD, the DPCI participants achieved significantly better results in terms of Comprehension on the question Q3 (p-value = 0.032). This indicated that only Composite and Observer design patterns better supported the participants in the source code comprehension. This result is interesting from the researcher's point of view because it seems that the interaction among the instances of different design patterns affects source code comprehension. Further and special conceived investigations are, however, needed because our primary goal here was: to assess whether the presence and the kind of documentation for design pattern instances affect source code comprehension. In this further analysis, we did not considered Q11 because it was formulated differently from the others.

4.4 Discussion

The results of this replication have largely confirmed those achieved in the previous experiments [12]. The software professionals achieved significant better Comprehension values when they received the documentation of the design pattern instances as a complementary information to comprehend source code.

In particular, the mean improvement achieved with the design pattern instances graphically documented was 25.5%, while it was 31.9% for those textually documented. The participants who used textually documented design pattern instances obtained on overage slightly better results in terms of Comprehension with respect to the participants who exploited graphically documented instances. A plausible justification for this result is that professionals were more comfortable with source code and the information provided in the comment (i.e., explicitly reported instances) was more than adequate to comprehend the code.

The time to perform the comprehension task did not increase with respect to the use of the source code alone. This result could be considered as unexpected because more documents/information to read and interpret could need more effort to execute the task. This should be even more evident for design pattern instances that were graphically documented. Then, these results suggest that the additional information provided by the documented design pattern instances reduced the effort to analyze the source code.

For GD, the descriptive statistics reported in Table 2 indicate that the average Comprehension value achieved by the participants when they correctly identified the instances to answer the question is 70.2% greater than the average value obtained when the instances were not correctly recognized. For TD, this difference is 62.2%. This finding is interesting from both the researcher and the project manager points of view. In fact, it seems relevant to help developers in recognizing pattern instances more than the kind of documentation used.

Regarding the source of information, we observed that the participants employing GD largely indicated as first source of information the design patterns. Differently, the participants employing TD indicated as first source of information the source code, while design pattern instances were classified as the second one. This slight difference in the results achieved on GD and TD could be due to the fact that the design pattern instances are documented in the source code in the latter case. Then, the participants considered the documented instances as an integral part of the code. Although this difference, the results show that the participants trusted the design pattern instances explicitly reported. It is also worth noting that the Internet was almost never used: 8 participants (6 used TD and 2 GD) stated that they on average used the Internet on 2 out of 14 questions. Therefore, the Internet was not considered a relevant source of information.

4.5 Threats to Validity

Conclusion validity concerns issues that affect the ability of drawing a correct conclusion. In our study, we used proper statistical tests. In particular, a non-parametric test (i.e., Mann-Whitney test for unpaired analyses) was used to statistically reject the null hypotheses.

Internal validity threats are mitigated by the design of the experiment. Each group of participants worked only on one task, with or without the design patterns instances. Fatigue is another possible threat for internal validity. We mitigate the fatigue effect allowing the participants to take a break. Another possible threat concerns the use of the Internet to exchange information. We prevented that monitoring the participants, while performing the task.

Construct validity may be influenced by the metrics used and social threats. The exploited metrics are widely used with purposes similarly to ours [26]. Regarding Comprehension, one of the authors not involved in the definition of the task built the questionnaire. A further threat could be related to the modification of the identifiers in the code.

External validity concerns the generalization of the results. Possible threats are related to the complexity of the comprehension task and the choice of participants. Regarding the first point, we selected a part of an open software system large enough to be considered not excessively easy. As for the participants, they are Italian professional junior/senior software developers. Moreover Southern and Central Italy are over-represented with respect to Northern Italy.

5 Related Work and Conclusion

Only few studies have been conducted to assess the support that design pattern instances provide in the execution of maintenance tasks and in the comprehension of source code [12], [25]. Prechelt *et al.* [25] studied whether design pattern instances explicitly and textually documented in the source code (through comment) improve the maintainers' performance in performing comprehension tasks with respect to a well-commented program without explicit reference to design patterns. The study involved 74 German graduate students and 22 USA undergraduate students, who performed maintenance operations on Java and C++ code, respectively. The data analysis revealed that maintenance tasks supported by explicitly documented design pattern instances were completed faster or with fewer errors. The most remarkable difference with our work is that we additionally analyze the effect of pattern instances graphically documented. Furthermore, we used professionals and the used experimental object is larger and more complex (20149 LOCs including comments with respect to 360 and 560).

The results achieved in our experiment and those achieved in [12] and [25] give strength to the usefulness of exploiting explicitly documented design pattern instances in the execution of comprehension tasks. Therefore, it seems worthily to document design pattern instances. This result, however, opens a managerial dilemma: Are the additional effort and cost, due to create and maintain the documentation of design pattern instances, adequately paid back by an improved comprehension of source code? Indeed, from a manager point of view, the adoption of graphically and textually documentation, as means to represent design pattern instances, should take into account the costs it will introduce. Furthermore, what is the less expensive method for representing instances? These points represent future directions for our work.

Another remarkable result of our experiment is: design pattern based development can increase the source code comprehension only in case the design pattern instances are correctly recognized in the source code. This open an interesting future direction for the research. In particular, it would be worth investigating: *(i)* the issues that led to certain patterns being better comprehended and recognized than others and *(ii)* new methods for representing design pattern instances, so easing their recognition. It will be also worth investigating whether the source code comprehension improves when graphically documented design pattern instances are added with sequence diagrams.

Acknowledgments. We thank the participants to the experiment and Paolo Mondelli for his support.

References

1. Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., Angel, S.: A Pattern Language - Towns, Buildings, Construction. Oxford University Press (1977)
2. Arisholm, E., Briand, L.C., Hove, S.E., Labiche, Y.: The impact of UML documentation on software maintenance: An experimental evaluation. IEEE Trans. Softw. Eng. 32(6), 365–381 (2006)
3. Basili, V., Shull, F., Lanubile, F.: Building knowledge through families of experiments. IEEE Trans. Softw. Eng. 25(4), 456–473 (1999)
4. Basili, V.R., Rombach, H.D.: The TAME project: Towards improvement-oriented software environments. IEEE Trans. Software Eng. 14(6), 758–773 (1988)
5. Bennett, K.H., Rajlich, V.T.: Software maintenance and evolution: a roadmap. In: Procs. of the Conference on the Future of Software Engineering, ICSE 2000, pp. 73–87. ACM, New York (2000)
6. Bieman, J., Straw, G., Wang, H., Munger, P., Alexander, R.: Design patterns and change proneness: an examination of five evolving systems. In: Procs. of Software Metrics Symposium, pp. 40–49. IEEE CS (2003)
7. Cepeda Porras, G., Guéhéneuc, Y.-G.: An empirical study on the efficiency of different design pattern representations in UML class diagrams. Empirical Softw. Eng. 15(5), 493–522 (2010)
8. Cohen, J.: Statistical power analysis for the behavioral sciences, 2nd edn. Lawrence Earlbaum Associates, Hillsdale (1988)
9. Conover, W.J.: Practical Nonparametric Statistics, 3rd edn. Wiley (1998)
10. Erlikh, L.: Leveraging legacy system dollars for e-business. IT Professional 2, 17–23 (2000)
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object Oriented Software. Addison-Wesley (1995)
12. Gravino, C., Risi, M., Scanniello, G., Tortora, G.: Does the documentation of design pattern instances impact on source code comprehension? Results from two controlled experiments. In: Procs. of the Working Conference on Reverse Engineering, pp. 67–76. IEEE CS (2011)
13. Guéhéneuc, Y.-G., Antoniol, G.: Demima: A multilayered approach for design pattern identification. IEEE Trans. Softw. Eng. 34(5), 667–684 (2008)
14. Heer, J., Agrawala, M.: Software design patterns for information visualization. IEEE Trans. Visualization & Comp. Graphics (Proc. InfoVis) 12, 853–860 (2006)
15. Jeanmart, S., Guéhéneuc, Y.-G., Sahraoui, H., Habra, N.: Impact of the Visitor Pattern on program comprehension and maintenance. In: Procs. of the Symposium on Empirical Software Engineering and Measurement, pp. 69–78. IEEE CS (2009)

16. Jedlitschka, A., Ciolkowski, M., Pfahl, D.: Reporting Experiments in Software Engineering. In: Shull, F., Singer, J., Sjoberg, D. (eds.) Guide to Advanced Empirical Software Engineering, pp. 201–228. Springer, London (2008)
17. Juristo, N., Moreno, A.: Basics of Software Engineering Experimentation. Kluwer Academic Publishers (2001)
18. Kampenes, V., Dyba, T., Hannay, J., Sjoberg, I.: A systematic review of effect size in software engineering experiments. Information and Software Technology 49(11-12), 1073–1086
19. Khomh, F., Guéhéneuc, Y.-G.: Do design patterns impact software quality positively? In: Procs. of Conference on Software Engineering and Maintenance, pp. 274–278 (2008)
20. Mayrhauser, A.V.: Program comprehension during software maintenance and evolution. IEEE Computer 28, 44–55 (1995)
21. OMG. Unified modeling language (UML) specification, version 2.0. Technical report, Object Management Group (July 2005)
22. Penta, M.D., Cerulo, L., Guéhéneuc, Y.-G., Antoniol, G.: An empirical study of the relationships between design pattern roles and class change proneness. In: Procs. of the International Conference on Software Maintenance, pp. 217–226. IEEE CS (2008)
23. Pfleeger, S., Atlee, J.: Software engineering - theory and practice, 3rd edn. Ellis Horwood (2006)
24. Prechelt, L., Unger, B., Tichy, W.F., Brössler, P., Votta, L.G.: A controlled experiment in maintenance comparing design patterns to simpler solutions. IEEE Trans. Software Eng. 27(12), 1134–1144 (2001)
25. Prechelt, L., Unger-Lamprecht, B., Philippsen, M., Tichy, W.: Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. IEEE Trans. Softw. Eng. 28(6), 595–606 (2002)
26. Ricca, F., Penta, M.D., Torchiano, M., Tonella, P., Ceccato, M.: How developers' experience and ability influence web application comprehension tasks supported by uml stereotypes: A series of four experiments. IEEE Trans. Software Eng. 36(1), 96–118 (2010)
27. Salton, G., McGill, M.J.: Introduction to Modern Information Retrieval. McGraw Hill, New York (1983)
28. Selfridge, P., Waters, R., Chikofsky, E.: Challenges to the field of reverse engineering. In: Proc. of the Working Conference on Reverse Engineering. IEEE CS (1993)
29. Vokac, M.: Defect frequency and design patterns: An empirical study of industrial code. IEEE Trans. Software Eng. 30(12), 904–917 (2004)
30. Vokác, M., Tichy, W.F., Sjøberg, D.I.K., Arisholm, E., Aldrin, M.: A controlled experiment comparing the maintainability of programs designed with and without design patterns-a replication in a real programming environment. Emp. Softw. Eng. 9(3), 149–195 (2004)
31. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., Wesslén, A.: Experimentation in Software Engineering - An Introduction. Kluwer (2000)
32. Zelkowitz, M.V., Shaw, A.C., Gannon, J.D.: Principles of software engineering and design. Prentice-Hall (1979)

Incremental Consistency Checking for Complex Design Rules and Larger Model Changes

Alexander Reder and Alexander Egyed

Johannes Kepler University, Linz, Austria
`alexander.{reder,egyed}@jku.at`

Abstract. Advances in consistency checking in model-based software development made it possible to detect errors in real-time. However, existing approaches assume that changes come in small quantities and design rules are generally small in scope. Yet activities such as model transformation, re-factoring, model merging, or repairs may cause larger model changes and hence cause performance problems during consistency checking. The goal of this work is to increase the performance of re-validating design rules. This work proposes an automated and tool supported approach that re-validates the affected parts of a design rule only. It was empirical evaluated on 19 design rules and 30 small to large design models and the evaluation shows that the approach improves the computational cost of consistency checking with the gains increasing with the size and complexity of design rules.

Keywords: consistency checking, performance, incremental checking.

1 Introduction

Errors in design models range from basic well-formedness problems (e. g., syntactic violations) to more advanced, multi-view inconsistencies. While designers may be willing to tolerate these errors [10,1], the designer should nonetheless be aware of their existence. Fortunately, recent progress on consistency management has demonstrated that modeling tools can be made to detect errors in design models in real time while retaining the free customizability of design rules. Existing approaches, such as the Model/Analyzer [6], re-validate design rules only if they are affected by model changes. Empirical evaluations have shown that such approaches are very fast: they can validate the impact of a design change in milliseconds in average with the performance being unaffected by the model size.

However, the performance of state-of-the-art incremental consistency checkers decreases with 1) the quantity of model changes, 2) the complexity of design rules, and 3) the number of design rules. For example, existing approaches assume that models change in small increments only (e. g., a class is renamed, a new message is added to a sequence diagram). Most model changes are indeed small. Unfortunately, there are a range of quite common modeling activities that cause larger model changes. Examples are model transformations [13], re-factoring [19], model branching or merging (as in subversion) [18], and model repairs [7,9,15]. These activities may be arbitrary complex and pose a challenge to incremental consistency checking because the increment becomes too large to handle it instantaneously. This problem is aggravated with the complexity of design rules. The more complex a design rule the more likely it is affected by a model change. And the problem is even further aggravated with an increasing number of design rules. The more design rules there are, the more likely are model changes to affect multiple design rules. Combined, they strongly impact the performance of incremental consistency checking.

This paper proposes a novel approach for improving the performance of incremental consistency checking. The basic idea is to not validate design rules in their entirety but to focus on the parts that are affected by model changes. Whether a part of a design rule is affected by a change is determined fully automatically based on observations of the design rule's validation which is stored as a *validation tree*. The approach does require an additional memory overhead for storing the validation tree. Complete validation trees can be voluminous but a second novel contribution is in the reduction of the validation tree to those parts of a design rule validation whose change can impact the validation result as a whole. For example, if $a = true$ and $b = false$ in the conjunction $a \wedge b$ then a change to a cannot affect the result of the conjunction and can be cut from the tree.

We evaluated our approach on 19 design rules, 30 industrial design models (approximately 130,000 elements), and roughly 1,500 random model changes. We will demonstrate through empirical evaluations that our approach achieves up to 20-fold performance for 19 designs rules we analyzed – the more complex the design rule, the larger the performance gains. While the scalability of the proposed approach is still linearly dependent on the quantity of model changes and the number of design rules, the performance gain implies that much larger model changes or quantities of design rules can be handled instantaneously than was possible to date. The memory cost is in average only 2-fold more expensive compared to state-of-the-art and increases linear with the size of the model. Our approach is fully automated, tool supported, and integrated with the modeling tool IBM Rational Software Architect. The implementation supports OCL as the constraint language and UML as the modeling language, but the approach is designed to be applicable to arbitrary modeling languages and their corresponding constraint languages.

The remainder of this paper is structured as follows. Section 2 defines the basic terms and a running example that are used in this paper. In Section 3 the main principles of our approach are shown. Section 4 shows the evaluation of our approach and Section 5 explains the threats of validity of the evaluations. An overview of the existing work on this topic is given in Section 6 and, finally, Section 7 concludes the work and gives an outlook about future work planned.

2 Definitions and Example

2.1 Basic Definitions

Definition 1. *A **model** represents the main aspects of a software project that must be implemented. It consists of **model elements** that contain **properties**, e. g., a name or a reference to other model elements. A **design rule** defines requirements that must be fulfilled in the model. A violation of a design rule causes an inconsistency in the model.*

The requirement of the design rule is expressed as a ***condition*** *that validates to true (consistent) or false (inconsistent). A design rule is written for a specific* ***context*** *that can be a single model element (the design rule will be validated once) or a type of model element (the design will be validated for each instance of this model element type in the model). Each validation can cause a separate inconsistency.*

$$\begin{aligned} Design\,Rule &:= \langle context,\, condition\rangle \\ condition\ &:\ context \rightarrow \{true,\, false\} \end{aligned}$$

Definition 2. *A design rule condition consist of a set of hierarchical ordered expressions where each expression consists of an operation (o), a set of 0 to * arguments (α) and a validation result (σ). The arguments of an* ***expression*** *(ϵ) are expressions itself and they are tree based ordered, i. e, each expression has exactly one parent and is in a set of arguments of an other expression except the root expression (ϵ_0).*

$$\begin{aligned} condition &:= \bigcup_{i=0}^{n} \epsilon_i | \begin{cases} \exists i,j : \epsilon_j \in \epsilon_i.\alpha & \textit{if } j > 0,\, i \neq j \\ \nexists i,j : \epsilon_j \in \epsilon_i.\alpha & \textit{if j=0, i}{\neq}\textit{j} \end{cases} \\ \epsilon &:= \langle o,\, \alpha,\, \sigma\rangle \end{aligned}$$

2.2 Running Example

Figure 1 introduces a small illustration to accompany the discussion in the paper. The example depicts an UML model containing two diagrams, a class and sequence diagram. The given model represents an early design-time snapshot of a video-on-demand (VOD) system. The class diagram (left) represents the structure of the VOD system: a 'User' that controls the system and watches videos, a 'Display' used for visualizing movies and receiving user input and a 'Streamer' for downloading and decoding movie streams. The sequence diagram (right) describes the process of selecting a movie and playing it. Since a sequence diagram contains interactions among instances of classes (objects), the illustration depicts a particular user invoking 'select' (a message) on the 'd' lifeline of type 'Display' which then invokes 'connect' on the 's' lifeline of type 'Streamer'. The movie starts playing once the 'play' message is issued which is followed by 'stream' and successive 'draw' messages.

$$\begin{aligned} &Message\, m : \\ &\qquad \left.\begin{aligned} &(\exists l_1 \in m.receiveEvent.covered, \\ &\quad l_2 \in m.sendEvent.covered : \\ &\exists a \in l_2.represents.type.ownedAttribute : \\ &a \neq null \Rightarrow a.type = l_1.represents.type) \end{aligned}\right\} (1.1) \\ &\qquad\qquad\qquad\qquad \wedge \\ &\qquad \left.\begin{aligned} &(\forall l \in m.receiveEvent.covered : \\ &\exists o \in l.represents.type.ownedOperation : \\ &o.name = m.name) \end{aligned}\right\} (1.2) \end{aligned} \tag{1}$$

Design Rule (1) discusses two conditions the model must satisfy: 1) whether a given message in a sequence diagram matches the direction of the class association (1.1), and

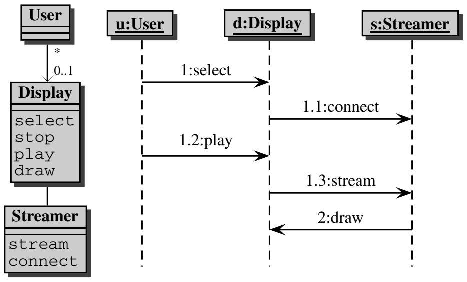


Fig. 1. UML Class and Sequence Diagram of a Video on Demand System

2) whether the given message has a same-named operation (1.2). The two conditions are expressed in one design rule and linked together by a conjunction ($\wedge$). This linking into more complex design rules is common, for example, to avoid unnecessary validations. As such, the search for a matching operation name (1.2) is useful only if the operation's class is identified correctly by the association (1.1).

Typically, rules are written from the perspective of a context – a type of model element. The context for Design Rule (1) is the UML type 'Message' (see $Message\, m$ at the very top) and the rule has to be validated separately for each message in the sequence diagram in Figure 1. There are thus five validations of that rule necessary in the illustration. Each evaluation validates the correctness of its message only.

If, for example, the design rule validates the message 'connect' then, first, both ends of the message are accessed through the receive event ($l_1 \in m.receiveEvent.covered$) and the send event ($l_2 \in m.sendEvent.covered$). As it is possible in UML that a message is assigned to more than one lifeline, the design rule iterates via an existential quantifier ($\exists l_1, l_2 \ldots$) over all returned lifelines which are assigned to the variables l_1 and l_2. For message 'connect' l_1 is instantiated with the lifeline 's' of type 'Streamer' and l_2 with the lifeline 'd' of type 'Display'. Next, the owned attributes (the association ends are expressed as attributes) of the senders lifeline type ($l_2.represents.type.ownedAttribute$) are compared to those of the receiver ($a.type = l_1.represents.type$). If one is found then there must be an association that connects sender type and receiver type. The second part of Design Rule (1) accesses the receiver lifeline – lifeline 's' in this example. All types (universal quantifier $\forall$) of the lifeline must include an operation that is named after the message name. The 's' lifeline is an instance of class 'Streamer' that includes operations ('ownedOperation' property) such as 'connect' and 'stream'. The existential quantifier then validates whether at least one operation matches the name of the message (='connect'). Thus, both conditions of Design Rule (1) are satisfied and the condition validates to true (=consistent).

2.3 Problem Description

Current state-of-the-art requires the re-validation of the entire design rule [6] if a model change affects it. This is computationally increasingly expensive with the complexity of the design rule or the number of model changes, thus these approaches scale only for small quantities of model changes (individual changes) and comparative small design rules. A solution explored in [3] is thus to describe design rules from the perspective of different model changes. However, this requires manual overhead and introduces errors if done incorrectly. Another solution would be to split up the design rule into smaller parts. However, doing so increases the number of design rules but would not cause any performance and memory advantages. This problem is aggravated by the quantity of model changes (e. g., as in model re-factoring, branching, merging, repair). To illustrate this, consider the case of a repair. Our previous work [9] demonstrated that between ten to twenty kinds of changes can resolve a typical inconsistency. This number seems small enough. However, for computing the effects of such repairs ([5] referred to them as side effects), permutations of these ten kinds of choices need to be explored where each permutations requires (incremental) consistency checking. The number of possible repair alternatives increases exponentially.

3 Model/Analyzer Approach

This section introduces an approach that improves incremental consistency checking. Our empirical evaluation shows a reduction up to 20-fold (average 10-fold) in context of 19 design rules we analyzed. Our approach automatically records the run-time behavior of design rules to reason about which parts of the design rule validation are affected by a model change. In the following, we demonstrate how to capture the validation of a design rule in form of a validation tree and how to identify which part of the validation tree is affected by a model change.

3.1 Principles

Incremental consistency checking builds a scope for each design rule – or in case of Model/Analyzer for each design rule validation. If a model element changes then all those design rule validations need re-validation that included the changed element in their scope. Take, for example, a simple conjunction $a \wedge b$. In fact, Design Rule (1) is a conjunction where a checks for the message direction and b checks for the method declaration. During the initial validation of this conjunction, the consistency checker will first validate a and if a is true then b will be validated also. If a is false then b need not be validated because the result does not depend on b. The validation of the conjunction $a \wedge b$ results in either true (=consistent) or false (=inconsistent).

A model change only then affects the validation result of conjunction $a \wedge b$ if either a or b changes. Clearly, if neither a nor b change then the validation result cannot be affected. However, not all changes to a or b affect the validation result. For example if $a = true$, $b = false$ and a changes then this change does not affect the result of the conjunction ($a \wedge b$ was false because of b and for as long as b remains false a change to a does not matter). In this case, we may well discard a from the change impact scope which means that a change to a should not trigger a re-validation of $a \wedge b$ (expressed in Table 1, row 3). We see that initially, both a and b are validated ($a = true$, $b = false$) but a is discarded from the scope (column 'initial'). If a changes (column 'change a'), no re-validation is performed. If b changes, however, we need to validate a because it may have changed since (by having discarded it from the change scope we no longer know what happened to it since). The scope stays the same unless a is false in which case b may be discarded (recall from above that b needs no validation if a is false).

Row 4 in Table 1 depicts another situation where $a = true$ and $b = true$. The validation result of the conjunction is thus true and it is clear that both a and b may affect this validation result if either one of them were to change. Thus, both need to be validated and nothing can be discarded. This is the worst case for a conjunction where there is no apparent savings in the validation time and scope (memory consumption). However, even here we find savings in how incremental validation is performed. For example, if b changes then it must have become false and no validation is necessary to determine that the validation result of the conjunction is false also, i. e., a need no re-validation and can be discarded from the scope.

Table 1. Initial Validation and Re-validation of a Conjunction

	a	b	$a \wedge b$	validate/discard		
				initial	change a	change b
1	false	false	false	$a/-$	b/a	$-/-$
2	false	true	false	$a/-$	$b/-$	$-/-$
3	true	false	false	ab/a	$-/-$	$a/\begin{cases} b & \text{if } a = \text{false} \\ - & else \end{cases}$
4	true	true	true	$ab/-$	$-/b$	$-/a$

To illustrate the benefits, consider the total number of validations and dismissals in Table 1. We see that the initial validation investigates at least a and often also b (in average 1.5 validations depending on situation). Here, our approach's performance is identical to the traditional Model/Analyzer approach. However, the advantage of our approach becomes apparent with the changes. Traditional approaches have to pay the initial validation cost for all subsequent changes. In our approach, we see that for changes to a only 0-1 (average 0.5) re-validations are necessary (instead of the 1.5). The same is true for changes to b with an even lower average of 0.25. These saving are small if we consider expressions individually but these saving accelerate with every expression. As an example, assume that a in $a \wedge b$ is another conjunction $a_1 \wedge a_2$: $(a_1 \wedge a_2) \wedge b$. If a_1 changes but a_1 was discarded from the scope (Table 1) then no re-validation is necessary. If a_1 was not discarded then it may affect $a_1 \wedge a_2$ and we need to re-validate $a_1 \wedge a_2$ to be certain. Only if the re-validation of $a_1 \wedge a_2$ shows that it indeed changes and Table 1 reveals that its change may affect $a \wedge b$ where $a = a_1 \wedge a_2$ then we re-validate $a \wedge b$ (where the validation result for a is already known because it was just validated). The more complex the design rule, the more significant must be the savings. Consider that $a_1 \wedge a_2 = true$ and $b = false$. In this case, the a branch, consisting of $a_1 \wedge a_2$, is discarded from the scope which means that neither a_1 nor a_2 can affect the design rule. Each upward re-validation step is thus a double filter: 1) to assess whether the change can impact the validation result of the step and 2) only if that validation result changes then the step above is re-validated.

To achieve these performance gains, our approach must retain some intermediate validation results from the initial validation or subsequent re-validation in form of a validation tree (will be discussed below). However, we will demonstrate that this memory consumption is moderate because significant parts of the validation tree can be discarded as shown in the small example above.

Naturally, our approach is not just limited to conjunctions. Design Rule (1) consist of a conjunction but its arguments are more complex expressions such as quantifiers. Table 2 shows the validated and discarded parts for the initial validation and re-validation for possible changes on an existential quantifier. An existential quantifier is similar to concatenated disjunctions. From this it follows that if the existential quantifier validates to true then one validation of the source elements must be kept to ensure that this quantifier can change its state only if at least this validation fails. All others can be discarded. Other logical expression, such as disjunctions, implications, negation, the universal quantifier, . . . , are analogous and we omit their discussion due to brevity.

Table 2. Initial Validation and Re-validation of an Existential Quantifier

	$A = \{a_1, a_2\}$	$\exists a \in A \mid a$	validate/discard					
			initial	add a_3	delete a_1	delete a_2	change a_1	change a_2
1	{false, false}	false	$Aa_1/-$	Aa_3/a_1	Aa_2/a_1	$-/-$	$a_1/-$	$-/-$
2	{false, true}	true	Aa_1a_2/a_1	$-/-$	$-/-$	Aa_1/a_2	$-/-$	Aa_1/a_2
3	{true, false}	true	$Aa_1/-$	$-/-$	Aa_2/a_1	$-/-$	Aa_2/a_2	$-/-$
4	{true, true}	true	$Aa_1/-$	$-/-$	Aa_2/a_1	$-/-$	Aa_2/a_1	$-/-$

3.2 Filtered Validation Tree

The validation tree is a structured log of the validation of a design rule and depicts every expression performed, the order they were performed, the model elements that were accessed, and all intermediate results generated. This validation tree will be generated the first time a design rule is validated on a model element, i. e., on start-up and when new model elements that match the context of a design rule are created. Figure 2 shows a validation tree for the message ‘connect‘. Algorithm 1 describes how the validation tree is built and how parts of it are discarded to reduce the impact of a model change (CPU savings) and reduce memory.

At the beginning, the algorithm distinguishes (line 2) between expressions that have a Boolean result (e. g., conjunction, existential quantifier) and all other expressions (e. g., model access, string or collection manipulations). If the expression is a Boolean expression then the first action (3) is to create a node in the validation tree that points to the expression (ϵ) (e. g., the conjunction for Design Rule (1) becomes the root node in the tree). The next step is the validation of the arguments (α) of the expression (6) which is done by a recursive call of the validate algorithm for all its arguments. The validation is guarded by a condition (5) that the argument needs validation only if the result is not already in the validation tree. Of course, during the initial validation no results are in the validation tree; however, we will see later that re-validation makes use of this algorithm also and the need for this condition will be explained in the next section. An edge will be added between the node of this expression and the node created during the validation of each argument (7). The algorithm distinguishes between the different logical operation types (o). Due to brevity only the conjunction (8) and the existential quantifier (10)

Algorithm 1. Initial Validation and Creation of a Validation Tree

```
1  validate(Expression e)
2      if (e.operation is a boolean operation)
3          add node(reference to e)
4          for (i=1 to #e.arguments) //validate arguments
5              if (e.arguments[i] is not in validation tree)
6                  validate(e.arguments[i])
7              add edge(e.node, arguments[i].node)
8              if (e is-a conjunction)
9                  if e.arguments[i].result!=false next
10             else if (e is-a existential)
11                 if e.arguments[i].result!=true next
12             ...
13             else next
14         e.result = e.operation(e.arguments) //compute validation result
15         if (e is-a conjunction) //filter 1 validation tree (discard)
16             if (e.arguments[1] and !e.arguments[2]) remove edge(e,e.arguments
                     [1])
17         else if (e is-a existential)
18             if (e.result=true)
19                 for (i=1 to #e.arguments)
20                     if e.arguments[i]=false remove edge(e, e.arguments[i])
21         ...
22     else
23         e.result = e.operation(e.arguments)
24         if (e.operation accesses model elements)
25             add node(reference to model elements)
```

are given (there were discussed above in detail), but the other operations can be derived from these two operation because of the rules of the Boolean algebra. Not all arguments need to be validated to compute the validation result of the expression (9, 11 are analogous to Table 1). After the validation has finished, the result is calculated (14) and the filtering of the validation tree starts (15-21). The filtering of the validation tree is the dismissal of previously validated nodes/edges (e. g., see Table 1 third row or Table 2 second row). For example, if the expression is a conjunction and the first argument is true but the second one is false then the edge to the first argument can be discarded.

As was said, the algorithm distinguishes between Boolean and non Boolean expressions. The non-Boolean expressions are usually model accesses (e. g., retrieve the name of a message) or manipulations (e. g., remove an association from a collection). Those expressions are processed in lines 23 to 25. Essentially, our approach keeps track of all model elements accessed for which we create leaf nodes in the validation tree. The actual results computed by these expressions are discarded eventually to minimize the memory overhead of our approach. The leaf nodes typically only contain the references to the model elements through which the results were computed. In Design Rule (1), the source of the first existential quantifier is a property call: $m.receiveEvent.covered$ which includes accesses to two model elements: the ‘receiveEvent‘ of the message ‘m‘ and from its result the ‘covered‘ property. This sequence of two property calls reveals the lifelines that act as message receivers. The leaf node will identify these accessed model elements and properties. If one of them should change then the existential quantifier would be (potentially) affected and may require re-validation. In our example the existential quantifier that iterates over all the operations of class ‘Streamer‘ creates one node and edge to the model element properties that are accessed to get the operations.

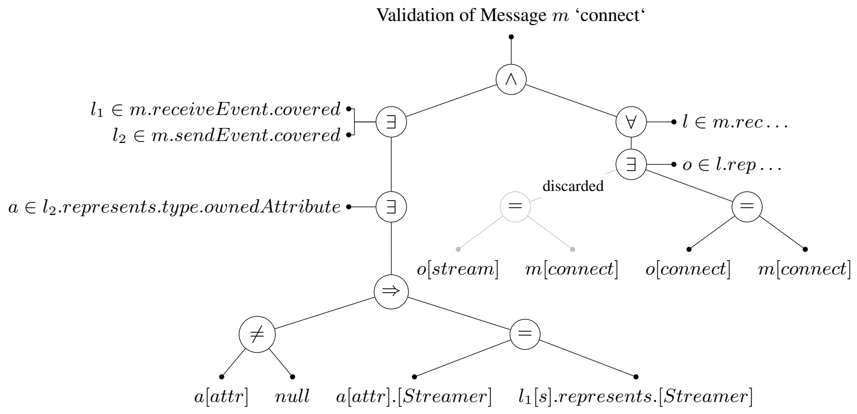


Fig. 2. Validation Tree for Message 'connect'

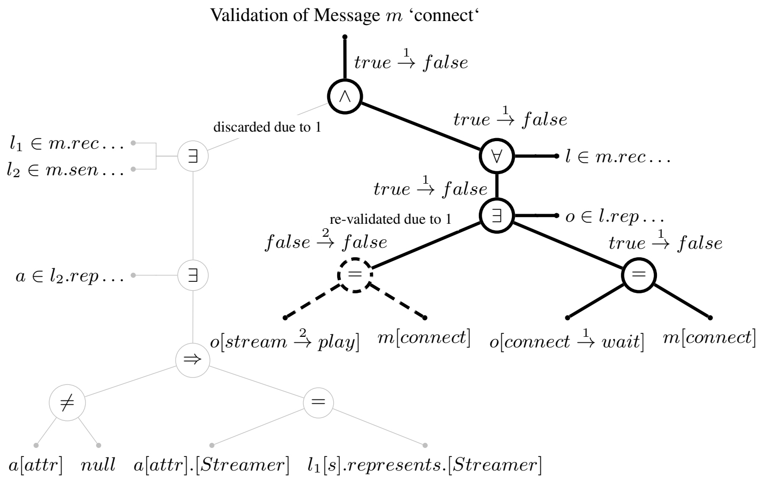
For each operation in the source a sub tree representing the condition of the quantifier is created. After the sub trees are created, all the sub trees that are not needed for the validation result of the existential quantifier are discarded (refer to Table 2).

Since the validation of a design rule discards parts of the validation tree, we speak of a *filtered validation tree*. The filtered validation tree contains only nodes representing the Boolean expressions with their Boolean validation results. Both are cheap to maintain in terms of memory consumption. All other validation results are discarded after the validation except for the model element/properties that were accessed to compute the results. These model accesses are references to the design model and such references are also cheap to maintain in terms of memory consumption.

3.3 Impact of a Change

Once a validation tree has been created, only those parts must be re-validated that are affected by the change. The initial generation of a validation tree is strictly top down whereas the incremental re-validation is mostly bottom up. The previous section discussed one part of the benefits of our approach in that the re-validation focuses on the filtered validation tree only and ignores changes that would have affected discarded parts of the validation tree. This saves both memory and improves performance because a change becomes less likely to affect the validation tree. This section discusses another part of the benefits of our approach. It demonstrates that changes propagate upward for as long as the nodes are affected by the change only. Since the validation tree stores all intermediate Boolean results, only changes to these results must be computed anew. The model elements accessed (scopes) are referenced at the bottom of the tree (the leaves) and the impact of model element changes thus always start at the leaves and is propagated upward towards the root.

We illustrate this on two change examples and the validation tree for message 'connect'. The first change (change 1) is the renaming of the operation 'connect' to 'wait' in Figure 1. The second change (change 2) is the renaming of operation 'stream' to 'play'. Their impacts on the validation tree are shown in Figure 3. Change 1 is drawn in a thick



Validation of Message m 'connect'
$true \xrightarrow{1} false$
discarded due to 1
$true \xrightarrow{1} false$
$l_1 \in m.rec\ldots$
$l_2 \in m.sen\ldots$
$l \in m.rec\ldots$
$true \xrightarrow{1} false$
re-validated due to 1
$o \in l.rep\ldots$
$false \xrightarrow{2} false$
$true \xrightarrow{1} false$
$a \in l_2.rep\ldots$
$o[stream \xrightarrow{2} play]$
$m[connect]$
$o[connect \xrightarrow{1} wait]$
$m[connect]$
$a[attr]$
$null$
$a[attr].[Streamer]$
$l_1[s].represents.[Streamer]$

Fig. 3. Impacts of two Changes on the Validation Tree

solid black line and change 2 as a thick dashed line. Algorithm 2 shows the handling of a change in pseudo code.

Change 1 affects the operation 'connect' ($o[connect \xrightarrow{1} wait]$) which is now named 'wait' (represented by the arrow with the number of the change on top of it from $connect \xrightarrow{1} wait$). Algorithm 2 first identifies all the leaf nodes that reference the changed model element. The re-validation is bottom up and starts at these leaf nodes (27, 28). For each leaf node, the re-validation saves the previous, old result (31) and re-validates the expression (32). Since the approach maintains the Boolean results in a validation tree only, it follows that no results are saved for leaf nodes and the 'oldValue' remains undefined. Leaf nodes are thus always re-validated and their results propagated upwards (33) to the parent node (35). In our example, the new name of the operation is retrieved and then propagated up to the equals expression in Figure 3. The equals expression must be re-validated using the new value from the left branch and either the old value from the right branch (if the validation tree has the value) or a computed value otherwise (note lines 5-6 in Algorithm 1). In our example, the result of the equals expression changes from true to false and as the value changed, the new value will be propagated up to the existential quantifier, the parent of the equals expression. As this was the only node that made the existential quantifier true, its change causes the re-validation of other branches. Recall that during the creation of the validation tree the branch for operation 'stream' was discarded because it did not affect the validation result of the existential quantifier. This branch may have changed since and needs to be re-validated. If there are no other elements or none of the other elements satisfy the existential quantifier (as in our example), the result changes and the new result is propagated up to the universal quantifier. This universal quantifier fails also because of the fail to the existential quantifier and the new value is propagated to the conjunction, the top node of the validation tree (it has no parent, line 34). This node will validate to false also, the overall evaluation of this design rule changes from consistent to inconsistent. As can be seen 11 out of 22 nodes (a complete re-validation) must be re-validated

Algorithm 2. Processing a Model Change

```
26  processChange(Element elem)
27      for all (node:validation trees | node references elem)
28          revalidate(node.expression)
29
30  revalidate(Expression e)
31      oldResult = e.result // is empty if leaf
32      e.result = validate(e)
33      if (oldResult != e.result) // filter 2 stop bottom up propagation
34          if (e has parent node)
35              revalidate(e.parent)
```

due to this change only. Furthermore, the left part of the validation tree (the first argument of the conjunction) will be discarded thereafter because it cannot influence the validation result after the change, cutting 11 nodes from the 22 nodes. This benefits the next re-validation because a change becomes less likely to affect the new filtered validation tree.

The second change affects operation ‘connect‘ ($o[stream \xrightarrow{2} play]$). Without the first change this change would not have affected the validation tree because this branch was discarded during the initial validation and only added again after the first change. The operation ‘connect‘ is thus in the change scope now. However, the re-validation of the second change stops at the equals relation because the result of this node does not change (33), i. e., it remains false. The upward propagation of changes thus stops once the re-validated result of a node is equal the previously known result (‘oldResult‘) of that node. In this case 3 out of the 11 remaining nodes must be re-validated only.

In contrast to other approaches, where the change of one model element triggers a re-validation of the design rule in its entirety, our approach only triggers the re-validation of those nodes in the validation tree that are affected by a model change. Since a discarded argument in an expression discards the entire branch, the reductions increase with the complexity of the design rule (the number of nodes). For the non-discarded arguments, it must be noted that the re-validation is mostly upwards from leaf to root nodes and new validations (top down) are limited to sub trees only. If multiple model elements change then the same validation tree may have to be validated multiple times, but these re-validations always start at distinct parts of the tree and may only join at common roots (and only if the change affects the parent expressions). Redundancies are possible only if the changes trickle to common roots which is often not the case. Further optimizations are possible here.

4 Evaluation

We empirically validated our approach on 30 industrial UML models ranging from small to large models (127 to 67,723 model element/properties). These models were evaluated on 19 design rules. The design models are in part taken from [6] and were transformed from UML 1.4 to UML 2.1 (this explains the differences in model sizes). The design rules were also taken from [6] and converted form Java to OCL design rules (some of them could not be converted due to the limited expressiveness of OCL). The

Table 3. Model Size, Design Rule Validations and Memory Overhead

Name	#Model Elements	#Scope Elements	#Design Rule Validations	MOH Brute f. [MB]	MOH Filtered [MB]	MOH MDT OCL [MB]
Video on Demand	90	127	63	2	2	1
ATM	220	763	304	20	10	7
Microwave Oven	290	296	138	29	13	9
Model View Controller	418	834	393	16	12	7
eBullition	513	892	341	53	18	10
Curriculum	763	1,350	595	150	43	4
Teleoperated Robot	1,115	1,969	885	97	34	5
Dice 3	1,274	1,649	599	74	14	3
ANTS Visualizer	1,282	3,119	1,225	169	93	6
Inventory and Sales	1,296	1,898	803	250	17	4
Course Registration	1,406	1,822	712	97	19	4
UML IOC F05a T12	1,453	2,441	998	67	23	6
VOD 3	1,558	4,652	1,789	175	110	7
Vacation and Sick Leave	1,658	2,681	1,084	145	65	5
Home Appliance	1,707	2,115	784	267	53	6
HDCP Defect Seeding	1,784	2,199	985	72	36	7
DESI 2.3	1,974	4,727	1,838	188	106	7
iTalks	2,212	4,049	2,289	417	130	6
Hotel Management Sys.	2,583	4,244	2,033	359	87	5
Biter Robocup	2,632	6,265	2,334	227	129	8
Calendarium	2,809	6,160	2,694	326	79	6
UML LCA F03a T1	2,983	2,912	1,243	108	53	3
<unnamed>	5,373	6,804	2,906	973	129	7
NPI	7,110	8,536	2,930	1,353	97	7
Word Pad	8,078	17,907	8,186	860	513	20
dSpace 3.2	8,761	12,994	5,869	N.A.	259	11
OODT	9,828	26,650	11,384	752	434	21
Insurance Network Fees	16,255	27,442	10,562	N.A.	172	58
<unnamed>	33,347	33,844	16,627	N.A.	382	111
<unnamed>	64,061	67,723	40,297	N.A.	724	61

complete list of design rules and the Model/Analyzer tool can be found on our tools homepage http://www.sea.jku.at/tools/.

Before we evaluate the performance of the new approach we have to ensure that the validation of the design rules are correct. To validate this, we compared the results with the standard MDT (Modeling Development Tools of Eclipse) implementation of OCL. Given the large number models and over 90,000 correct design rule validations, the approach can be considered correct.

The evaluation of our approach covers the memory overhead and the performance. The evaluations were done using our implementation for the IBM Rational Software Architect (RSA) on an Intel Core 2 Quad CPU @2.83GHz with 8GB (4GB available for the RSA) RAM and 64bit Linux (3.1.9). We compare our evaluation results against incremental approaches exclusively and do not address the improvements against batch

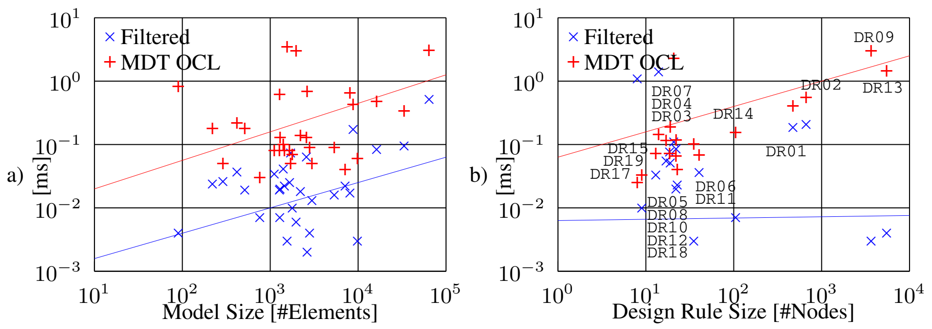


Fig. 4. Evaluation Time (in ms) of MDT OCL and the Optimized Approach

evaluation which is already done in our former work [6]. We evaluated the memory overhead based on three criteria: 1) the memory used by a *Brute force* validation, i. e., each intermediate validation result as well as the property calls are cached, 2) the memory overhead of the filtered validation tree, and 3) the memory overhead of the MDT OCL validation which caches the scope only. For larger models the *Brute force* validation could not be completed due to out-of-memory exceptions (N.A.). The memory consumption was measured using the ‘Runtime‘ interface of Java and before measuring the garbage collector was activated. The measured data were double checked with data from the TPTP profiler for eclipse.

Table 3 shows the evaluation results for the memory overhead (MOH in Mega Byte) in relationship to the model sizes, the accessed model element properties, and the quantities of validated design rules. Compared to the worst case (*Brute force*), the reduction of the used memory (*Filtered*) is between 50% and 80% (except for the very small models). As can be seen, the reduction depends not only on the model size but also on the number of validated design rules and accessed model elements. The main memory overhead (*MDT OCL*) is caused by the scope that must be built, even for other incremental consistency checking approaches that are scope based.

The evaluation of the performance is done in the same environment using ‘System.nanoTime()‘ (resolution $1\mu s$). We compared the evaluation of the validation using the MDT OCL environment with the validation of the filtered validation tree. Figure 4 shows the timing results regarding the model size (a-left) and regarding the design rule complexity (b-right). We measured the average re-validation of 50 random model changes on each model. The model changes cover the modification of model element properties, the addition of model elements and their deletion. As can be seen, the re-validation times (measured in ms) are nearly independent of the model size, but the state-of-the-art OCL validation times are about 2 to 20-fold slower than compared to our new approach. Slightly different are the results for the design rule complexity. Whereas the re-validation time remains stable for our new approach, the re-validation using the MDT OCL increases linear with the design rule complexity (please note the logarithmic scale on both axes for both diagrams).

It is intuitive to believe that our approach should perform better for larger, more elaborate rules because our approach does not require the re-validation of the entire rule (=equivalent to the entire validation tree) but only some paths from the leaves to the root. If a validation tree has m nodes (=expressions) then the run-time complexity of the normal approach should be $O(m)$ whereas the run-time complexity of the optimized approach should be $O(log(m))$. Figure 4 (b) confirms this hypothesis for all rules. The larger the rules (x-axis, measured in the average sizes of their evaluation trees), the more significant the saving.

5 Threats to Validity

The threats to validity are mostly centered on the random testing of changes. Random changes can lead to models that may not be valid and do not conform to the UML standard. Still, they are possible changes and it may be useful to know that our approach is superiors, perhaps even with changes that are impossible. Another aspect has to do with the fact that random changes may under represent expensive changes. This assumes that there are changes that are likely and expensive. However, previous work has shown that likely changes are rarely expensive changes [8]. The reason we relied on random changes was simply our desire to perform large quantities of model changes. Second, the validation time for the state-of-the-art MDT OCL validations is measured without the generation of the scope because to do so the UML implementation must be instrumented (as on the case of our former work). Using the new approach this is not necessary any more. However, this implies that the results of our approach should be even better because computing the scope would have been higher, as would have been the memory cost.

6 Related Work

Cabot and Teniente present an event triggered approach to detect inconsistencies in UML/OCL conceptual schema [4]. They use a list of events that trigger the re-evaluation of consistency rule. An event is a modification in the model and using this events they reduce the amount of rules that must be re-validated in the model. They also use a syntax tree that is annotated with events that potentially violate the constraint expressed in OCL. In contrast to our approach the use a static analysis of the OCL constraint and as such the incremental characteristic is limit to single constraints only.

Jouault et al. [12] developed an incremental approach using ATL (AtlanMod Transformation Language) to transform OCL rules and to trigger only those rules that are affected by a model change. This approach similar to ours as it uses model element and their properties to trigger the re-evaluation of constraints. But their main focus is on a static analysis of the constraints where the trigger events are extracted whereas we are able to consider parts of a constraint only. Unfortunately, they provide no evaluation of their approach regarding the evaluation time and memory consumption.

Similar, Blanc et al. [3] achieve near instant performance thanks the re-writing of design rules for each relevant model change. This requires the engineer to re-factor consistency rules to understand the impact of model changes. If done correctly, this leads to good performance. However, since writing these annotations may cause errors, they are no longer able to guarantee the correctness of incremental consistency checking.

Bergmann et al. [2] present an query based approach. This approach is similar to the approach presented in this paper, but they use a query language (IncQuery) that is executed on EMF models. Their approach is based on the Viatra2 [20] framework and in RETE [11] networks. In their approach the queries must be stored permanently in memory and the values must be updated after each model change. In contrast, in our approach we only store references on the model elements and the Boolean values of the validation tree, which shortens the massive memory consumption problem. Unfortunately, the smallest unit in their timings are 10ms which is rather high for the problem addressed in our paper.

Nentwich et al. present xLinkIt [14], a language that evaluates the consistency of "documents", including UML design models. Design rules are expressed in a uniform manner and xLinkIt is capable of checking the consistency of models incrementally. However, it requires between 5 and 24 seconds for evaluating changes and the tool is thus not able to keep up with an engineer's rate of model changes. The approach by Reiss [16] is in principle alike xLinkIt. Rather than defining consistency rules on XML documents, Reiss defines consistency rules as SQL queries which are then evaluated on a database which may hold a diverse set of artifacts. Reiss' use of a database makes his approach certainly more incremental. However, the incremental updates in his study take about 30 seconds to 3 minutes. ArgoUML [17] was probably the first UML design tool to implement incremental design checking but it required annotated consistency rules. Their annotations were lightweight but so where their computational benefits.

In the context of pervasive computing Xu et al. optimized the re-validation of design rules [21]. The also use validation trees to for their optimization but in contrast to our approach they process modifications of the context (the location) only. However, as we address model-based software development we have to deal with more types of changes. Furthermore, we optimize the tree in the post process to achieve better results regarding reduced memory consumption and re-validation effort.

7 Conclusions and Future Work

This work introduced a novel approach to the incremental validation of design rules in design models. Empirical validation on 19 design rule has shown that our approach reduces the time to re-validate a design rule up to 95%. This observation was made on a large number of design models and we found that it outperformed the state of the art under all situations by a large margin. Indeed, we have not encountered a single design rule that would not benefit from our approach. This work paves the way for processing a much larger number of model changes and/or more complex model changes with instant or near instant response times. In our future work, we will use this re-validation approach to simulate repair actions and determine the effects that such actions have on the overall design model and on other design rules.

Acknowledgments. This research was funded by the Austrian Science Fund (FWF): P21321-N15.

References

1. Balzer, R.: Tolerating Inconsistency. In: Belady, L., Barstow, D.R., Torii, K. (eds.) ICSE, pp. 158–165. IEEE Computer Society/ACM Press (1991)
2. Bergmann, G., Horváth, Á., Ráth, I., Varró, D., Balogh, A., Balogh, Z., Ökrös, A.: Incremental Evaluation of Model Queries over EMF Models. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 76–90. Springer, Heidelberg (2010)
3. Blanc, X., Mounier, I., Mougenot, A., Mens, T.: Detecting model inconsistency through operation-based model construction. In: Schäfer, W., Dwyer, M.B., Gruhn, V. (eds.) ICSE, pp. 511–520. ACM (2008)
4. Cabot, J., Teniente, E.: Incremental integrity checking of UML/OCL conceptual schemas. Journal of Systems and Software 82(9), 1459–1478 (2009)
5. Demsky, B., Rinard, M.: Data structure repair using goal-directed reasoning. In: Proceedings of the 27th International Conference on Software Engineering, ICSE 2005, pp. 176–185. ACM, New York (2005)
6. Egyed, A.: Instant consistency checking for the UML. In: Proceedings of the 28th International Conference on Software Engineering, ICSE 2006, pp. 381–390. ACM, New York (2006)
7. Egyed, A.: Fixing Inconsistencies in UML Design Models. In: ICSE, pp. 292–301. IEEE Computer Society (2007)
8. Egyed, A.: Automatically Detecting and Tracking Inconsistencies in Software Design Models. IEEE Trans. Software Eng. 37(2), 188–204 (2011)
9. Egyed, A., Letier, E., Finkelstein, A.: Generating and Evaluating Choices for Fixing Inconsistencies in UML Design Models. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, ASE 2008, pp. 99–108. IEEE Computer Society, Washington, DC (2008)
10. Fickas, S., Feather, M., Kramer, J.: Proceedings of ICSE-97 Workshop on Living with Inconsistency (1997)
11. Forgy, C.: Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem. Artificial Intelligence 19, 17–37 (1982)
12. Jouault, F., Tisi, M.: Towards Incremental Execution of ATL Transformations. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 123–137. Springer, Heidelberg (2010)
13. Mens, M., Ragnhild, S., D'Hondt, M.: Detecting and Resolving Model Inconsistencies Using Transformation Dependency Analysis. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 200–214. Springer, Heidelberg (2006)
14. Nentwich, C., Capra, L., Emmerich, W., Finkelstein, A.: xlinkit: a consistency checking and smart link generation service. ACM Trans. Internet Technol. 2(2), 151–185 (2002)
15. Nentwich, C., Emmerich, W., Finkelstein, A.: Consistency management with repair actions. In: Proceedings of the 25th International Conference on Software Engineering, ICSE 2003, pp. 455–464. IEEE Computer Society, Washington, DC (2003)
16. Reiss, S.P.: Incremental Maintenance of Software Artifacts. IEEE Trans. Software Eng. 32(9), 682–697 (2006)
17. Robbins, J.E.: ArgoUML, v0.32.1 (March 2011), `http://argouml.tigris.org`
18. Sabetzadeh, M., Nejati, S., Liaskos, S., Easterbrook, S.M., Chechik, M.: Consistency Checking of Conceptual Models via Model Merging. In: RE, pp. 221–230. IEEE (2007)

19. Van Der Straeten, R., D'Hondt, M.: Model refactorings through rule-based inconsistency resolution. In: Proceedings of the 2006 ACM Symposium on Applied Computing, SAC 2006, pp. 1210–1217. ACM, New York (2006)
20. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. Sci. Comput. Program. 68(3), 214–234 (2007)
21. Xu, C., Cheung, S.C., Chan, W.K.: Incremental consistency checking for pervasive context. In: Proceedings of the 28th International Conference on Software Engineering, ICSE 2006, pp. 292–301. ACM, New York (2006)

Evaluating the Impact of Aspects on Inconsistency Detection Effort: A Controlled Experiment

Kleinner Farias, Alessandro Garcia, and Carlos Lucena

OPUS Research Group/LES, Informatics Department, PUC-Rio
Rio de Janeiro - RJ - Brazil
{kfarias,afgarcia,lucena}@inf.puc-rio.br

Abstract. Design models represent modular realizations of stakeholders' concerns and communicate the design decisions to be implemented by developers. Unfortunately, they often suffer from inconsistency problems. Aspect-oriented modeling (AOM) aims at promoting better modularity. However, there is no empirical knowledge about its impact on the inconsistency detection effort. To address this gap, this work investigates the effects of AOM on: (1) the developers' effort to detect inconsistencies; (2) the inconsistency detection rate; and (3) the interpretation of design models in the presence of inconsistencies. A controlled experiment was conducted with 26 subjects and involved the analysis of 520 models. The results, supported by statistical tests, show that the effort of detecting inconsistencies is 20 percent lower in AO models than in their OO counterparts. On the other hand, the inconsistency detection rate and the number of misinterpretations are 43 and 37 percent higher in AO models than in OO models, respectively.

Keywords: Aspect-Oriented Modeling, Model Composition, Inconsistency, Developer Effort, Empirical Studies.

1 Introduction

Modeling languages (e.g., UML [11] and its extensions) provide different types of models, such as class and sequence diagrams, to represent the structure and behavior of software systems. These complementary models represent the design decisions that developers will implement later. In practice, these models often suffer from the inconsistency problems [16]. These inconsistencies are mainly caused by the mismatch between the overlapping parts of complementary models and the lack of formal semantics to prevent these contradictions [2][3]. Consequently, developers must invest some effort to detect and properly deal with these inconsistencies [6]; otherwise, emerging misinterpretations of the design models can compromise the resulting implementation.

Different modeling languages support different forms of modular decomposition and may influence how developers detect or even neglect inconsistencies [3]. This might be particularly the case with aspect-oriented modeling (AOM) [7][17] as it intends to improve design modularity of otherwise crosscutting concerns. Current research in AOM varies from UML extensions [7][17][19][20] to alternative strategies for model weaving. Unfortunately, nothing has been done to investigate whether aspect-oriented models can alleviate the burden of dealing with model inconsistencies. Someone might hypothesize that they might help developers to understand the design before implementing it. Others could also postulate that the improved modularization would reduce the effort to detect inconsistencies or even reduce misinterpretations arising between complementary design models.

Unfortunately, it is by no means obvious whether these assumptions hold (or not). First, it may be the case that additional constructs in AO models lead to detrimental effects on design understanding. Second, it is still not clear if an aspect affecting multiple join points may increase the inconsistency detection and improve the model interpretation. Third, developers might get “distracted” by the global reasoning motivated by the presence of crosscutting relations [10] between classes and aspects. At last, developers might even invest more effort using AO models while examining all points that are crosscut by the aspects [6].

In this context, this paper reports a controlled experiment (Section 3) aimed at investigating the impact of AOM on: (1) the rate of inconsistency detection; (2) the developers' effort to detect these inconsistencies; and (3) developers' misinterpretation rate. We compare the use of AO models to OO models in a particular context: the use and understanding of design models by developers needed to produce the corresponding implementation. The results (Section 4) supported by statistical tests and qualitative analysis, show that AO models alleviate the effort to detect inconsistencies. But, it neither reduces inconsistency detection rate nor misinterpretation rate.

Moreover, we also discuss some additional findings (Section 4.4). For instance, we observed that the downsides of AOM were, to a large extent, caused by the degree of quantification [10] of the aspects. That is, the higher the number of modules affected by an aspect, the lower the inconsistency detection rate and the higher the misinterpretation rate. Moreover, we observed that developers tended to detect inconsistencies more quickly in AO models when the scope of aspect pointcuts was narrow. Equally relevant was the finding that the required mental model is directly influenced by the number of crosscut relationships.

To the best of our knowledge, our results are the first to pinpoint the potential (dis)advantages of AOM in imprecise multi-view modeling. After presenting how we tried to mitigate the possible threats to validity (Section 5), we make it clear the contributions of our experiment in the light of the related work (Section 6) and present final remarks (Section 7).

2 Background

2.1 Aspect-Oriented Modeling

Aspect-oriented modeling (AOM) languages aim at improving the modularity of design models by supporting the modular representation of concerns that cut across multiple software modules. This superior modularization of crosscutting concerns is achieved by the definition of a new model element, called *aspect.* An aspect can crosscut several modules within a system. These relations between aspects and other modules are called *crosscut* relationships. These basic concepts and other aspect-oriented modeling elements are usually represented as classic UML stereotypes in AOM languages [7][17]. The AOM language used throughout our study is a UML profile [17][19][20]. The choice of the UML profile for AOM is based on some reasons. First, the Unified Modeling Language [11] is the standard for designing software systems. Second, the use of stereotypes reduces the gap between subjects with low skill (or experience) and highly skilled (or experienced) subjects. Third, the model reading technique used by the subjects would not be influenced by new notation issues; therefore, the interpretation of the models is exclusively influenced by the use of the concepts in object-oriented and aspect-oriented modeling. Finally, UML profile for AO programming is the approach more common for structural and behavioral diagrams [11].

Fig. 1 presents an illustrative example of the models used in our study: a class and a sequence diagram of the AOM language used in our study. The notation supports the visual representation of aspects, crosscutting relationships and other AOM concepts. The stereotype <<aspect>> represents an aspect, while the dashed arrow decorated with the stereotype <<crosscut>> represents a crosscutting relationship. Inner elements of an aspect are also represented, such as *pointcut* (<<pointcut>>) and advice. An advice adds behavior before, after, or around the selected join points [7]. The stereotype associated with an advice indicates when (<<before>>, <<after>> or <<around>>) a join point is affected by the aspect. The join point is a point in the base element where the advice specified in a specific pointcut is applied.

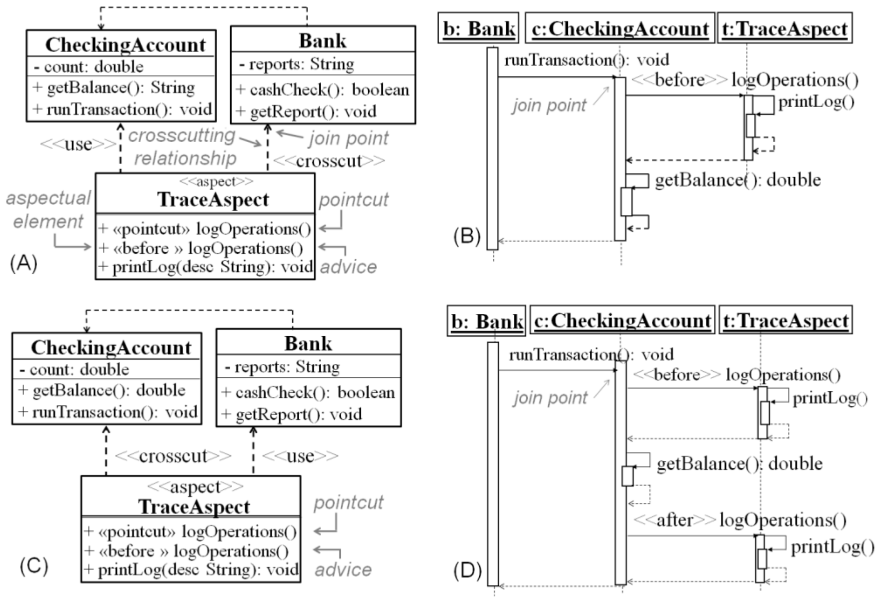


Fig. 1. An illustrative example of aspect-oriented models used in our study. (A) and (B) represent the conflicting structural diagrams. (C) and (D) represent the structural and sequence diagrams without inconsistencies.

2.2 Model Inconsistency and Detection Effort

The multiple views of a software system inevitably have conflicting information [2]. If software developers do not detect and properly deal with these inconsistencies the potential benefits of the use of the models (e.g., gain in productivity) can be compromised. Developers must invest some considerable effort (time) to detect these inconsistencies; otherwise, the potential benefits of the use of models such as specification of the implementation of a system can be compromised. Two broad categories of inconsistencies were used in this study: (1) *syntactic inconsistencies*, which arise when the models not conforming to the modeling language's metamodel; and (2) *semantic inconsistencies*, where the meaning of the model element does not match that of the actual design model. We have particularly selected semantic inconsistencies that are: (i) detectable by developers [2], and (ii) difficult or impossible to detect automatically. We focused on inconsistencies that have been documented elsewhere [3] and used in a previous empirical study [2]. A complete description is also available at our complementary website [9], and two representative examples are presented below:

1) *Conflicting relationships*: the nature of a relationship diverges in structural and behavioral models. For instance, according to the sequence diagram, the advice of an aspect A crosscuts the behavior of class B; however, the semantics of the advice in A dictates when the class diagram should have either a <<crosscut>> or a <<use>> relationship between A and B. For example, Fig. 1 presents this kind of inconsistency. The aspect *t:TraceAspect* crosscuts the *c:CheckingAccount* objects (Fig. 1.B). In this case, the relationship between *TraceAspect* and *CheckingAccount* should be <<crosscut>> instead of <<use>> (see Fig. 1.C) given the logging semantics of the advice *logOperations()*. In the structural diagram (Fig. 1.A), the aspect *TraceAspect* has a <<use>> relationship with the class *CheckingAccount* instead of <<crosscut>> relationship.

2) *Messages with different return types*: the return type of a message *m* from an object A to an object B does not match with the return type of the method M in the corresponding class B in the class diagram. For instance, the method *CheckingAccount.getBalance* has conflicting return types: *string* in the class diagram and *double* in the sequence diagram. A similar conflict can occur with the return type of a around advice [17] and the return type from a method execution being advised by the latter.

Developers detect inconsistencies when they identify conflicting information in the models and, then, report that the models cannot be implemented. This decision often relies on "guessing" the semantics of the model elements. To reach this conclusion, developers need to invest some effort: the time (in minutes) to go through the model and infer that the models suffer from inconsistencies.

3 Study Methodology

3.1 Goal, Research Questions, and Context

We formulate the goal of this study using the GQM template [5] as follows:

Analyze AO and OO modeling techniques *for the purpose of* investigating the impact *with respect to* detection effort and misinterpretation *from the perspective of* developers *in the context of* multi-view design models.

Based on this, we focus on the three research questions:

RQ1: Does AOM affect the efficiency of developers to detect multi-view model inconsistencies?

RQ2: Does AOM influence effort invested by developers to detect model inconsistencies?

RQ3: Do AO models lead to a different misinterpretation rate as compared to OO models?

The context selection is representative of situations where developers implement classes (or aspects) based on design models. The experiment was conducted within two postgraduate courses at the Pontifical Catholic University of Rio de Janeiro (PUC-Rio) and the Federal University of Bahia (UFBA). Both courses are taught in the first year of Master and Doctoral programs in Computer Science. Therefore, all the subjects (18) hold a Master's or Bachelor's degree, or equivalent. In addition, eight (8) professionals from three companies also participated in the experiment. Most of the professionals held a Master's or Bachelor's degree.

3.2 Hypothesis Formulation

First Hypothesis (H_1). The first research question investigates whether developers by using AO models produce a lower (or higher) inconsistency detection rate than by using OO models. Usually developers do not indicate the presence of existing inconsistencies in multi-view models [3]. The main reason is that they can make implicit assumptions about the correct design decisions based on previous experience. Moreover, they might feel forced to produce an implementation even in the presence of inconsistency. Thus, our intuition is that developers identify fewer inconsistencies in AO models than OO models because they might get distracted by the global reasoning motivated by the presence of additional crosscutting relations in the models. Consequently, they may have a higher number of implicit assumptions to assemble the "big picture" of a system. However, it is by no means obvious that this hypothesis hold. Perhaps, the increased modularity of AOM models may help developers to switch more quickly between the behavioral and structural views while implementing their aspects. Consequently, the software developer may localize more inconsistencies than in OO models. These hypotheses are summarized as follows:

***Null Hypothesis 1, $H_{1\text{-}0}$*:** The inconsistency detection rate in AO models is equal or higher than in OO models.
***$H_{1\text{-}0}$*:** DetectionRate (AO) $\geq$ DetectionRate (OO)
Alternative Hypothesis 1, $H_{1\text{-}1}$: The inconsistency detection rate in AO models is lower than in OO models.
***$H_{1\text{-}1}$*:** DetectionRate (AO) < DetectionRate (OO)

Second hypothesis (H_2). The second research question investigates whether developers invest less (or more) effort to detect inconsistencies in AO models than OO models. The superior modularity of AO models may help developers to better match and contrast the structural and behavioral information about the crosscutting relations. In this case, developers may switch more quickly between the behavioral and structural views while systematically implementing their aspects. Thus, our expectation is that the higher the number of crosscutting relationships (an aspect crosscutting a wider scope) in the model, the lower the effort to detect inconsistencies. This assumption is based on the superior ripple effects of inconsistencies observed in AO models when model composition techniques are applied [6]. This propagation can directly affect the effort in detecting inconsistencies, since developers, facing the complexity of the propagations, avoid doing any implementation. That is, by using AOM developers tend to get more quickly convinced about the severity of multi-view inconsistencies. This means that they are more likely to report them and not going forward on the design implementation. However, it is not clear whether this intuition holds because, at first, developers may examine all model elements affected (or not) by the inconsistencies, or even the inconsistencies, to some extent, may even be confined in the aspectual elements. This leads to the second null hypothesis and an alternative hypothesis as follows:

***Null Hypothesis 2, $H_{2\text{-}0}$*:** The effort to detect inconsistencies in AO models is equal or higher than in OO models.
$H_{2\text{-}0}$: EffortToDetect (AO) $\geq$ EffortToDetect (OO)
***Alternative Hypothesis 2, $H_{2\text{-}1}$*:** The effort to detect inconsistencies in AO models is lower than in OO models.
***$H_{2\text{-}1}$*:** EffortToDetect (AO) < EffortToDetect (OO)

Third hypothesis (H_3). The third research question investigates whether the misinterpretation rate (MisR) of the developers is higher (or lower) in AO models than in OO models. The chief reason of the disagreement between developers' interpretation is the contradicting understanding of the design models. They are often caused by inconsistencies emerging from the mismatches between the diagrams specifying the multiple, complementary views of the software system [3]. Contradicting design models make it difficult for developers to think alike and, hence, producing code with the same semantics. The key reason is that software implementation widely depends on cognitive factors. Someone could consider that additional AOM concepts, such as crosscutting relationships or aspects, may negatively interfere in a common understanding of design models by different developers. For instance, developers need to precisely grasp the actual meaning of the crosscutting relations (in addition to all other relations), and when they are actually established during the system execution. Then, as developers have to examine all join points affected by the aspects, their extra analyses can increase the opportunities of diverging interpretations. However, this expectation might not hold because the crosscutting modularity may improve the overall understanding of the design a when compared to pure OO models. This would lead to the following null and alternative hypotheses:

Null Hypothesis 3, $H_{3\text{-}0}$: The misinterpretation rate (MisR) in AO models is equal or higher in AO models than in OO models.
***$H_{3\text{-}0}$*:** MisR(AO) $\geq$ MisR(OO)
***Alternative Hypothesis 3, $H_{3\text{-}1}$*:** The misinterpretation rate in AO models is lower than in OO models.
***$H_{3\text{-}1}$*:** MisR(AO) < MisR(OO)

3.3 Experiment Design

Selection of subjects. Subjects (18 students and 8 professionals) were selected based on two key criteria: the level of theoretical knowledge and practical experience related to software modeling and programming. The subjects studied in educational systems that place a high value on key principles of software modeling and programming. In addition, the subjects were exposed to more than 120 hours of courses (lectures and laboratory) exclusively dedicated to software design, software modeling, OO programming, and AO software development. It can be considered they underwent an intensive modeling-specific and programming training. As far as practical knowledge is concerned, the main selection criterion was that subjects had, at least, 2 years of experience with software modeling and programming acquired from real-world project settings.

Paired comparison design. All subjects were submitted to two treatments (AO and OO modeling) to allow us to compare the matched pairs of experimental material. Each treatment had a questionnaire with five multiple-choice questions. The first treatment had only questions with AO models while the second one had only questions with OO models. The subjects were assigned *randomly* and *equally* distributed to these treatments so that the effects of the order could be discarded. Therefore, the experimental design of this study is by definition a *balanced design*.

As the subjects were submitted to two treatments, an ever-present concern was the information that the subject could gain from the first treatment to perform the experiment with the second treatment. To minimize the "gain in information," some experimental strategies [4][5] were followed. First, the models used in the study were fragments of class and sequence diagrams from realistic, industrial design models of different application domains. Hence, the subjects had no prior information and no accumulated knowledge about the semantics of the model elements. Second, each question had a class and sequence diagram representing different functionalities of a software system. Third, each pair of structural and behavioral models had different kinds of inconsistencies (Section 2.2), and the meanings of their elements were completely different. Therefore, we can assume that the performance of subjects was not influenced by the treatments of previous questions.

*Tasks***.** In both treatments, the subjects received a pair of corresponding class (structural) and sequence (behavioral) diagrams. They were asked how they would implement particular classes (or aspects) based on these diagrams. That is, rather than stimulated to review or inspect the diagrams, the subjects were encouraged to implement particular model elements (classes or aspects). The goal is to identify how developers would deal with inconsistencies in the context of concrete software engineering tasks. The subjects should choose, then, the most appropriated implementations between the five possible answer options. In each question, the subjects were required to register the time invested to answer the question ("start time" and "end time"). They were also stimulated to justify their answers on the answer sheet. In total, ten questions were answered. After the experiment, the subjects were also interviewed to clarify the results.

*Objects***.** In the questions of the first treatment, the OO class diagram had, on average, 7 classes and 8 relationships, while in the second treatment the questions had an AO class diagram with, on average, 5 classes and 2 aspects, and 8 relationships. The corresponding AO and OO sequence diagrams had, on average, 5 objects and 15 messages between the objects (and/or aspects). Each pair of OO or AO diagrams had two kinds of inconsistencies. The inconsistencies were always related to contradictions between the class and sequence diagrams. That is, there was conflicting information between those diagrams, as the examples given in Section 2.2. Considering the answer options in each question, they were planned according to the following schema. The first answer option is according to the class diagram while the second one is just according to the sequence diagram. The third answer option is based on the combination of the information presented in both diagrams. The fourth one is incorrect considering all two diagrams. All questions had a fifth answer option where the subjects could indicate that an inconsistency was detected in the models. The subjects were encouraged to carefully explain their answers. Further details of the experimental design can be found in [9].

3.4 Variables and Quantification Method

The independent variable of this study is the choice of the modeling language. It is nominal and two values can be assumed: AO modeling and OO modeling. These variables describe the treatments, and we investigate their impacts on following dependent variables.

Inconsistency detection rate (Rate) and Inconsistency detection effort (Effort). The *Rate* variable is intended to measure the overall rate of inconsistencies detected by all subjects (RQ1). It represents the ratio of the number of subjects that detect inconsistencies in a question divided by the number of subjects that answer the question without notifying the presence of inconsistency. The *Effort* variable represents the mean of time (minutes) spent by the subjects to detect inconsistencies in a question (RQ2). Note that subjects detect inconsistencies when they explicitly indicate that they are unable to achieve a suitable implementation from the contracting diagrams.

Misinterpretation rate (MisR). This variable represents the degree of variation of the answers (RQ3). That is, it measures the concentration of the answers over the four possible alternatives (the fifth alternative represents the detection of inconsistency). Our concern is if the differences in (un)detected inconsistency affects the design interpretation of the subjects. An undetected inconsistency is not necessarily problematic [3] if all subjects have the same interpretation. For example, if the 26 subjects have the same answer (e.g., the alternative “A”) for a question, then the inconsistencies in the diagrams did not lead to misinterpretations (MisR = 1). On the other hand, if the developers’ answers spread equally over the four alternatives, then the inconsistencies cause serious misinterpretations (MisR = 0). That is, the misinterpretation rate is 0 if answers are distributed equally over all options, and 1 if the answers are concentrated only one answer option. According to [3], this variable can be measured as follows.

$$MisR(k_0, \dots, k_{K-1}) = 1 - 2\,\frac{\sum_{0 \le i < K} k_i i}{N(K-1)}$$

Where:
K: The number of alternatives for a question

k_i: The number of times alternative i was selected, where $0 \leq i < K$ and
(for all i : $0 \leq i < K - 1$: $k_i \geq k_{i+1}$)
N: The sum of answers over all alternatives: $N = \sum_{0 \leq i < K} k_i$

3.5 Operation

Preparation phase. The subjects (students and professionals) were not aware of the research questions (and hypotheses) of our study in order to avoid biased results. The motivation of the students was to gain extra points for their grade. The results obtained by the students had no effect on their grade; instead, their dedication and quality of the justifications of the sheet and interviews. The professionals received the same questions as a printable questionnaire. All subjects received a refresher training to be sure of their familiarity with the modeling concepts used in the study.

Execution phase. The experiment tasks were run within two courses at two different Brazilian universities (PUC-Rio and UFBA). Both runs were carried out in a classroom following typical exam-like settings. However, because of time constraints and location, the professionals run the experiment in their work environment. However, the experiment was carefully controlled. All subjects received 10 questions and the answer sheets. It is important to point out that there was no time pressure for the subjects, but they were rigorously supervised to correctly register the time. Therefore, we are confident that the time was recorded properly. For clarification reasons, the subjects were encouraged to justify their answers. After finishing the experiment, the subjects filled out a questionnaire to collect their background, i.e. their academic background and work experience. Observational studies were conducted to improve understanding how the tasks in the experiment were performed,. This allowed a more effective observation and monitoring of the tasks of the subjects. To obtain an additional feedback from the subjects, they were also encouraged to write down the rationale used to answer the questions.

Interview phase. Additionally, a semi-structured interview approach [5] was performed, which followed a funnel model, i.e. one initial open question was presented and followed by more specific ones. It was organized in topics with open and close questions in such a way that qualitative evidence on the research questions could be gathered. An interview guide was created based on the authors' experience and the study design. The interviews were recorded and transcribed into text. All subjects were selected for interviews. Each interview lasted from 30 to 55 minutes, depending on how talkative the subjects were.

4 Experimental Results

4.1 RQ1: Detection Rate in AO and OO Models

Descriptive Statistics. The first research question investigates if developers detect more (or less) inconsistencies in AO models or OO models. Developers detected more inconsistencies in OO models than AO models. The superior detection rate in OO models can be explained comparing means and medians (Table 1). Developers detect, on average, by about 43.24 percent more inconsistencies in OO models than AO models, i.e. a mean of 0.37 (AO) compared with a mean of 0.53 (OO). The difference observed between the medians also favors the OO models. This comprises a superiority of 42.85 percent in the number of the cases in which developers reported to be unable to provide an implementation. The results suggest that OO models enable developers to identify more inconsistencies than AO models. This contradicts somehow the intuition that the improved modularity of AOM helps developers to localize inconsistencies.

Hypothesis Testing. Since the Shapiro-Wilk and Kolmogorov-Smirnov normality tests [1] indicates that the data are normally distributed, the paired t-test is applied to test H_1. The collected *t-statistic* is 4.03 with the *p-value* = 0.01 (Table 1). This small *p-value* (< 0.05) indicates the first null hypothesis ($H_{1\text{-}0}$) can be rejected. This implies that the average difference of the detection rate in AO and OO models is not zero. Therefore, there is strong evidence (at the 0.05 level significant) that developers detect more inconsistencies in OO models than in AO models. The mean differences between the pairs of AO and OO models indicate the direction in which the result is significant. For example, considering the varying detection rate for AO and OO models, the mean difference is negative (-0.16). This implies that the detection rate in AO models was statistically lower than in OO models. Moreover, the non-parametric Wilcoxon test is applied to eliminate any threat related to statistical conclusion validity. The low value of the p-value = 0.031 collected (< 0.05) also confirmed the aforementioned conclusion. Therefore, we can reject the null hypothesis $H_{1\text{-}0}$.

Table 1. Descriptive statistics and Stastical tests for measures

Variables	Treat.	Mean	St Dev	Min.	25th	Med.	75th	Max	%diff	Wilcoxon	Paired t-test		
										p-value	*t*	*p*	*MD*
Detection	AO	0.37	0.09	0.23	0.29	0.35	0.46	0.54	43.24	0.031	4.03	0.01	-0.16
	OO	0.53	0.11	0.38	0.42	0.5	0.67	0.69					
Effort	AO	5.28	1.67	4	4.08	4.22	7	7.8	19.69	0.033	3.1	0.03	-1.48
	OO	6.32	1.57	4.33	5.06	6.08	7.71	8.65					
MisR	AO	0.51	0.07	0.38	0.45	0.52	0.57	0.58	37.25	0.029	2.94	0.04	-0.19
	OO	0.7	0.07	0.62	0.64	0.69	0.77	0.81					

*with 4 degree of freedom, a significance level of α = 0.05, MD: mean difference, p: p-value, St Dev: standard deviation

4.2 RQ2: Detection Effort in AO and OO Models

Descriptive Statistics. The second research question investigates the effort that developers should invest to detect inconsistencies in AO and OO models. Developers spend more effort to detect inconsistencies in OO models than AO models. The mean of detection effort is 5.28 (minutes) in AO models and 6.32 in OO models. This comprises a representative increase of 19.69 percent against plain UML models. This lower effort in the use of AOM is also observed comparing the medians. The detection effort ranges from 4.22 (minutes) in AO models to 6.08 in OO models, which represents an increase of 44.07 percent in the latter case. This phenomenon confirmed our initial intuition that the superior modularity of AO models would accelerate the inconsistency detection. In fact, during the interviews, the subjects (18) reported that the manifestation of inconsistencies in crosscutting relations made the implementation to be prohibitive. Hence, the subjects reported more quickly in the AO model than in OO models. We noticed they were keener to match and contrast the structural and behavioral information governing the crosscut relations. Therefore, developers often report conflicting crosscutting relations as the reason for not progressing towards the implementation. This implies that although developers detect fewer inconsistencies in AO models, they spend less effort to localize them.

Hypothesis Testing. Since the Shapiro-Wilk and Kolmogorov-Smirnov normality tests [1] indicate that the data are normally distributed, the paired t-test is applied to test H_2. The collected *t-statistic* is 3.1 with the *p-value* = 0.03 (Table 1). This small *p-value* (< 0.05) indicates the second null hypothesis ($H_{2\text{-}0}$) can be rejected. This suggests that the average difference of the inconsistency detection effort in AO and OO models is not zero. Thus, there is strong evidence (at the 0.05 level significant) that developers invest more effort to detect inconsistencies in OO models than in AO models. The detection effort in AO and OO groups assumes a negative value for the mean difference (-1.48), while the p-value (0.03) is less than 0.05. This implies that detection effort in OO models is statistically higher than in AO models. Moreover, the non-parametric Wilcoxon test is applied to eliminate any threat related to statistical conclusion validity. The low value of the p-value collected (0.033) also confirmed the previous conclusion. Therefore, we can reject the null hypothesis $H_{2\text{-}0}$.

4.3 RQ3: Misinterpretation Rate in AO and OO Models

Descriptive Statistics. The third research question investigates whether AO models lead to a higher or lower misinterpretation rate than OO models. Table 1 shows the descriptive statistics to the misinterpretation measures of AO and OO models. Recall that MisR varies between 0 and 1, and that MisR = 1 indicates that developers do not have misinterpretation. On the other hand, MisR = 0 indicates that the developers' answers spread equally over the four different alternatives, which represent the most serious misinterpretations. OO models cause less misinterpretation (higher MisR value) than AO models. The misinterpretation rate is 37.25 percent lower in OO models; the mean is 0.51 in AO groups against 0.7 in OO groups. This upward trend is also observed in the medians: 0.52 in AO models against 0.68 in OO models, comprising an increase of 32.69 percent. The results suggest that the presence of inconsistencies in AO models entails a higher detrimental impact on model interpretation by developers than in OO models. Our initial expectation that by using contradicting AO design models would increase the number of diverging interpretations was confirmed. During the interviews and examining the answer sheets, the subjects (22) reported that the need to scan all join points affected by the aspects increased the likelihood of different interpretations.

Hypothesis Testing. We analyze the strength of the aforementioned result testing H_3 as follows. As in the previous analysis, the paired t-test is applied to test H_3 as the measures assume a normal distribution. Table 1 shows the pairwise p-values and mean differences across pairs for each measure. As the mean difference is negative (-0.19) and p-value (0.04) is less than 0.05, we can conjecture that there is significant evidence that the number of diverging interpretations in AO models is statistically higher than in OO models. We also applied the non-parametric Wilcoxon test to check this conclusion. The p-value (0.029) also assumed a low value ($p < 0.05$). Therefore, as the p-value is less than 0.05, and the mean difference is negative, we can conclude that: there is evidence that the MisR in AO models is significantly lower than in OO models. Therefore, we reject the null hypothesis $H_{3\text{-}0}$.

4.4 Discussion

We have identified five outstanding findings from the answer sheets, interviews, and observational study.

1) Higher Aspect Quantification and Inconsistency Detection. First, aspects with higher quantification [10] harmed inconsistency detection (RQ1) and the model interpretation (RQ3) by developers. We observed that when an aspect had six crosscutting relationships and, therefore, affected multiple join points (11, in this case), the subjects spend more time on performing global reasoning. The analysis of several aspect effects in the structural diagrams made developers often to neglect the analysis of behavioral interactions at each specific join point in the behavioral diagrams. According to the interviewees, this effect distracts away developers from observing possible inconsistencies between the structural and behavioral views. We observed, for example, that the inconsistency detection rate in OO models was 71 percent higher than in AO models when the latter were composed of aspects with high quantification; in these circumstances, the mean in OO models was 0.65 compared to 0.38 in AO models. We noticed that 20 subjects explicitly reported that they felt distracted by the presence of high density of crosscutting relationships among the model elements.

2) Overlapping Information about Crosscutting Relationships. Conversely, we observed that the subjects tended to detect more quickly inconsistencies in AO models when the scope of aspect pointcuts was narrow. In these cases, developers invested effort in only confronting structural and behavioral information about the crosscutting relations. According to the subjects, they could observe inconsistencies more quickly in AO models because structural diagrams often express the type of an advice (i.e. before, after or around), which is also a behavioral information that is present in the sequence diagram. Then, they could easily identify inconsistencies between: (i) the types of advices in the class diagram, and (ii) when a particular message was being advised by the aspect in the sequence diagram.

3) Crosscutting Relationships and Diverging Mental Models of the "Big Picture." Data analysis suggests that uniform interpretation of AO models by different developers is harder to achieve than in OO models. The subjects had difficulties to create a "big picture" view from the conflicting class and sequence diagrams. This view represents a "mental model" reflecting how software developers perceive the problem, think about it, and solve it by producing the expected code from the diagrams. This understanding shapes the actions of the developers and defines the approach to guide the design realization in the code. In particular, the developers apparently had diverging mental models when the model inconsistencies were sourced in the cross-cutting relationships. In these cases, developers came up with very different solutions for realizing crosscutting relationships in the code. They provided different answers on which and when the advice should affect the base model elements. Consequently, the communication from designers to programmers seems to be more sensitive to inconsistencies in aspect-oriented models.

4) The Level of Model Detail Matters. Developers usually consider the sequence diagrams as the basis for the design implementation. Note that in this case, the developers do not report the presence of inconsistency. This can be explained for some reasons. First, sequence diagrams are less abstract than class diagrams. This leads developers to rely on the behavioral diagrams than structural diagrams. Second, sequence diagrams are closer to the final implementation; hence, developers become confident that the information present on it is the correct one compared with the class diagram. As a result, it means that when models are used to guide the implementation of design decisions, inconsistencies in behavioral diagrams have a superior detrimental effect than those in class diagrams.

5) Identifying Fewer Inconsistencies in Less Time. Developers identify fewer inconsistencies in AO models than in OO models. However, they spend less effort to detect it in AO models. During the interviews, it was possible to observe that the main reason why developers stop in AOM and go ahead in OOM is that inconsistencies in AOM cause more severe doubts than in OOM. Hence, developers do not feel comfortable with using their experience to overcome the inconsistency problems given the problem at hand. Note that the subjects identify fewer inconsistencies in AOM not because they spent less time, but because it is seen as a "wicked problem." Thus, the developers may be more afraid of dealing with problems in AO models rather than OO models. Finally, the results suggest that developers might insert more defects into code by using AO models. This can be motivated for two reasons: (1) low inconsistency detection, and (2) high disagreement on design interpretations.

5 Threats to Validity

Internal Validity. Inferences between our independent variable and the dependent variables are internally valid if a causal relation involving these two variables is demonstrated [5]. Our study met the internal validity because: (1) the temporal precedence criterion was met; (2) the covariation was observed, i.e. the dependent variables varied accordingly when the independent changed; and (3) there is no clear extra cause for the detected covariation. Our study satisfied all these three requirements for internal validity.

External Validity. It refers to the validity of the obtained results in other broader contexts [5]. Thus, we analyzed whether the causal relationships investigated during this study could be held over variations in people, treatments, and other settings. Some characteristics that strongly contributed to this were identified. First, the subjects used: (1) a practical AOM technique to perform the tasks; and (2) the design models were fragments of real-world models. Second, the reported controlled experiment was rigorously performed, in particular, when compared with controlled experiments previously reported [3].

Construct Validity. It concerns the degree to which inferences are warranted from the observed cause and effect operations included in our study to the constructs that these instances might represent. All variables of this study were quantified using a suite of effort metrics or indicators that were previously defined and independently validated in experiments of inconsistency detection [2][3]. Moreover, the concept of effort used throughout our study is well known in the literature [8] and its quantification method was reused from previous work [2][3]. Therefore, we are confident that the quantification method used is correct, and the quantification was accurately performed.

Statistical Conclusion Validity. Experimental guidelines were followed to eliminate this threat [5]: (1) the assumptions of the statistical tests (paired t-test and Wilcoxon) were not violated; (2) collected datasets were normally distributed; (3) the homogeneity of the subjects' background was assured; (4) the quantification method was properly applied; and (5) statistic methods were used. The Kolmogorov-Smirnov and Shapiro-Wilk tests [1] were used to check how likely the collected sample was normally distributed.

6 Related Work

Aspect-oriented modeling is a very active research field [7][17]. However, there is little related work focusing on the quantitative and qualitative assessment of AOM. The current AOM literature does highlight the importance of performing empirical studies [8]. However, none of them empirically investigate the research topics addressed in our research questions. Research has been mainly carried out in two areas: (1) defining new AOM techniques [7][17], and (2) proposing new weaving mechanisms [13]. Several authors have proposed new modeling languages, focusing on the definition of constructs, such as <<aspect>> and <<crosscut>>. These constructs represent concepts of aspect-orientation as UML-based extensions [7][17][18][19][20]. For example, Clarke and Baniassad [7] make use of UML templates to specify aspect models. On the other hand, the chief motivation of some works is to provide a systematic method for weaving aspect and base models (e.g. [12][13]). For example, Klein and colleagues [13] present a semantic-based aspect weaving algorithm for hierarchical message sequence charts (HMSC). They use a set of transformations to weave an initial HMSC and a behavioral aspect expressed with scenarios. Moreover, the algorithm takes into account the compositional semantics of HMSCs.

Empirical studies of AOM (such as [6]) have not been conducted, in particular, in the context of modeling inconsistencies (or defects). Only the literature on OO modeling does highlight that empirical studies have been done on identifying defects in design models [2][3]. Lange and Chadron [3] investigate the effects of defects in UML models. The two central contributions were: (1) the description of the effects of undetected defects in the interpretation of UML models, and (2) the finding that developers usually detect more certain kinds of defects than others. In conclusion, there are two critical gaps in the current understanding about AOM: (1) the lack of practical knowledge about the developers' effort to localize inconsistencies, and (2) the lack of empirical evidence about the detection rate and misinterpretations when understanding AO models.

7 Concluding Remarks

This paper reports an empirical investigation about the impact of AOM on the inconsistency detection rate, the effort to detect inconsistencies, and the misinterpretation rate. We observed that developers detected fewer inconsistencies in AO models than OO models. The reason is that they got more distracted by the global reasoning motivated by the presence of crosscut relations and overlooked the negative effects of existing model inconsistencies. According to the subjects, a complex crosscutting collaboration between modules led developers to unconsciously make more *implicit assumptions* about the correct design decisions. As a consequence, aspects with higher quantification were the cause of a lower detection rate of inconsistencies.

Second, developers spent less effort using AO models to detect each inconsistency than in OO models. This was mainly due to a higher degree of overlapping information in structural and behavioral views of AOM. Third, the software developers presented a superior rate of misinterpretation in AO models, mostly thanks to the additional number of modeling concepts (e.g., crosscut relationships and aspects). They also had to examine all join points affected by the aspects. This extra analysis increased the degree of disagreement by developers while interpreting AO models and producing the code. It is important to highlight that all the aforementioned findings were independent of inconsistencies being assessed.

References

1. Levine, D., Ramsey, P., Smidt, R.: Applied Statistics for Engineers and Scientists. Duxbury (1999)
2. Lange, C., Chaudron, M.: An Empirical Assessment of Completeness in UML Designs. In: 8th Empirical Assessment in Software Engineering 2004, pp. 111–121 (2004)
3. Lange, C., Chaudron, M.: Effects of Defects in UML Models – An Experimental Investigation. In: International Conference on Software Engineering 2006, Shangai, China, pp. 401–410 (May 2006)
4. Kitchenham, B., et al.: Evaluating Guidelines for Reporting Empirical Software Engineering Studies. Empirical Software Engineering 13(1), 97–112 (2008)
5. Wohlin, et al.: Experimentation in Software Engineering: an Introduction. Kluwer Academic Publishers, Norwell (2000)
6. Farias, K., Garcia, A., Whittle, J.: Assessing the Impact of Aspects on Model Composition Effort. In: Aspect-Oriented Software Development 2010, Saint Malo, France, pp. 73–84 (2010)
7. Clarke, S., Banaissad, E.: Aspect-Oriented Analysis and Design the Theme Approach. Addison-Wesley, Upper Saddle River (2005)
8. France, R., Rumpe, B.: Model-Driven Development of Complex Software: A Research Roadmap. In: Future of Software Engineering at ICSE 2007, pp. 37–54 (2007)
9. Evaluating the Impact of Aspects on Inconsistency Detection Effort: a Controlled Experiment (2012), `http://www.les.inf.puc-rio.br/opus/models2012-aom`
10. Filman, R., Friedman, D.: Aspect-Oriented Programming is Quantification and Obliviousness. In: RIACS (2000)

11. OMG, Unified Modeling Language: Infrastructure, v2.2, Object Management Group (February 2010)
12. Whittle, J., Jayaraman, P.: Synthesizing Hierarchical State Machines from Expressive Scenario Descriptions. ACM TOSEM 19(3) (January 2010)
13. Klein, J., Hélouët, L., Jézéquel, J.: Semantic-based Weaving of Scenarios. In: 5th Aspect-Oriented Software Development, Bonn, Germany (March 2006)
14. AspectJ (2011), `http://www.eclipse.org/aspectj`
15. Dobing, B., Parsons, J.: How UML is Used. Communications of the ACM 49(5), 109–113 (2006)
16. Brun, Y., Holmes, R., Ernst, M., Notkin, D.: Proactive Detection of Collaboration Conflicts. In: 8th SIGSOFT ESEC/FSE, Szeged, Hungary, pp. 168–178 (2011)
17. Losavio, F., Matteo, A., Morantes, P.: UML Extensions for Aspect Oriented Software Development. Journal of Object Technology 8(5), 85–104 (2009)
18. Wimmer, M., Schauerhuber, A., Kappel, G., Retschitzegger, W., Schwinger, W., Kapsammer, E.: A survey on UML-based aspect-oriented design modeling. ACM Computing Survey 43(4), 1–33 (2012)
19. Aldawud, O., Elrad, T., Bader, A.: A UML Profile for Aspect- Oriented Software Development. In: Workshop on Aspect-Oriented Modeling at AOSD (2003)
20. Chavez, C., Lucena, C.: A Metamodel for Aspect-Oriented Modeling. In: Workshop on Aspect-Oriented Modeling with the UML, at AOSD 2002, Netherlands (April 2002)

On Integrating Structure and Behavior Modeling with OCL

Lars Hamann, Oliver Hofrichter, and Martin Gogolla

University of Bremen, Computer Science Department
Database Systems Group, D-28334 Bremen, Germany
{lhamann,hofrichter,gogolla}@informatik.uni-bremen.de
http://www.db.informatik.uni-bremen.de

Abstract. Precise modeling with UML and OCL traditionally focuses on structural model features like class invariants. OCL also allows the developer to handle behavioral aspects in form of operation pre- and postconditions. However, behavioral UML models like statecharts have rarely been integrated into UML and OCL modeling tools. This paper discusses an approach that combines precise structure and behavior modeling: Class diagrams together with class invariants restrict the model structure and protocol state machines constrain the model behavior. Protocol state machines can take advantage of OCL in form of OCL state invariants and OCL guards and postconditions for state transitions. Protocol state machines can cover complete object lifecycles in contrast to operation pre- and postconditions which only affect single operation calls. The paper reports on the chosen UML language features and their implementation in a UML and OCL validation and verification tool.

Keywords: Structure modeling, Behavior modeling, UML, OCL, Protocol state machine, State invariant, Guard, Transition postcondition.

1 Introduction

Executable UML [23] is designed to specify a system at a high level of abstraction, independent from specific programming languages and decisions about the implementation. Executable UML follows the ideas of the Shlaer-Mellor methodology, which separated concerns about the structure [34] and the behavior [33] of a system to be developed. It is defined as a profile of the Unified Modeling Language (UML) [26]. Executable UML models are testable, and can be compiled into less abstract programming languages to target a specific implementation. Executable UML supports model-driven development (MDD) through specification of platform-independent models. The approach proposed in this paper follows these ideas.

When using Executable UML, a system is decomposed into multiple modeling sub-languages: A class diagram defines the system structure in terms of the classes and associations; a state machine defines the states, events, and state transitions for a class instance; an action language defines the actions or operations that perform processing on model elements; the system behavior is determined by the state machines and the operations realized in the action language.

Our tool USE (UML-based Specification Environment) supports the development of class diagrams by validating OCL class invariants and operation pre- and postconditions [7,8,19]. Recently, the tool was extended with an action language [3] which is based on the Object Constraint Language (OCL) [27,36]. The present contribution explains our support for state machines in order to complete the description of behavior. Within our tool, we integrate class diagram validation with UML protocol machine validation on the basis of OCL state invariants and OCL guards and postconditions for transitions. In contrast to Executable UML, our approach extends OCL in order to express actions and operation implementations, but does not need to define a separate action language.

The need for integrating structure and behavior modeling in the OCL context arose from monitoring running Java applications in terms of UML class diagrams and OCL constraints and our state machine approach. In [12] we describe the monitoring of a non-trivial Java application with constraints. Other applications of our state machine implementation include middle-sized example models.

The rest of this paper is organized as follows. Section 2 introduces with a running example the main state machine features which we employ on the type level (at design time). Section 3 puts the state machine features which we handle in the context of UML and our implementation. In Sect. 4, model validation of state machines in connection with class diagrams is discussed on the instance level (at runtime). Section 5 connects our contribution with related work, before we conclude in Sect. 6.

2 Structure and Behavior at Design Time by Example

Our running example describes a digital support system for a library. The structural system requirements are shown in form of a UML class diagram in the top of Fig. 1. The system supports the administration of users, book copies, and books represented by respective classes and appropriate attributes. Two associations can establish object connections: the association `Borrows` between the classes `User` and `Copy` is meant to express that a `User` object has currently borrowed a `Copy` object, and the association `BelongsTo` between the classes `Copy` and `Book` expresses that a `Copy` object is an exemplar of a particular `Book` object. Further properties are specified by restricting multiplicities, role names (in the example, class names with lower first letter) and invariants (e.g., uniqueness requirements for the attributes `name`, `signature`, and `title`, as well as a range restriction for the attribute `year`). All classes possess operations for initializing objects. The association `Borrows` can be manipulated from both participating classes through the operations `borrow` and `return`. In order to support easy recognition of operation names, the first letter of the respective class has been added to these names (`borrowU`, `returnU`, `borrowC`, `returnC`). The `return` operations also modify the attribute `numReturns`.

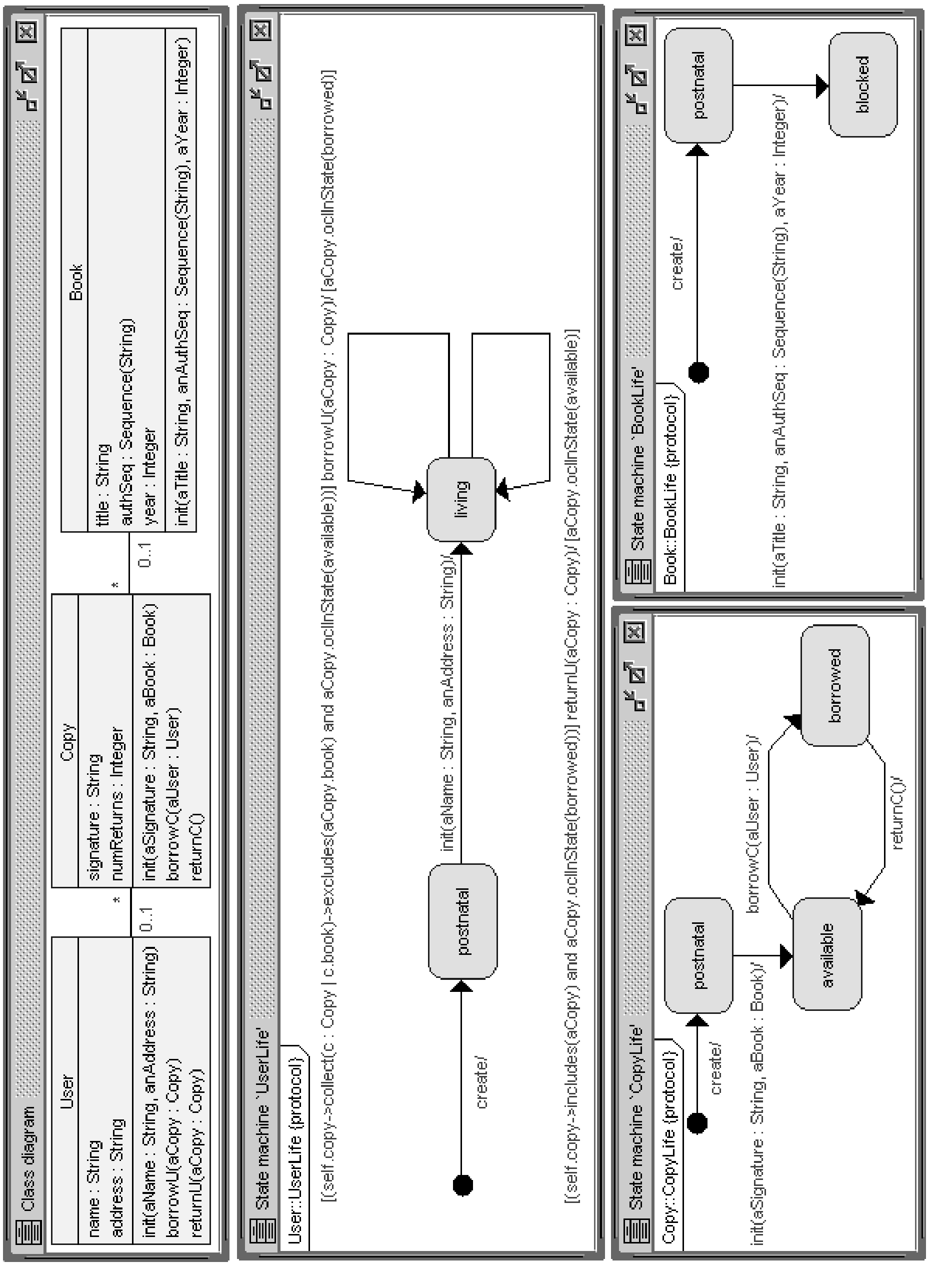

Class diagram
User
name : String
address : String
init(aName : String, anAddress : String)
borrowU(aCopy : Copy)
returnU(aCopy : Copy)
*
0..1
Copy
signature : String
numReturns : Integer
init(aSignature : String, aBook : Book)
borrowC(aUser : User)
returnC()
*
0..1
Book
title : String
authSeq : Sequence(String)
year : Integer
init(aTitle : String, anAuthSeq : Sequence(String), aYear : Integer)
State machine 'UserLife'
User::UserLife {protocol}
create/
postnatal
init(aName : String, anAddress : String)/
living
[(self.copy->collect(c : Copy | c.book)->excludes(aCopy.book) and aCopy.oclInState(available))] borrowU(aCopy : Copy)/ [aCopy.oclInState(borrowed)]
[(self.copy->includes(aCopy) and aCopy.oclInState(borrowed))] returnU(aCopy : Copy)/ [aCopy.oclInState(available)]
State machine 'CopyLife'
Copy::CopyLife {protocol}
create/
postnatal
init(aSignature : String, aBook : Book)/
available
borrowC(aUser : User)/
borrowed
returnC()/
State machine 'BookLife'
Book::BookLife {protocol}
create/
postnatal
init(aTitle : String, anAuthSeq : Sequence(String), aYear : Integer)/
blocked

Fig. 1. Example System Requirements for Structure and Behavior (Design Time)

The behavioral system requirements are shown in the bottom of Fig. 1 as UML protocol state machines possessing states and transitions. For every class, the valid object lifecycles are depicted, which restrict the order of creation events and operation calls. As a central means to make the model precise, OCL is used in various places: States are described by state names and state invariants in form of boolean OCL expressions; transitions include (a) the triggering create or call event, (b) a guard in form of a boolean OCL expression asserting that the transition only takes places when the guard holds, and (c) a postcondition in form of a boolean OCL expression asserting that the transition only takes place in the case that after the transition the postcondition holds. Traditionally, the notion guard is used in connection with state machines; however, because of the symmetric behavior of the guard and postcondition, the guard may also be called transition precondition.

The state invariants may optionally be shown in the protocol state machine diagrams, however, we have suppressed them here. For example for the class `Book`, the two proper, non-pseudo states possess the following state invariants.

```
postnatal [title.isUndefined and authSeq->isUndefined and
           year.isUndefined and copy->isEmpty()]
blocked   [title.isDefined and authSeq->isDefined and year.isDefined]
```

In state `postnatal` (after `create`), all attributes must be undefined and the book must not be linked to any copy. In state `blocked` (after a call to the initialization operation `init`), all attributes are defined, but note that no statement about the linked copies is made, because there may or may not be copies for that book in the library (either `copy->notEmpty()` or `copy->isEmpty()` may hold).

The transitions are either labeled with the `create` event which brings the respective object into life or with an event which calls an operation of the object. The protocol state machine for the class `Book` asserts a finite lifecycle demanding that after object creation only the operation `init` may be called once. The state machine for class `Copy` guarantees that after creation and initialization, the `borrowC` and `returnC` operations switch between the states `available` and `borrowed`. The state machine for the class `User` is the only one employing OCL for transition guards and postconditions. But please be aware of the fact that all states are accompanied by OCL state invariants. Both operations, `borrowU` and `returnU` in class `User` are allowed in state `living`, however, OCL restrictions via transition guards and postconditions apply. The guard (precondition) for `borrowU` guarantees that a user cannot borrow two copies of the same book, for fairness reasons. And the guard asserts that only available, not borrowed copies can be handled with the operation `borrowU`. The postcondition of `borrowU` checks that the copy, which was available before the transition took place, is now unavailable. Conversely, the guard for `returnU` asserts that the copy to be returned belongs to the current user and is indeed a copy in state `borrowed`. The postcondition checks that the parameter copy is indeed `available` after the `returnU` call. Note that these simple example restrictions do not guarantee unproblematic behavior in all possible implementations. The state invariants, guards, and postconditions have been chosen for demonstration purposes.

An implementation on the modeling level of the operations can be realized in our language SOIL (Simple OCL-based Imperative Language) [3]. Such an implementation is indispensable for animating and validating the model. SOIL allows the developer to make system state manipulations with attribute assignments, object and link creation and destruction, and control flow using conditionals, loops, and operation calls. As an example, we show implementations for the operations of the classes `User` and `Copy`.

```
class User -- pre- and postconditions not shown
operations
  init(aName:String,anAddress:String)
    begin self.name := aName;
    self.address := anAddress; end

  borrowU(aCopy:Copy)
    begin aCopy.borrowC(self); end

  returnU(aCopy:Copy)
    begin aCopy.returnC(); end
end

class Copy
operations
  init(aSignature:String, aBook:Book)
    begin self.signature := aSignature; self.numReturns := 0;
    insert (self, aBook) into BelongsTo; end

  borrowC(aUser:User)
    begin insert(aUser, self) into Borrows; end

  returnC()
    begin delete(self.user, self) from Borrows;
    self.numReturns := self.numReturns+1; end
end
```

These operation implementations allow the developer to build up simple or complex test states and scenarios with call sequences easily. Consequently, model properties like consistency or the reachability of protocol states can be checked with scenarios constructed with SOIL statements. The SOIL command sequence in the upper right side of the forthcoming Fig. 3 is an example for such a test scenario. The validity of model properties formulated in OCL as class invariants, operation pre- and postconditions, state invariants, and transition pre- and post-conditions is checked against these scenarios and by this also against the SOIL implementation given for the operations. When writing down a particular test scenario, the developer will have expectations on particular (class or state) invariants and (operation and transition) pre- and postconditions. These informal expectations are formally checked by the tool USE, and the validation results give detailed feedback to the developer about the possible discrepancy between her expectations and the actual facts: *What you write down doesn't mean exactly* *what you think it means. And when it does, it doesn't have the consequences you expected.* [15, p. XIII]

3 Behavior Modeling with Protocol State Machines

3.1 Protocol State Machines in UML

The UML defines two different kinds of state machines: *Behavioral state machines* and *protocol state machines* [26, p. 535]. As the name suggests, the former can model the behavior of a model element by specifying actions which are linked to state transitions, whereas the latter focus on the specification of correct usage protocols, leaving out concrete actions associated with transitions [26, p. 547]. These protocols can be specified for any model element of type *Classifier* [26, p. 544]. The metamodel for state machines provided by the UML allows to model highly structured state machines composed of, for example, composite states, multiple regions and substate machines. At the current stage, our approach supports only a well-defined subset of these features leaving out mainly concepts to structure state machines, but allowing nearly the same expressiveness. Issues arising from the high structuring possibilities can for example be found in [21]. Next we describe the protocol state machine language as implemented in our work. Starting with the syntactical and semantical rules defined in the UML, we continue by showing the current features supported in our approach and how they are interpreted at runtime.

As other languages for (finite) state machines the core part of the state machines defined by the UML are states and transitions. The UML distinguishes between concrete and pseudo-states [26, p. 536, 549, 559]. A state machine instance cannot have a pseudo-state as its current state after a transition has been completed. Pseudo-states are only traversed during the execution of a transition. One example of such pseudo-states are choice points for a transition. Both kinds of states are derived from the metatype *Vertex* for which directed transitions are defined. Behavioral state machines consist of transitions which need a source and target vertex. In addition, transitions can specify a trigger (e.g., a call event), a guard and an effect, i. e., a behavior [26, p. 536].

As we will see, several parts of state machines can be enriched with additional boolean OCL expressions in order to add additional constraints. States can be enriched with a OCL state invariant which characterizes the state in more detail. The state invariant for a given state must be true, if a state machine is in this state. An OCL guard of a transition must be true to be able to execute this transition. For example, this allows to separate two outgoing transitions from one state with the same trigger. In protocol state machines it is also allowed to specify a boolean OCL expression which describes the system state after a protocol transition has been taken. This expression is called a postcondition of the protocol transition.

The initial pseudo-state together with a single outgoing transition marks a concrete state as the default state of the state machine. The transition from the initial state to the default state can only define a behavior and no trigger or guard [26, p. 550]. Furthermore, the initial state, as all other pseudo-states, cannot specify a state invariant, whereas concrete states can.

Transitions inside a protocol state machine are defined by the metaclass *ProtocolTransition* [26, p. 546]. This class extends the transition class of the behavioral state machine and makes some extensions and restrictions. The main restriction for protocol transitions is that they cannot specify an effect, because they specify the usage of a protocol of a class and not its behavior. An effect of a transition is instead specified in a declarative way by means of a postcondition which cannot be specified for ordinary transitions. The trigger of a protocol transition is usually an operation call, but it can also be an event.

When a protocol state machine defines at least one transition, which refers to an operation, a call to this operation is only valid, if there exists a currently valid transition for this call event. If an operation of the owning class is not referred by a protocol state machine, a call to this operation is valid for any state of the state machine [26, p. 549]. The specification of events other than call events inside a protocol state machine defines requirements for the environment using the owning class, stating that the event can only be sent to an instance of the owning class under the current conditions specified by the protocol state machine [26, p. 549]. An additional constraint specified for a transition is usually called a guard, but for protocol transitions the naming is aligned to the area of operations, calling this constraint a precondition.

3.2 Supported Concepts for Behavior Validation

Our approach supports protocol state machines which allows to specify valid call sequences for lifecycles of an instance. A protocol state machine is defined in the context of a class. The concrete syntax of such definitions is shown below.

```
class A
 attributes
   ...
 operations
   ...
 statemachines
    psm ALife -- psm: Protocol State Machine
     states
      s_i:initial
      s_k [ state_invariant_k ]
      ...
      s_n:final
    transitions
      s_src -> { [ pre_cond ] call_event [ post_cond ] } s_trg
      ...
    end
end
```

First, more than one state machine (in the following we use the term state machine to refer to protocol state machines) can be specified for a class. Beside a name, each state machine defines two sections: `states` and `transitions`. The state section contains the definition of the pseudo- and the concrete states. A state machine must define exactly one pseudo-state of type *initial* acting as the entry point of the state machine. As already mentioned, the initial state cannot define any information except a name for the state. Concrete states are defined by their names and an optional state invariant expressed as a boolean OCL expression in the context of the owning class. State invariants will be discussed in detail during the description of the runtime behavior of state machines. Beside the concrete states and the initial pseudo-state, multiple final states can be defined.

The transition section specifies the structure of valid call sequences to the owning class. The textual syntax is aligned to the graphical representation in the state machine diagrams. For transitions, the source (`s_src`) and target state (`s_trg`) separated by an arrow (`->`) are mandatory. Except for the outgoing transition from the initial state, a `call_event` is also mandatory. These call events refer to an operation of the owning class. The call event for the outgoing transition of the initial state can either be left out or must be named `create` because a newly created object in our approach is immediately initialized with instances of all defined state machines for its class. The `call_event` can be surrounded by a pre- and postcondition given as a boolean OCL expression. Like pre- and postconditions for operations they can access the context object (the instance receiving the call event) and the parameter values of the call event. The postcondition can additionally make use of the OCL `@pre` keyword to access the values which were valid when the call event was triggered.

When a USE model containing state machines is loaded, static checks are made. These include checking the uniqueness of state names inside a single state machine and the well-formedness of transitions, i.e., checking that state names and transitions do refer to existing states and operations.

3.3 Protocol State Machines at Runtime

To validate a specified model, our approach allows the developer to instantiate it and observe its behavior. The instantiation can be done in several ways, e.g., by manually manipulating the system state using the graphical user interface or shell commands or by specifying statements in SOIL [3]. If an object of a class is created, which contains state machines[1], it is linked to the corresponding state machine instances. These state machine instances are initialized with the default state, i.e., the state reached by the outgoing transition of the initial state, as their current state.

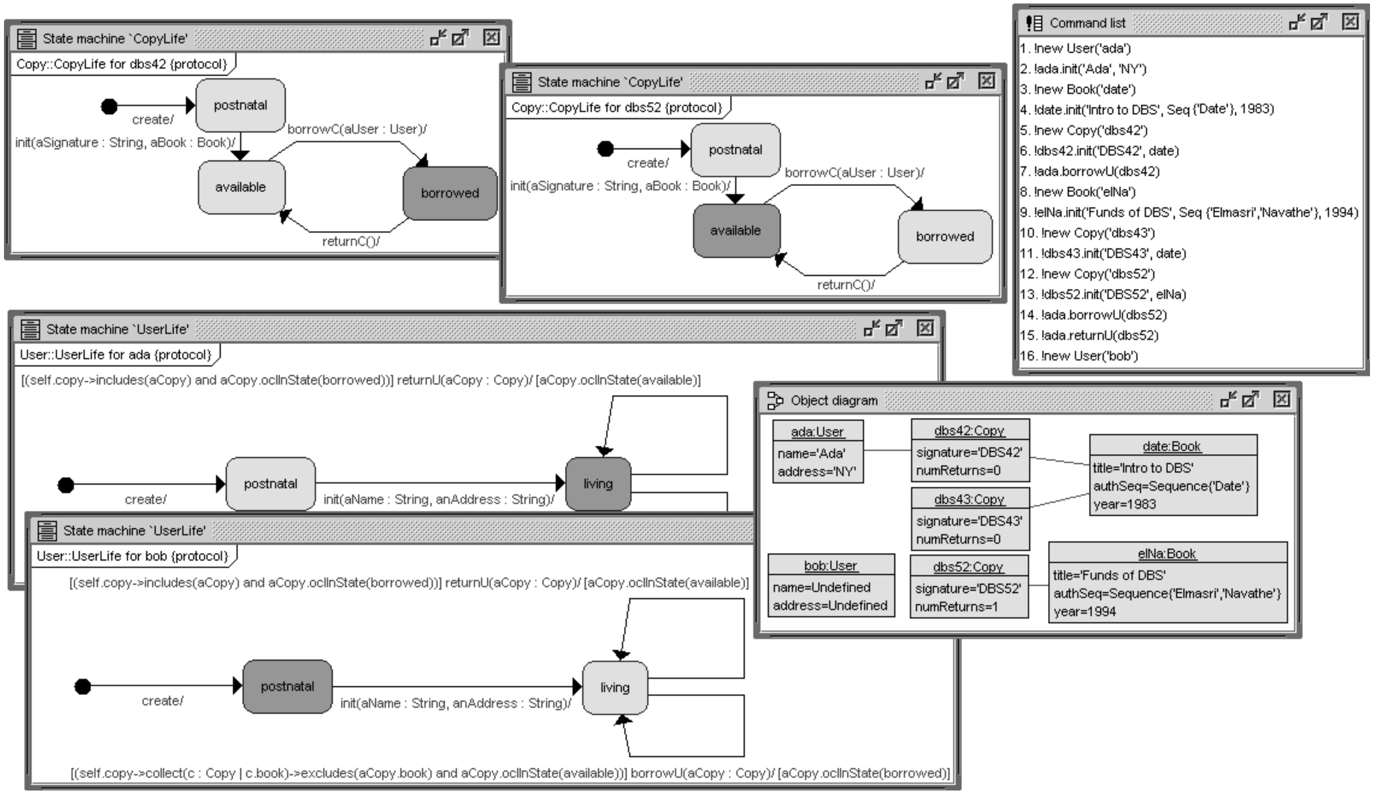
If an operation is called on an object, all state machines, which specify a transition referring to the operation call, are checked for enabled transitions. A

[1] In the following we refer to objects of classes with defined state machines when using the word object.

transition is called *enabled*, if it is an outgoing transition leaving the current state of a considered state machine instance, if it refers to the called operation and if it has a currently valid precondition [26, p. 584]. If at least one enabled transition for each state machine under consideration exists, the operation call is valid. The transition to take is determined after the operation has been executed. This is done by evaluating for each previously enabled transition the postcondition and the state invariant of the target state. For each considered state machine instance there must be exactly one transition fulfilling both conditions. By using this mechanism, we (currently) disregard non-deterministic state machines and executions which are however generally allowed in UML. Otherwise, the operation execution is invalid. The concrete error situation is reported to the user stating that either there exists no valid transition or multiple transitions are currently valid. When a state machine instance is currently in an unstable state, i.e., it is executing a transition, all nested operation call events need to be ignored. Otherwise, a call to another operation on the same object by a called operation could for example change the current state making the previously enabled transition invalid. The modeler can turn on a notification mechanism for such situations.

The explained runtime behavior of state machines lead to valid call sequences respecting state invariants, transition pre- and postconditions, if the state of an object is only modified by operations specified by protocol state machines. However, as we described earlier, a protocol state machine can leave out operations, making them callable at any time. Because these unconsidered operations could also modify the state of an object, it is not guaranteed that a state invariant stays valid while a state machine instance remains in a certain state. Therefore, our approach is able to validate state invariants after any change to the system state, e.g., attribute assignments or link creations. A violation of state invariants is immediately reported to the user, who can then react to the error.

Another unique feature of our approach is the possibility to determine the current state of the state machines by the specified state invariants [11]. For this, the validation of transitions and state invariants can be suppressed. After a system state is constructed without the validation of state machines, the user can invoke the state determination command. The command tries to determine the current state for each state machine instance by evaluating its state invariants. If exactly one state invariant of a state machine instance evaluates to true, the state of this instance is modified. This can, for example, be used, if a given system state needs to be constructed without the execution of operations and afterwards an operation call sequence has to be validated. An application of this mechanism is the USE monitor [10,12] which allows to connect to a running Java application and to retrieve a snapshot of the current application state. When connecting to the application, not all information about previously called operations is available, and therefore the current states must be calculated to obtain the valid state machine configuration.



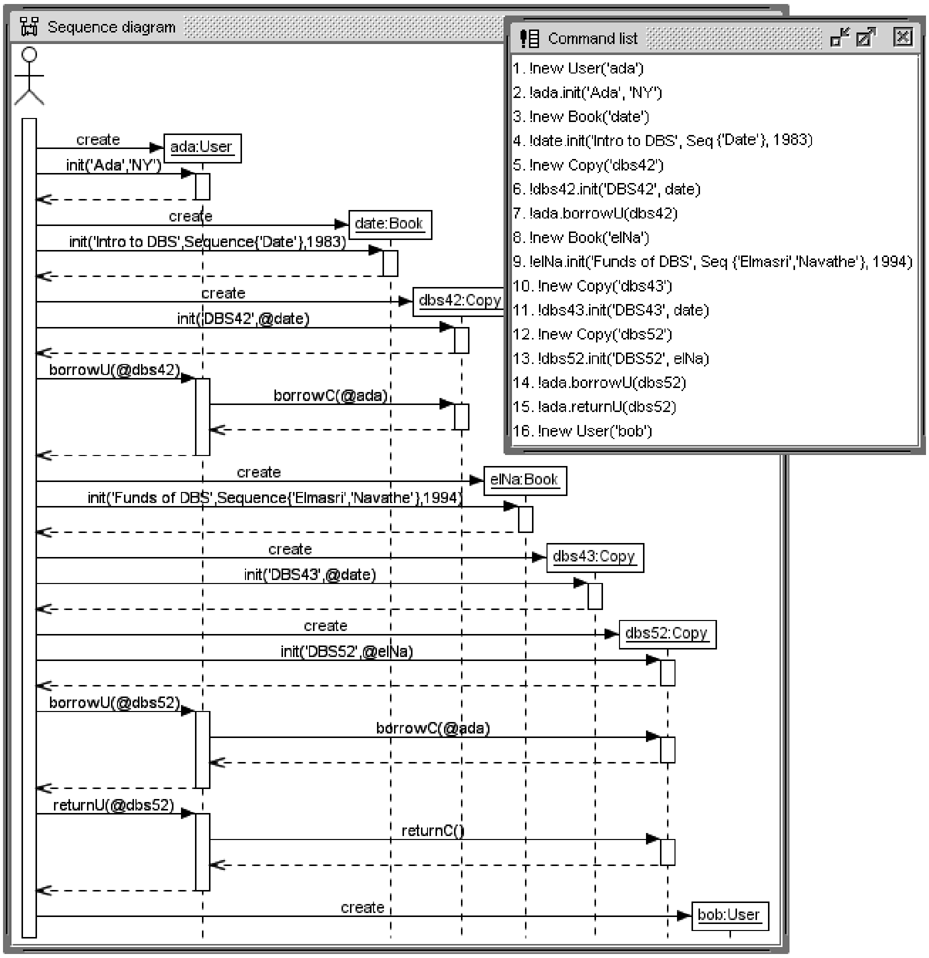
```
Command list
1. !new User('ada')
2. !ada.init('Ada', 'NY')
3. !new Book('date')
4. !date.init('Intro to DBS', Seq {'Date'}, 1983)
5. !new Copy('dbs42')
6. !dbs42.init('DBS42', date)
7. !ada.borrowU(dbs42)
8. !new Book('elNa')
9. !elNa.init('Funds of DBS', Seq {'Elmasri','Navathe'}, 1994)
10. !new Copy('dbs43')
11. !dbs43.init('DBS43', date)
12. !new Copy('dbs52')
13. !dbs52.init('DBS52', elNa)
14. !ada.borrowU(dbs52)
15. !ada.returnU(dbs52)
16. !new User('bob')
```

State machine `CopyLife'
Copy::CopyLife for dbs42 {protocol}
postnatal
create/
init(aSignature : String, aBook : Book)/
borrowC(aUser : User)/
available
borrowed
returnC()/
State machine `CopyLife'
Copy::CopyLife for dbs52 {protocol}
postnatal
create/
init(aSignature : String, aBook : Book)/
borrowC(aUser : User)/
available
borrowed
returnC()/
State machine `UserLife'
User::UserLife for ada {protocol}
[(self.copy->includes(aCopy) and aCopy.oclInState(borrowed))] returnU(aCopy : Copy)/ [aCopy.oclInState(available)]
create/
postnatal
init(aName : String, anAddress : String)/
living
State machine `UserLife'
User::UserLife for bob {protocol}
[(self.copy->includes(aCopy) and aCopy.oclInState(borrowed))] returnU(aCopy : Copy)/ [aCopy.oclInState(available)]
create/
postnatal
init(aName : String, anAddress : String)/
living
[(self.copy->collect(c : Copy | c.book)->excludes(aCopy.book) and aCopy.oclInState(available))] borrowU(aCopy : Copy)/ [aCopy.oclInState(borrowed)]
Object diagram
ada:User
name='Ada'
address='NY'
dbs42:Copy
signature='DBS42'
numReturns=0
date:Book
title='Intro to DBS'
authSeq=Sequence{'Date'}
year=1983
dbs43:Copy
signature='DBS43'
numReturns=0
bob:User
name=Undefined
address=Undefined
dbs52:Copy
signature='DBS52'
numReturns=1
elNa:Book
title='Funds of DBS'
authSeq=Sequence{'Elmasri','Navathe'}
year=1994

Fig. 2. Example Scenario for Structure and Behavior (Runtime)

4 Structure and Behavior at Runtime by Example

This section will explain how to apply the proposed concepts for the example. Whereas Fig. 1 pictures structure and behavior of the library system on a type level (design time), Fig. 2 displays structure and behavior of one system test scenario on the instance level (runtime). The object diagram in the lower right represents the objects, their attribute values and links after the SOIL command sequence in the upper right part of Fig. 3 has been executed. In the left of Fig. 2, the upper two state machine instances show the current protocol state for the `Copy` objects `dbs42` and `dbs52`, respectively. Also in the left, the lower two state machine instances display the current protocol state for the `User` objects `ada` and `bob` in dark grey. Please note, that the state of both `Copy` objects and the state of both `User` objects are different. The state sequence which the `Copy` object `dbs52`



```
1. !new User('ada')
2. !ada.init('Ada', 'NY')
3. !new Book('date')
4. !date.init('Intro to DBS', Seq {'Date'}, 1983)
5. !new Copy('dbs42')
6. !dbs42.init('DBS42', date)
7. !ada.borrowU(dbs42)
8. !new Book('elNa')
9. !elNa.init('Funds of DBS', Seq {'Elmasri','Navathe'}, 1994)
10. !new Copy('dbs43')
11. !dbs43.init('DBS43', date)
12. !new Copy('dbs52')
13. !dbs52.init('DBS52', elNa)
14. !ada.borrowU(dbs52)
15. !ada.returnU(dbs52)
16. !new User('bob')
```

Fig. 3. Sequence Diagram and SOIL Commands for Example Scenario

went through was `postnatal`, `available`, `borrowed` and again `available`. We can conclude this from the executed operation sequence and from the attribute value `1` for attribute `numReturns`. In the shown operation sequence, all OCL restrictions have been checked and no violation occurs: all class invariants, state invariants and transition pre- and postconditions have been evaluated to `true`. Please note, that full OCL support in our approach means that we can relate OCL queries concerning structure with behavioral descriptions, for example, the OCL query in Fig. 2 checks relevant `Copy` properties and these can be compared with the current protocol state and the value of the state invariants.

This scenario can be extended by further operation calls. For example, the `User` object `ada` could try to borrow the `Copy` object `dbs43`. In this situation, the guard for the `borrowU` call on the transition from `living` to `living` would prevent the transition to take place: User `ada` has already borrowed another copy of the `Book` object `date`. On the USE shell, a message will inform about the violation and the fact that the transition should not and will not occur. The following message will be shown.

```
!ada.borrowU(dbs43)
 >> Error: No valid transition available in protocol state machine
 >> 'User::UserLife [current state: living]' for operation call
 >> User::ada.borrowU(dbs43) due to failing transition guard.
```

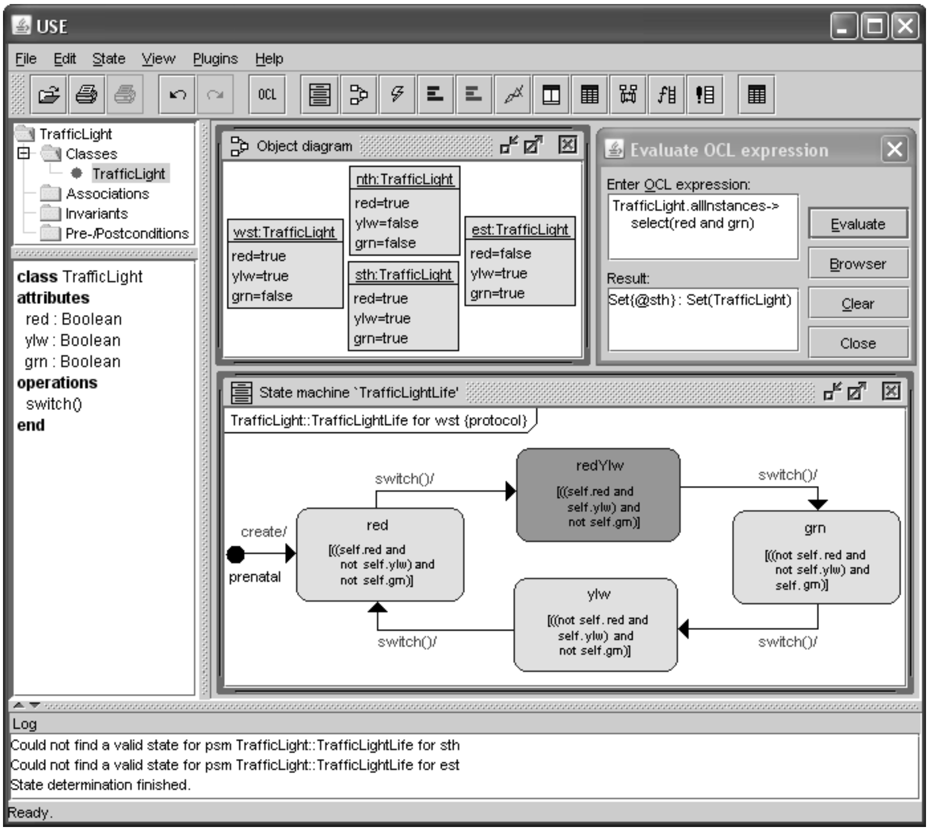
Analogous error messages would be displayed on the shell, if the transition postcondition or the state invariant of the next state would be violated. Summarizing we can say that taking a transition may be aborted due to four possible reasons:

- a failing transition guard (precondition),
- a failing transition postcondition,
- a failing state invariant in the resulting state, and
- non-deterministic transitions, e.g., multiple transitions for the same trigger.

In Fig. 4, another example explains the usage of state invariants and the state determination option. For a `TrafficLight` class with three boolean attributes representing the red, yellow, and green bulbs, a protocol state machine allows the traffic light to step through four phases, where each phase is represented by a single state and a state invariant in form of an OCL expression characterizing the signal in terms of the bulbs.[2] The object diagram shows four test traffic lights equipped with randomly determined attribute values for the bulbs, not all representing valid signal configurations. The attribute values have been modified not by operations, but with direct attribute assignments.

In the log window at the bottom, the result of executing the state determination command is given. This command aims to bring the state machine instances into the state corresponding to their state invariants, if possible. The command

[2] The phases are the phases used in Germany, whereas in other countries, e.g., in Italy, the phases are different.



```
class TrafficLight
attributes
  red : Boolean
  ylw : Boolean
  grn : Boolean
operations
  switch()
end
```

wst:TrafficLight	nth:TrafficLight	sth:TrafficLight	est:TrafficLight
red=true	red=true	red=true	red=false
ylw=true	ylw=false	ylw=true	ylw=true
grn=false	grn=false	grn=true	grn=true

Enter OCL expression:

```
TrafficLight.allInstances->
    select(red and grn)
```

Result:

```
Set{@sth} : Set(TrafficLight)
```

State invariants:

- redYlw: `[((self.red and self.ylw) and not self.grn)]`
- red: `[((self.red and not self.ylw) and not self.grn)]`
- grn: `[((not self.red and not self.ylw) and self.grn)]`
- ylw: `[((not self.red and self.ylw) and not self.grn)]`

```
Could not find a valid state for psm TrafficLight::TrafficLightLife for sth
Could not find a valid state for psm TrafficLight::TrafficLightLife for est
State determination finished.
```

Fig. 4. Example for Usage of State Invariants and State Determination Option

can be issued through an entry in the 'State' menu. For two traffic lights (`sth` and `est`), a valid state fitting one of the four state invariants could not be found; the other state machine instances are moved into a state determined by a state invariant. The displayed state machine instance in the middle belongs to the `TrafficLight` object `wst` and shows that the attribute values (`wst.red=true` and `wst.ylw=true` and `wst.grn=false`) fit to the OCL state invariant expression (`self.red and self.ylw and not(self.grn)`) belonging to the current state `redYlw` shown in dark grey. As our approach supports OCL during all development phases, the complete system state can be inspected with OCL expressions at any point in time. The OCL query expression in the upper right retrieves all present traffic light objects which currently show both `red` and `grn`. The state determination together with OCL querying allows to check positive and negative test cases with respect to structure (objects and attributes) and behavior (operations and state machines).

5 Related Work

Specifying behavior in OCL. OCL not only allows for specifying structural model features but also constraints on the behavior of objects by means of pre- and postconditions. In order that pre- and postconditions can be interpreted unambiguously, a detailed semantics of operation specifications is needed. The approach in [14] addresses this. However, according to [16], pre- and postconditions describe static aspects of the system, as they compare states of a system, which are static entities. Therefore in [16,17] the so-called action clause is introduced to the Object Constraint Language and is provided with a semantics.

Semantics of State Machines. In our approach we use UML protocol state machines to constrain the model behavior. The structure and the semantics of protocol state machines are discussed in [28]. The authors present an approach which applies protocol state machines to produce class contracts. The semantics of behavioral state machines is discussed in [20]. The authors apply the semantics for validity proofs of refinement transformations on behavior state machines. A formal semantics for the integration of UML statecharts into OCL, which makes it possible to formulate expressions over states in UML statecharts is presented in [5]. However the authors refer to an older UML version, whereby postconditions of protocol state machine transitions are not handled. The dynamic semantics of state machines is discussed in [2].

Usage of State Machines. Different approaches for the usage of state machines in the software testing context exist. Model-based testing (MBT) tools often use UML state machines as a basis for automatic test case generation. The approach in [38] makes it possible to automatically generate state machine diagrams from use cases. This approach is also implemented in a tool and evaluated in different case studies. The approach in [31] applies behavioral state machines for modeling reactive systems and automatic generation of test cases. Based on this, the input-output conformance of the systems is tested. The presented test approach is implemented by the so-called TEAGER tool suite. In [37], the authors report on an industrial cooperation for model-based testing applying UML state machines with a German rail engineering company. Based on a given UML state machine this approach makes it possible to automatically generate unit tests. The use of UML state machines for requirements validation is described in [25]. The authors apply Formal Concept Analysis (FCA) in analyzing the association between a set of test scenarios with a set of transitions specified in a UML state machine model. The authors of [35] use protocol state machines in the field of network security. They introduce Veritas, a tool which uses applications network traces to automatically generate protocol state machines. The generated state machines are able to represent incomplete knowledge about a protocol and are labeled as probabilistic protocol state machines (P-PSM). K-statecharts are an extension of UML statecharts which allow the use of knowledge-logic formulae in the statechart transition guard and are used for runtime verification of system behavior [4].

Tools. In [32] a tool set which supports static and dynamic validation of UML models is presented. The tool mOdCL is based on Maude, an executable formal specification language and is able to validate invariants and pre- and postconditions during the execution of a system [29]. In contrast to our approach and like the tool set presented in [32], mOdCL leaves out handling and runtime validation of protocol state machines. In [29], the authors report on the experiences with the development of a tool for dynamic enforcement of OCL constraints. Applying aspect-oriented programming (AOP), ocl2j automatically instruments OCL constraints in Java programs. In [24] a prototype of a tool being able to check the conformance of components within the UML extension for real-time (UML-RT) to the respective protocol state machines, which specify the legal communication between components, is described. Rhapsody is a verification environment for UML models. The tool implements an own semantics of statecharts, as discussed in [13]. The tool TABU allows for verification of reactive systems behavior [9]. For this purpose the behavior is modeled by state machines and automatically transformed into the used formal specification SMV (Symbolic Model Verifier). Additionally a number of CASE tools suchs as [6] allow for modeling statecharts, but are not able to validate state machines at runtime. In contrast to our approach, [1] and [22] don't provide full OCL support. Epsilon [18] is a platform which allows for model validation. However handling for state machines is not integrated.

Our contribution profits from these related works. It is however the only one which combines state machine validation with full OCL support for structural modeling and validation.

6 Conclusion

We have made a proposal for integrated structure and behavior modeling and validation. Full OCL support for (class and state) invariants and (operation and transition) pre- and postconditions guarantees that the underlying graphical models become precise. We combine descriptive requirements with an OCL-like imperative language. The models are validated and verified by test scenarios.

We plan to extend the supported UML state machine features, in particular, we will care for structuring mechanism like nested states. A number of improvements on the user interface can be realized, for example, an optional indication of protocol state machine states on object lifelines in sequence diagrams. Features of the behavior models like state reachability and other dynamic properties like liveness could be supported in a (semi-)automatic way. Consistency, redundancy and other relationships between the structural and behavioral model features should be investigated. Methodological questions about the usage of (class and state) invariants, and (operation and transition) pre- and postconditions must be discussed. Last but not least, larger case studies must give further feedback about the applicability and efficiency of the approach.

References

1. Abstract Solutions Ltd: Executable UML (xUML). Internet (2012), http://www.kc.com/XUML/
2. Börger, E., Cavarra, A., Riccobene, E.: Modeling the Dynamics of UML State Machines. In: Gurevich, Y., Kutter, P.W., Odersky, M., Thiele, L. (eds.) ASM 2000. LNCS, vol. 1912, pp. 223–241. Springer, Heidelberg (2000)
3. Büttner, F., Gogolla, M.: Modular Embedding of the Object Constraint Language into a Programming Language. In: Simao, A., Morgan, C. (eds.) SBMF 2011. LNCS, vol. 7021, pp. 124–139. Springer, Heidelberg (2011)
4. Drusinsky, D.: tak Shing, M.: Using UML Statecharts with Knowledge Logic Guards. In: Schürr and Selic [30], pp. 586–590 (2009)
5. Flake, S., Müller, W.: Formal semantics of static and temporal state-oriented OCL constraints. Software and System Modeling 2(3), 164–186 (2003)
6. Geiger, L., Zündorf, A.: Statechart Modeling with Fujaba. Electr. Notes Theor. Comput. Sci. 127(1), 37–49 (2005)
7. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL Models in USE by Automatic Snapshot Generation. Journal on Software and System Modeling 4(4), 386–398 (2005)
8. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-Based Specification Environment for Validating UML and OCL. Science of Computer Programming 69, 27–34 (2007)
9. Gutiérrez, M.E.B., Barrio-Solórzano, M., Quintero, C.E.C., de la Fuente, P.: UML Automatic Verification Tool with Formal Methods. Electr. Notes Theor. Comput. Sci. 127(4), 3–16 (2005)
10. Hamann, L., Gogolla, M., Kuhlmann, M.: OCL-Based Runtime Monitoring of JVM Hosted Applications. In: Cabot, J., Clariso, R., Gogolla, M., Wolff, B. (eds.) Proc. Workshop OCL and Textual Modelling (OCL 2011). ECEASST, Electronic Communications (2011), journal.ub.tu-berlin.de/eceasst/issue/view/56
11. Hamann, L., Hofrichter, O., Gogolla, M.: OCL-Based Runtime Monitoring of Applications with Protocol State Machines. In: Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., Kolovos, D. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 384–399. Springer, Heidelberg (2012)
12. Hamann, L., Vidács, L., Gogolla, M., Kuhlmann, M.: Abstract Runtime Monitoring with USE. In: Ferenc, R., Mens, T., Cleve, A. (eds.) Proc. CSMR 2012 (2012)
13. Harel, D., Kugler, H.: The Rhapsody Semantics of Statecharts (or, On the Executable Core of the UML) - Preliminary Version. In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) INT 2004. LNCS, vol. 3147, pp. 325–354. Springer, Heidelberg (2004)
14. Hennicker, R., Knapp, A., Baumeister, H.: Semantics of OCL Operation Specifications. Electr. Notes Theor. Comput. Sci. 102, 111–132 (2004)
15. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006)
16. Kleppe, A., Warmer, J.: Extending OCL to Include Actions. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 440–450. Springer, Heidelberg (2000)
17. Kleppe, A., Warmer, J.: The Semantics of the OCL Action Clause. In: Clark, T., Warmer, J. (eds.) Object Modeling with the OCL. LNCS, vol. 2263, pp. 213–227. Springer, Heidelberg (2002)

18. Kolovos, D., Rose, L., Paige, R.: The Epsilon Book. Internet (2012), http://www.eclipse.org/epsilon/doc/book
19. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive Validation of OCL Models by Integrating SAT Solving into USE. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 290–306. Springer, Heidelberg (2011)
20. Lano, K., Clark, D.: Semantics and Refinement of Behavior State Machines. In: Cordeiro, J., Filipe, J. (eds.) ICEIS, vol. (3-1), pp. 42–49 (2008)
21. Lano, K., Clark, D.: Axiomatic Semantics of State Machines, pp. 179–203. John Wiley & Sons, Inc. (2009)
22. Lano, K., Kolahdouz-Rahimi, S.: UML RSDS Model Transformation and Model-Driven Development Tools. Internet (2012), http://www.dcs.kcl.ac.uk/staff/kcl/uml2web
23. Mellor, S.J., Balcer, M.: Executable UML: A Foundation for Model-Driven Architectures. Addison-Wesley (2002)
24. Moffett, Y., Beaulieu, A., Dingel, J.: Verifying UML-RT Protocol Conformance Using Model Checking. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 410–424. Springer, Heidelberg (2011)
25. Ng, P.: A Concept Lattice Approach for Requirements Validation with UML State Machine Model. In: SERA, pp. 393–400. IEEE Computer Society (2007)
26. OMG (ed.): UML Superstructure 2.4.1. Object Management Group (OMG) (August 2011), http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF
27. OMG (ed.): Object Constraint Language 2.3.1. Object Management Group (OMG) (January 2012), http://www.omg.org/spec/OCL/2.3.1/
28. Porres, I., Rauf, I.: From Nondeterministic UML Protocol Statemachines to Class Contracts. In: ICST, pp. 107–116. IEEE Computer Society (2010)
29. Roldán, M., Durán, F.: Dynamic Validation of OCL Constraints with mOdCL. ECEASST 44 (2011)
30. Schürr, A., Selic, B. (eds.): MODELS 2009. LNCS, vol. 5795. Springer, Heidelberg (2009)
31. Seifert, D.: Conformance Testing Based on UML State Machines. In: Liu, S., Maibaum, T.S.E., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 45–65. Springer, Heidelberg (2008)
32. Shen, W., Compton, K.J., Huggins, J.: A UML Validation Toolset Based on Abstract State Machines. In: ASE, pp. 315–318. IEEE Computer Society (2001)
33. Shlaer, S., Mellor, S.J.: Object Lifecycles: Modeling the World in States. Yourdon Press, EngleWood Cliffs (1992)
34. Shlaer, S., Mellor, S.J.: Object-Oriented Systems Analysis: Modelling the World in Data. Yourdon Press, EngleWood Cliffs (1992)
35. Wang, Y., Zhang, Z., Yao, D.D., Qu, B., Guo, L.: Inferring Protocol State Machine from Network Traces: A Probabilistic Approach. In: Lopez, J., Tsudik, G. (eds.) ACNS 2011. LNCS, vol. 6715, pp. 1–18. Springer, Heidelberg (2011)
36. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Object Technology Series. Addison-Wesley, Reading (2003)
37. Weißleder, S.: Influencing Factors in Model-Based Testing with UML State Machines: Report on an Industrial Cooperation. In: Schürr and Selic [30], pp. 211–225 (2009)
38. Yue, T., Ali, S., Briand, L.C.: Automated Transition from Use Cases to UML State Machines to Support State-Based Testing. In: France, R.B., Küster, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 115–131. Springer, Heidelberg (2011)

Multi-perspectives on Feature Models

Julia Schroeter[1], Malte Lochau[2], and Tim Winkelmann[2]

[1] TU Dresden
Institute for Software- and Multimedia-Technology
julia.schroeter@tu-dresden.de
[2] TU Braunschweig
Institute for Programming and Reactive Systems
{m.lochau,t.winkelmann}@tu-bs.de

Abstract. Domain feature models concisely express commonality and variability among variants of a software product line. For supporting separation of concerns, e.g., due to legal restrictions, technical considerations and business requirements, multi-view approaches restrict the configuration choices on feature models for different stakeholders. However, recent approaches lack a formalization for precise, yet flexible specifications of views that ensure every derivable configuration perspective to obey feature model semantics. Here, we introduce a novel approach for preconfiguring feature models to create multi-perspectives. Such customized perspectives result from composition of various concern-relevant views. A structured view model is used to organize features in view groups, wherein a feature may be contained in multiple views. We provide formalizations for view composition and guaranteed consistency of perspectives w.r.t. feature model semantics. Thereupon, an efficient algorithm to verify consistency for entire multi-perspectives is provided. We present an implementation and evaluate our concepts by means of various experiments.

Keywords: Software Product Lines, Feature Models, Preconfiguration, Customization, Automated View Composition.

1 Introduction

In software product line (SPL) engineering, the variability and commonality among product variants of the same domain are expressed in a domain feature model [15,28,36]. It organizes features in a hierarchical structure as well as dependencies and constraints between them. In general, the entire domain feature model is used to derive product variants. However, there are various case scenarios which require the variant space defined by the domain feature model to be further restricted. Reasons for those preconfigurations are driven by business or legal concerns, e.g., to enable a variable pricing strategy for offering features as packages to various stakeholders [25]. Other concerns may be of technical nature, e.g., to restrict the overall variant space to a representative subset for efficiently testing complete SPLs.

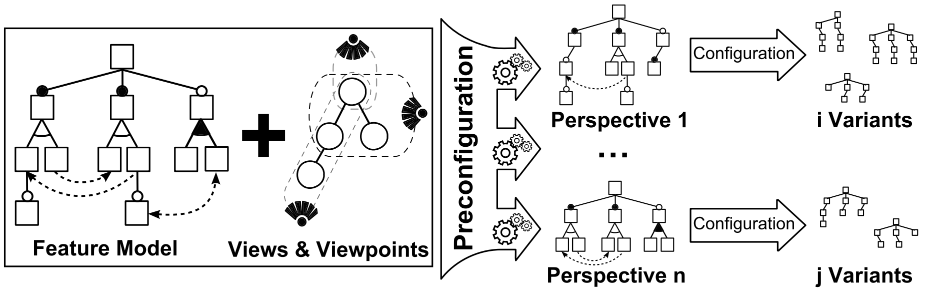


Fig. 1. Perspectives are created by joining multiple views on a domain feature model

It seems promising to express these concerns by grouping features in a separate model orthogonally to the domain feature model. Prior the derivation of a product by a specific stakeholder, concern-related groups are selected and the domain feature model is filtered accordingly to ensure that restricted features are not available for selection. Those groups are perceived as hierarchically organized views crosscutting the feature model structure. According to the ISO/IEC/IEEE 42010:2011, *Systems and software engineering* standard[1], "a view is a representation of the whole system from the perspective of a related set of concerns". In other words, a view shows only features that belong to concerns of a stakeholder. Multiple approaches to create views on feature models exist [3,20,13,27]. Though, these approaches focus on the multi-dimensional separation of concerns (MDSoC) and a particular view is not intended to derive a complete product variant, but rather to allow for specific configuration decisions only. But, to the best of our knowledge, there are no approaches that aggregate *and* integrate views to tailor and customize the variant space and result in a semantical refinement of the feature model.

To tackle these challenges, we propose multi-perspectives on feature models. Therefore, in a perspective we aggregate multiple views to refine the variant space of the original domain feature model, as shown in Fig. 1. We define further requirements to guarantee consistency of any potential perspective and introduce viewpoints to explicitly define allowed view combinations. Every viewpoint requires to incorporate a feature model perspective that states a specialization, i.e., a refinement of the original feature model semantics. A domain feature model and a view model are unified in a multi-perspective model, which imposes a conservative extension to the domain feature model. Ensuring multi-perspective model consistency is, in general, hard to maintain due to its crosscutting nature w.r.t. the feature model and the potential overlappings of view groups in a view model. Therefore, besides a comprehensive brute force approach, we also provide an incremental heuristic for efficiently verifying the consistency of multi-perspective models. Furthermore, our concepts support customization on feature model level in the way that stakeholder-specific features added to the domain feature model are restricted to that particular stakeholder's perspective.

[1] http://www.iso-architecture.org/ieee-1471/

The structure of this paper is as follows. We review feature models with group cardinalities in Sect. 2. In Sect. 3, we describe requirements for modeling pre-configurations and customizations of feature models by means of an illustrative example and describe how they are addressed in our multi-perspective approach. In Sect. 4, we formalize our concepts of multi-perspectives on feature models, we outline consistency requirements and provide an efficient algorithm for their verification. In Sect. 5, we present an implementation of the concepts, and provide an evaluation concerning efficient consistency verification. Finally, we present related work in Sect. 6 and conclude in Sect. 7 [2].

2 Feature Models with Group Cardinalities

We first review some basic notions concerning syntax and semantics of *feature models* with group cardinalities. A multitude of feature modeling variants exists in the literature [10]. Here, we refer to the approach introduced in the feature oriented domain analysis (FODA) study [22]. Feature models organize features in a tree structure. Different kinds of edges represent different hierarchical decomposition relations between a parent feature and groups of child features. These concepts are graphically represented in feature diagrams as shown for instance in Fig. 2, where we use the notation according to Czarnecki and Eisenecker [15]. For a feature diagram language to be conceptional complete, i.e., *fully expressive*, further *constraints* are to be provided, i.e., at least *requires* and *exclude* edges [32]. Those constraints are often represented as additional propositional formulas over features, arbitrarily crosscutting the feature tree [11]. Further constructs for enhancing feature models are mentioned in the literature, e.g., abstract features, feature cardinalities, feature attributes and feature references which are out of scope of this paper. We consider the two notions *feature model* and *feature diagram* as synonyms in the following and introduce an abstract syntax for feature models with group cardinalities.

Definition 1 (Feature Model). *A feature model is a 4-tuple* $(F, \prec, \lambda, \Phi)$, *where* F *is a finite set of feature nodes,* $\prec \subseteq F \times F$ *is a decomposition relation on* F, $\lambda : \mathcal{P}(F) \rightharpoonup \mathbb{N}_0 \times \mathbb{N}_0$ *is a partial cardinality function assigning intervals to feature groups, and* Φ *is a set of propositional formulas over* F.

A *well-formed* feature model *FM* must further satisfy the following rules (cf. [33]): (1) relation $\prec$ forms a *rooted tree* on F, (2) in feature groups $F' \in dom(\lambda)$, except the singleton group F_r solely containing the root feature, all features have the same parent node under $\prec$, (3) domain $dom(\lambda)$ of partial function λ fully partitions F, feature groups are non-empty, i.e, $\emptyset \notin dom(\lambda)$, and (4) cardinalities $\lambda(F') = (k, l)$ of feature groups $F' \subseteq F$ define reasonable intervals for child features, i.e, $k \leq l$ and $l \leq |F'|$ holds.

Cardinalities define the four common decomposition types for feature groups $F' \in dom(\lambda)$, where $n = |F'|$, as follows: $\lambda(F') = (n, n)$ for *mandatory* groups,

[2] A detailed version of our concepts and proofs can be found in [33].

$\lambda(F') = (0, n)$ for *optional* groups, $\lambda(F') = (1, 1)$ for *alternative* groups and $\lambda(F') = (1, n)$ for *or* groups (cf. Fig. 2). For the group F_r of the root feature, we assume $\lambda(F_r) = (1, 1)$. Propositional formulas $\phi \in \Phi$ are boolean formulas $\phi \in \mathbb{B}(F)$ over feature names in F expressing cross-tree constraints. In particular, according to Heymans et al. [19], we can restrict Φ to solely contain binary *require* constraints leading from feature f to feature f' to be implications $\phi_{rq} = f \to f'$ and binary *exclude* constraints, i.e., implications $\phi_{ex} = f \to \overline{f'}$. The set Φ is interpreted as the conjunction $\bigwedge_{\phi \in \Phi} \phi$ of all constraints. By $\mathcal{FM}(F)$ we refer to the set of all well-formed feature models over features F.

The semantics of a feature model *FM* defines the *variant space*, i.e., the set of valid product configurations. A product configuration is a subset $F_{pc} \subseteq F$ of features selected for a concrete product variant. Hence, the semantical evaluation function

$$[\![\cdot]\!] : \mathcal{FM}(F) \to \mathcal{P}(\mathcal{P}(F))$$

maps feature models $FM \in \mathcal{FM}(F)$ over features F into the domain of sets of *valid* product configurations obeying the decomposition types and constraints, i.e., $[\![FM]\!] \in \mathcal{P}(\mathcal{P}(F))$. The semantical evaluation function defines the *maximum set* of valid product configurations such that

$$\begin{aligned}[\![FM]\!] = \{F_{pc} \in \mathcal{P}(F) \mid\ & f_r \in F_{pc} \wedge \\ & (f \in F_{pc} \wedge f \prec F' \wedge \lambda(F') = (k, l) \Rightarrow k \leq |\{f' \in F' \cap F_{pc}\}| \leq l) \wedge \\ & (f'' \in F_{pc} \wedge f''' \prec f'' \Rightarrow f''' \in F_{pc}) \wedge F_{pc} \models \bigwedge_{\phi \in \Phi} \phi\}\end{aligned}$$

where $f \prec F' :\Leftrightarrow \forall f' \in F' : f \prec f'$. Thus, validity of configurations $F_{pc} \in \mathcal{P}(F)$ requires (1) that the root node f_r is selected, (2) satisfaction of group cardinalities concerning features f' in groups F' decomposing selected nodes f, (3) justification of a selected feature f'' by means of the presences of its parent feature f''', and (4) satisfaction of all global constraints in Φ on F_{pc}. A feature model *FM* is *satisfiable*, if $[\![FM]\!] \neq \emptyset$ (cf. [19]). We assume any given feature model under consideration to be satisfiable in the following.

3 Model-Based Multi-perspectives on Feature Models

In this section, we introduce our modeling concepts and practices for multi-perspective SPL engineering. We use a case study to derive requirements for modeling tailorings and customizations as conservative extensions of feature models by means of implicit preconfigurations. Therefore, *viewpoints* are introduced to integrate sets of views, each restricting a feature model to subsets of features, e.g., dedicated to particular business concerns relevant for a group of stakeholders. Feature model preconfigurations, i.e., *perspectives*, then result from the aggregation of those views related to a viewpoint under consideration.

3.1 Illustrative Example: Document Management System

Our case study describes an SPL for a server-side document management system (DMS) that provides methods to store, search and retrieve documents. The DMS domain feature model consisting of 23 features is shown on the left of Fig. 2. The root feature `DocumentManagementSystem` represents the document management application. The DMS supports four `DocumentTypes` organized in an *or* group. The optional `OCR` feature is specialized to `PDFOCR` and/or `ImageOCR`, each requiring the according document type. The `Indexing` feature has `FileNameIndex` as a mandatory feature, whereas `GeneralIndex` and `MetaDataIndex` (with three further specializations) are alternative features. The `Search` feature provides a `FileNameSearch`, `GeneralSearch` and `MetaDataSearch` and further `TitleSearch`, `ContentSearch` and `AuthorSearch`.

Due to business concerns, a feature model is to be tailored to a restricted view on the complete configuration space prior deriving customized DMS products. A view model is used to associate features and business concerns. As indicated on the right of Fig. 2, different stakeholder concerns are separated into view groups denoted as circles, e.g., `Basic` vs. `Premium` members. To each view group, a particular set of relevant features is assigned. This is highlighted by equal hatchings in Fig. 2, e.g., feature `TextType` is selectable for any stakeholder of the `Core` group, whereas the `OCR` features are dedicated to `Gold` members. The hierarchy among groups reflects specializations of concerns from parent groups to child groups. A particular viewpoint is then chosen by a stakeholder (e.g., `Viewpoint SilverUser`) to aggregate the corresponding view groups (e.g., `Silver`, `Premium` and `Core`) and create a perspective on the feature model, which contains only features assigned to the implicitly selected view groups.

In the following, we collect the requirements for preconfigurations on feature models and describe how we address them in our modeling framework for multi-perspective SPL engineering. We give a concise formalization of the framework in Sect. 3.

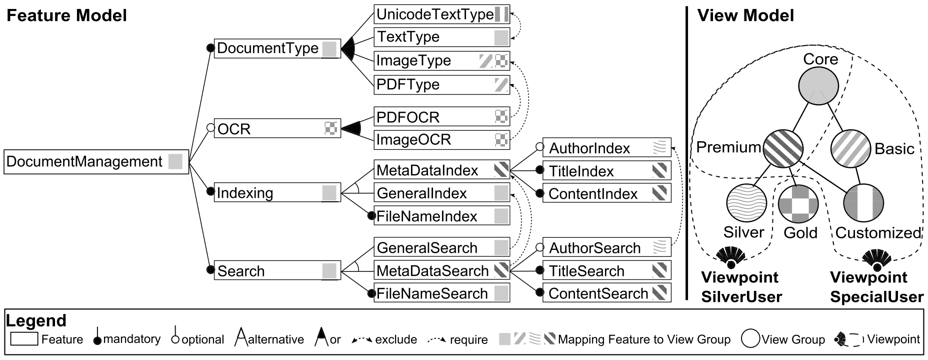


Fig. 2. Feature model and view model of the document management system example

3.2 Multi-perspective SPL Engineering

In SPL engineering, we distinguish the processes of *domain engineering* and *application engineering* [15,28,36]. We describe, how to extend these processes to support a model-based approach for preconfigurations and customizations of domain feature models. In addition to modeling commonality and variability among products of an SPL in a domain feature model, we propose to specify multiple perspectives for integrating different stakeholder concerns already in the domain engineering process.

From the case scenarios described above, we already obtained some requirements, a multi-perspective approach has to satisfy. We give a detailed overview on these requirements and explain concretely how they are addressed in our approach. First of all, we propose the notion of perspectives.

Requirement 1. *A perspective on a domain feature model is a virtual view resulting from the aggregation of multiple views, where each view is dedicated to a stakeholder's concern.*

Perspectives are specializations of feature models [16]. The intuition is that a perspective allows to derive a set of products being a valid subset of the original domain feature model. In particular, perspectives assemble multiple concerns, i.e., subsets of domain features relevant to particular stakeholders into preconfigured feature models as shown in Fig. 1. For modeling complex relationships between concerns, feature model views are to be organized in groups in the view model. For instance, the `Premium` group encapsulates those concerns of the DMS dedicated to `Premium` customizers. A hierarchy on the set of groups in the view model allows for step-wise refinement of perspectives by adopting the principles of inheritance. In the DMS, the `Core` group has the `Premium` and `Basic` group as direct subgroups, thus both inherit the concerns of the `Core` group. The `Premium` group is further refined by the `Silver` and `Gold` group, whereas `Customized` refines the `Basic` group. In addition, crosscutting concerns are expressed in terms of multiple inheritance: the `customized` group may inherit concerns from `Premium` groups, as well as from `Basic` as both are direct predecessors within the group hierarchy. Hence, we propose the introduction of a view model that specifies those various kinds of relationships between stakeholder's concerns.

Requirement 2. *A view model specifies relationships among stakeholder's concerns by hierarchical view refinements and multiple inheritance.*

However, some groups in a view model may not be allowed to aggregate perspectives from its assembled views. Hence, a distinction between *concrete* and *partial* group views is required. Therefore, we propose a *viewpoint* concept. Nuseibeh et al. use this concept to describe a concrete perspective on a system [17,26]. In our approach we use viewpoints for explicitly denoting collections of related group views being permitted to form a *valid* perspective accessible to stakeholders. The view model structure implies that a group referenced by a viewpoint and all its predecessor groups are aggregated in the perspective.

Requirement 3. *A viewpoint specifies sets of views to build a valid perspective.*

After a view model has been created and features are assigned to groups of the view model, viewpoints are identified to create perspectives in the application engineering process. In the view model of the DMS example, two viewpoints are defined, each denoted by an eye-like symbol and a dashed line that encloses the set of included view groups, e.g., viewpoint `SpecialUser` includes groups `Customized`, `Basic`, `Premium`, and `Core`. As a consequence, a perspective solely aggregated for views of the groups `Basic` and `Premium` is invalid in that view model. For concise modeling of multiple perspectives on feature models, two special constructs in the view model are valuable: a core group and singleton groups. The core group builds the unique top element of the view group hierarchy thus collecting all concerns common to each stakeholder by subsuming every viewpoint. In the DMS example, the `Core` group refers to concerns common to every stakeholder, i.e., core features that are mandatory to every product configuration, e.g., `DocumentType`. Singleton groups are dedicated to one particular viewpoint, therefore denoting customizations for particular stakeholders. For instance, the `UnicodeTextType` feature is exclusively dedicated to stakeholders in the `Customized` group. Hence, we constitute the following requirement.

Requirement 4. *A unique core group collects those concerns common to all stakeholders and singleton groups define customizations on feature model level.*

For a seamless integration of the multi-perspective approach into existing SPL workflows, we propose the modeling and derivation of perspectives as a conservative extension to feature models. Therefore, we separate the view model from the original domain feature model and use an additional multi-perspective model to integrate both. Such a model combines the feature model with a view model as shown in Fig. 2 so that both can be created independently. Hence, multi-perspective models define a flexible $n : m$ mapping between (stakeholder) view groups and feature model views. For instance, the DMS feature `ImageType` is assigned to the view groups `Basic` and `Gold` and is therefore included in both corresponding views.

However, the presence of such complex mappings complicates the verification whether all potential perspectives on a feature model, that are induced by design decisions during domain engineering, constitute meaningful, i.e., feature model semantics preserving preconfigurations.

Requirement 5. *All perspectives derivable from a multi-perspective model preserve feature model semantics by imposing refinements of the configuration space.*

In Sect. 4, we provide a formalization of the multi-perspective model that includes all requirements outlined above. Thereupon, constructive criteria for multi-perspective model consistency are developed, and an efficient heuristic algorithm for their verification is provided. We conclude this section by discussing some engineering practices and modeling patterns for creating meaningful preconfigurations:

– *Core Features* that are mandatory for all product variants [10], e.g., `DocumentType`, are to be included in the core group together with all features that

are not explicitly assigned to other view model groups. Those core features are available per default in every perspective.
- *Optional, Replaced and Excluded Features.* Features that should not be contained in all perspectives, features that exclude each other in all perspectives, and even features to be excluded from any perspective can be specified by declaring them optional features in the feature model. Similar to *abstract features* [16], this can be also used to hide entire subtrees such as `OCR`.
- *Hierarchical Restrictions.* Even though feature models and view models are orthogonal, we propose hierarchies on group views to some extent correspond to feature tree hierarchies to keep the overall consistency graspable.
- *Customizations.* Stakeholders often request customized and not yet available features. Those features will be exclusively accessible to the particular stakeholder. Our multi-perspective approach supports such SPL customizations on feature model level that avoids feature model pollutions. Therefore, singleton groups provide stakeholder-specific viewpoints for creating customized perspectives, where even replacements of features by customized ones can be modeled.

In application engineering, valid perspectives create preconfigurations on the domain feature model before product variants are derived. Therefore, the stakeholder chooses a viewpoint by selecting groups from the view model reflecting his concerns. As the view model is hierarchically structured, all ancestor groups of the stakeholder-specific group including the core group are contained in the viewpoint, thus deriving valid perspectives from a viewpoint constitutes an automated task.

4 Formalization of Multi-perspectives on Feature Models

Views select subsets of configuration parameters to restrict the access to a feature model. Formally, a view projects from a feature model $FM \in \mathcal{FM}(F)$ to a subset of features $F' \subseteq F$ and related constraints.

Definition 2 (Feature Model View). *A view of feature model $FM \in \mathcal{FM}(F)$ is a pair (F_V, Φ_V) consisting of a subset $F_V \subseteq F$ of selectable features, and a subset $\Phi_V \subseteq \Phi$ of constraints such that $\phi_V \in \mathbb{B}(F_V)$ for each $\phi_V \in \Phi_V$.*

By $\mathcal{V}_{FM}$, we refer to the set of all views of a feature model FM. In general, a view $V_{FM} \in \mathcal{V}_{FM}$ contains an arbitrary selection of features $F_V \subseteq F$ and corresponding constraints $\phi_V \in \Phi_V$. In Fig. 2, six views are highlighted via different hatchings marking features selected into the same view. Note that feature `ImageType` is projected into two views as it is assigned to two view groups in the view model.

Views are associated with *perspectives* by interpreting them as variability-reduced feature models, i.e., a partial tree of the original feature tree. For the sake of simplicity, we assume $\mathcal{FM}(F_V) \subseteq \mathcal{FM}(F)$, where $F_V \subseteq F$, i.e., $\mathcal{FM}(F)$ also contains feature models on subsets F_V of F. Formally, a perspective $FM_V \in \mathcal{FM}(F)$ for a view $V_{FM} \in \mathcal{V}_{FM}$ defines a projection function

$$p_{FM} : \mathcal{V}_{FM} \rightharpoonup \mathcal{FM}(F), \text{ where } p_{FM}(V_{FM}) = (F_V, \prec_V, \lambda_V, \Phi_V)$$

such that (1) $\prec_V \subseteq F_V \times F_V \subseteq \prec$ denotes the *restriction* of $\prec$ onto F_V, and (2) λ_V is reduced to F_V as follows

– if $F' \in dom(\lambda)$ and $F' \cap F_V \neq \emptyset$, then $F' \cap F_V \in dom(\lambda_V)$ and
– if $\lambda(F') = (k, l)$, then $\lambda_V(F' \cap F_V) = (k, l - \mid F' \setminus \{F_V \cap F'\} \mid)$.

Function p_{FM} is partial as views $V_{FM} \in \mathcal{V}_{FM}$ exist, whose projection applications yield a syntactically *ill-formed* feature model $p_{FM}(V_{FM}) \notin \mathcal{FM}(F)$. Furthermore, even if $p_{FM}(V_{FM}) \in \mathcal{FM}(F)$ holds, the perspective $p_{FM}(V_{FM})$ is not necessarily *semantically* refining *FM*, i.e., $[\![p_{FM}(V_{FM})]\!] \not\subseteq [\![FM]\!]$. Therefore, we call a view $V_{FM} \in \mathcal{FM}(F)$ if *FM-consistent*

1. $p_{FM}(V_{FM}) \in \mathcal{FM}(F)$,
2. $[\![p_{FM}(V_{FM})]\!] \subseteq [\![FM]\!]$ and
3. $p_{FM}(V_{FM})$ is *satisfiable*.

Property 1. holds, if the feature selection preserves the tree structure of *FM* and obeys feature group constraints. For property 2., constraints are to be considered. For each constraint $\phi \in \Phi \setminus \Phi_v$ and $F' \subseteq F$ to be the subset of features appearing in ϕ, we require $F' \cap F_v = \emptyset$. We weaken this property as we focus on feature models with binary constraints. Thus, feature selections must solely support feature implications to be satisfiable, as exclude constraints are either fully supported, or they cannot be invalidated in a view, because one of the features is not present.

Lemma 1. *A view $V_{FM} \in \mathcal{V}_{FM}$ on a satisfiable FM is FM-consistent if*

– *$f_r \in F_V$,*
– *$f \in F_V$ and $f' \prec f$, then $f' \in F_V$,*
– *$f \in F_V$ and $f \prec F'$ with $\lambda(F') = (k, l)$, then $|F' \cap F_V| \geq k$ and*
– *$f \in F_V$ and $f \rightarrow f' \in \Phi$, then $f' \in F_V$, thus $f \rightarrow f' \in \Phi_V$.*

By $\mathcal{V}^c_{FM} \subseteq \mathcal{V}_{FM}$ we refer to the subset of *FM-consistent* views on feature model *FM*. Views $V_{FM} \notin \mathcal{V}^c_{FM}$ are *partial* views. In Fig. 2, the core view marked with solid grey hatchings is *FM-consistent* as it satisfies all conditions of Lemma 1, whereas the remaining views are not, but rather refine the core view by individual features for particular concerns. As stated in Req. 1, we consider perspectives to result from integrating multiple, potentially inconsistent views into an aggregated *FM*-consistent view (Req. 5). For aggregating multiple views, we introduce a *view composition operator*

$$\oplus : \mathcal{V}_{FM} \times \mathcal{V}_{FM} \rightarrow \mathcal{V}_{FM}, \text{ where } V_{FM} \oplus V'_{FM} = V''_{FM} = (F_V \cup F_{V'}, \Phi_{V''})$$

such that $\Phi_{V''} \subseteq \Phi$ and $\phi \in \Phi_{V''} :\Leftrightarrow \phi \in \mathbb{B}(F_{V''})$.

Lemma 2. *The view composition operator is commutative and associative.*

Due to the crosscutting constraints in Φ, feature model semantics is not compositional [4]. Hence, view composition does not commute with *FM* semantics

$$[\![p_{FM}(V_{FM} \oplus V_{FM'})]\!] \neq [\![p_{FM}(V_{FM})]\!] \cup [\![p_{FM}(V'_{FM})]\!].$$

In particular, from $\Phi_{V''} \neq \Phi_V \wedge \Phi_{V'}$, it follows that constraints $\phi \in \Phi_{F_{V''}} \cap \Phi_F$ with $\phi \notin \Phi_{F_V} \cup \Phi_{F_{V'}}$ may exist. Fortunately, view composition is closed under *FM-consistency*.

Proposition 1 (Closedness of FM-consistent View Composition). *For FM-consistent views* $V_{FM}, V'_{FM} \in \mathcal{V}^c_{FM}$*, it holds that* $V_{FM} \oplus V'_{FM} \in \mathcal{V}^c_{FM}$*.*

Proof. Well-formed tree structures and group constraints are preserved as both views are *FM-consistent* for feature model *FM* and composition is monotone on F. For constraints $\phi = f \rightarrow f'$ with $f, f' \in F_{V''}$, $\phi \in \Phi_{V''}$ is guaranteed as ϕ is either contained in V and/or V'. Otherwise, one view must have contained f without f' which contradicts the *FM-consistency* assumption.

The opposite direction of Prop. 1 does not hold. For instance, when aggregating all views in Fig 2, the result is indeed *FM-consistent*. We use view composition to derive stakeholder-specific feature model perspectives. The aggregation of those perspectives depends on the organization of their *viewpoints* in a *view model*.

4.1 View Models

Multi-view approaches partition feature models for separation of concerns relevant to different stakeholders. But, as concerns are potentially interrelated, we suppose views to be hierarchically organized in groups. Various (potentially partial) views are aggregated into well-defined *viewpoints* to derive tailored feature model preconfigurations. For capturing the relationships between views and viewpoints, we introduce a *view model*.

Definition 3 (View Model). *A view model is a pair* (VP, G)*, where* $VP = \{vp_1, vp_2, \ldots, vp_m\}$ *is a finite set of viewpoints and* $G = \{g_1, g_2, \ldots, g_n\}$ *is a finite set of view model groups, i.e., a collection of predicates* $g_i \subseteq VP$ *over viewpoints.*

Predicates $g_i \in G$ indicate the corresponding subset $g_i \subseteq VP$ of viewpoints to be *members* of that group, thus sharing the concerns dedicated to that group. The subsets introduce an implicit *hierarchy relation* $<_G \subseteq G \times G$ on groups (cf. Req. 2), where $g <_G g' :\Leftrightarrow g \subset g'$, thus defining a *predecessor* relation via strict inclusion of viewpoint sets. Relation $<_G$ is a *strict* partial order as we allow groups with equal predicates $g_i = g_j$ to be distinguished in G by their indices i and j. Hence, two groups g_i and g_j are either related under $<_G$ or incomparable, i.e., either (1) disjoint or (2) overlapping (including set equality). We define an *overlapping group relation* $\sqcap_G \subseteq G \times G$

$$g_i \sqcap_G g_j :\Leftrightarrow i \neq j \wedge g_i \cap g_j \neq \emptyset \wedge g_i \not<_G g_j \wedge g_j \not<_G g_i$$

being irreflexive, symmetrical and non-transitive. For *well-formed* view models, we require $<_G$ to be upwards closed in G, i.e., there exists a unique *core group* $g_{core} \in G$ with $g_{core} = VP$, thus $g <_G g_{core}$ for each $g \in G$ (cf. Req. 4). We use the following notations

– a group $g' \in G$ is a *direct predecessor* of $g \in G$, if $g <_G g'$ and there is no $g'' \in G$ such that $g <_G g'' <_G g'$,
– a group $g \in G$ is *most specific* for a viewpoint $vp \in VP$, if $vp \in G$, and there is no $g' \in G$ with $vp \in G$ and $g' <_G g$.

The core group has no predecessors. Due to overlappings of groups, a view model *VM* allows any other group to have multiple direct predecessors and viewpoints to have multiple most specific groups (cf. Req. 2). We further distinguish *abstract* and *concrete* groups. Group $g \in G$ is concrete, if it is *most specific* to at least one viewpoint $vp \in VP$, otherwise it is *abstract* (cf. Req. 3). In the graphical representation of the view model in Fig 2, circles denote groups and a lattice structure is used to visualize the multiple inheritance hierarchy on groups. For instance, `Silver` $<_G$ `Premium` $<_G$ `Core` holds, whereas `Silver` and `Gold` are unrelated under $<_G$. The viewpoints `SilverUser` and `SpecialUser` are denoted by eye-like symbols and dashed lines mark the groups the viewpoint is part of. Therefore, `Premium` $\sqcap_G$ `Basic` holds, because `SpecialUser` $\in$ `Premium` $\cap$ `Basic`. For singleton groups $g \in G$, where $g = \{vp_i\}$, used to assign exclusive properties to viewpoints $vp \in VP$, we also require uniqueness in G. For instance, the singleton group `Customized` restricts the availability of the customization feature `UnicodeTextType`. Each viewpoint in a view model aggregates its assigned and inherited views to build a *perspective*. The set of valid perspectives on the feature model are specified in a *multi-perspective model*.

4.2 Multi-perspective Models

The integration of feature model views and view models in a *multi-perspective model* imposes multiple perspectives, one for each viewpoint.

Definition 4 (Multi-Perspective Model). *A multi-perspective model is a triple* (FM, VM, σ)*, where* $FM \in \mathcal{FM}(F)$ *is a feature model,* $VM = (VP, G)$ *is a view model, and* $\sigma : G \to \mathcal{V}_{FM}$ *is view mapping function.*

We require every feature of feature model *FM* to be mapped to at least one view, i.e., for each $f \in F$, there is some $g \in G$ with $\sigma(g) = (F_g, \Phi_g)$ such that $f \in F_G$. The mapping σ in Fig. 2 is denoted by similar hatchings of features and groups. Thus, the `Core` group maps to solid gray features and group `Customized` maps to the customization feature `UnicodeTextType`. For accessing a feature model perspective, a stakeholder chooses a viewpoint according to its concerns. Viewpoints $vp \in VP$ refer to aggregated views V_{vp} by joining all views mapped to groups of that viewpoint

$$V_{vp} = \sigma(g_{core}) \oplus \sigma(g_1) \oplus \sigma(g_2) \cdots \oplus \sigma(g_k), \text{ where } vp \in g_i, 1 \leq i \leq k$$

By $\mathcal{V}_{MP} \subseteq \mathcal{V}_{FM}$, we denote the set of all views of any viewpoint in a multi-perspective model *MP* on *FM*. The restrictions of feature models to views $\mathcal{V}_{MP}$ define multiple perspectives $FM_{vp} = p(V_{vp})$. All potential perspectives must preserve the original feature model semantics, i.e., being *FM-consistent* (cf. Req. 5).

4.3 Consistency of Multi-perspective Models

Despite multi-views, multi-perspectives on feature models do not require all views to obey consistency properties, but only those being non-partial, i.e., visible to a viewpoint (cf. Req. 3). A multi-perspective model is *consistent* if all derivable perspectives are projected from *FM-consistent* views.

Lemma 3. *The multi-perspective model $MP = (FM, VM, \sigma)$ is consistent, if $\mathcal{V}_{MP} \subseteq \mathcal{V}_{FM}^{C}$.*

The perspectives of both viewpoints in Fig. 2 are *consistent* as both are built from *FM-consistent* views. However, if, e.g., `AuthorIndex` is removed from group `Silver`, the perspective of viewpoint `SilverUser` becomes inconsistent as `AuthorSearch` requires `AuthorIndex`. Note that `OCR` and related sub features are excluded from any preconfigured feature model, as currently no viewpoint is defined that includes the corresponding `Gold` group.

A brute force algorithm for the verification of *MP* consistency works as follows (cf. [33]): (1) iterate over all viewpoints, (2) iterate over all groups of the viewpoint, (3) compose all views of the groups, (4) check satisfiability and *FM-consistency* of the composed perspective. Our experiments have shown that this algorithm works well for models of limited size w.r.t. the number of features and groups, but it does not scale for more complex models with numerous viewpoints [33]. In addition, *FM-consistency* includes to verify *satisfiability* which is presumably NP-complete [8]. Because of inclusions and overlappings in the group hierarchy, many redundant checks are performed due to commonality between viewpoints contained in non-disjoint groups. To check large-scale models, we propose a more conservative criterion imposing a sufficient, but not necessary requirement, that is verifiable in an efficient, incremental way by iterating over groups instead of viewpoints. We interpret view models $VM = (VP, G)$ as lattices $(G_c, \rightarrow)$ with *concrete* groups $G_c \subseteq G$ as nodes and edges $g \rightarrow g'$ leading from g to g' if $g' <_G^* g$ without *concrete* groups between g' and g. Starting from the core group, Algorithm 1 checks satisfiability only once for the core group, and then incrementally checks for every edge $g \rightarrow g'$ the preservation of the conditions of Lemma 1 by considering sets of features added via partial views of abstract groups passed from g to g' (denoted by $F_{g \rightarrow g'}$) and those added via g. Thus, a traversal with complexity equivalent to breath-first-search on $(G_c, \rightarrow)$ is performed, where each segment $g \rightarrow g'$ is checked based on previous steps.

Theorem 1 (Multi-Perspective Model Consistency). *If MP passes Algorithm 1 successfully, then MP is consistent.*

Proof (cf. [33]). First, the algorithm always terminates because $(G_c, \rightarrow)$ is a finite, directed acyclic graph. The incremental traversal ensures concrete views aggregated from views of groups via hierarchical inclusions to preserve *FM-consistency*, if all its predecessors under $<_G$ are *FM-consistent*. Prop. 1 ensures views arbitrarily composed for groups $g \sqcap_G g'$ to preserve *FM-consistency*, even though not explicitly checked.

Algorithm 1 Incremental Heuristic for Multi-Perspective Consistency Check

```
Input: $FM$, $(G_c, \rightarrow)$, $\sigma$
Require: $g_{core} \in G_c$
$\forall g \in G_c : g.F = \sigma(g)$ {feature sets mapped and aggregated to groups}
$\forall g \in G_c : g.cons =$ true {flag for group views FM-consistency}
$g_{core}.cons := check(g_{core}.F, FM)$ {consistency checks, cf. Lemma 1}
$\forall g \in G_c : g.done =$ false {predecessor nodes of node completely checked}
$g_{core}.done :=$ true
for all $g \in G_c$ where $g.done =$ true do
  for all $g' \in G_c$ where $g \rightarrow g'$ do
    $g'.F := g'.F \cup g.F \cup F_{g \rightarrow g'}$ {add features from predecessors between $g$ and $g'$}
    $g'.cons := check(g'.F, FM) \wedge g'.cons$ {check consistency preservation}
    if $\forall g'' \in G_c$ where $g'' \rightarrow g' : g''.done =$ true then
      $g' done :=$ true {all predecessors of $g'$ checked}
    end if
  end for
  $G_c := G_c \setminus g$ {check of $g$ done}
end for
return true if $\forall g \in G_c : g.cons =$ true
```

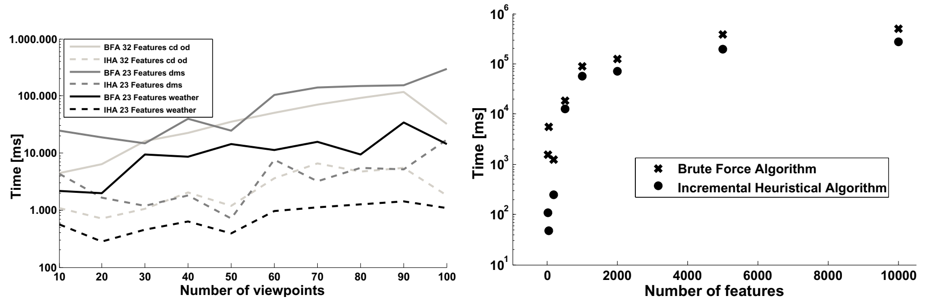
The opposite direction of Theorem 1 does not hold as the algorithm may produce false negatives, i.e., reporting models to be inconsistent, even though all potential viewpoints have consistent views after aggregation of all overlapping views. To avoid false negatives, we further combine the `check` procedure of the heuristic with an explicit call of the exhaustive verification of groups presumably being inconsistent. This way, the number of explicit satisfiability checks is drastically reduced compared to the brute force algorithm (cf. Sect. 5).

5 Implementation and Evaluation of Multi-perspectives

We implemented our multi-perspective approach in a tool called *Conper* [34] as an extension of the *FeatureMapper* environment [18]. Further information, the source code, examples, and screen casts on how to use the tool are provided online[3]. The implementation contains a multi-perspective editor to create view models and viewpoints together with multi-perspective mappings, and to perform consistency checks. The algorithms of Sect. 4.3 are integrated in the editor to verify multi-perspective model consistency. A perspective can be created on each consistent viewpoint, which is then used as preconfigured input for the variant editor of the *FeatureMapper*.

To evaluate the performance of both consistency check algorithms, we used 12 consistent feature models with varying numbers of features, cross-tree constraints (CTCs) and different cross-tree constraint ratio (CTCR). Two further feature models are created using the *FeatureMapper*. From the software product line online tools (SPLOT) [2] repository, we chose 5 manually created and 5 generated

[3] `https://github.com/multi-perspectives/cluster/`



(a) Varying the number of viewpoints. (b) Feature models with varying size.

Fig. 3. Comparing the consistency check algorithms

feature models with up to 10,000 features. The Choco constraint satisfaction problem (CSP) solver [1] is used to check the satisfiability of our feature models (cf. [9,24,23]). In addition, we use randomly generated multi-perspective models. Based on the 12 feature models explained above, we generated group models with a maximum height of 5, a maximum number of child-groups per group of 3, a maximum number of groups assigned to a viewpoint of 3, and a maximum number of assigned features to groups of 5. The measurements are performed on a laptop with an Intel Core i5-2520M CPU with 2.5GHz, 8GB RAM and on a Windows 7 SP1 64-bit operating system.

Fig. 5 (a) shows the influence of the number of viewpoints on the performance of the consistency check, whereas the time on the x-axis is displayed logarithmically. The measurements were performed on multi-perspective models with a varying size of viewpoints combined with varying feature models. The results show that for higher number of viewpoints, the consistency checks took more time, and the heuristical algorithm performs much better. In Fig. 5 (b), we show the influence of the feature model size, i.e., the respective number of features on the performance of the consistency algorithm, where the time on the x-axis is displayed logarithmically. We see a significant performance increase for the heuristic algorithm compared to the brute force approach as expensive satisfiability checks are performed infrequently by the heuristic algorithm. Only in the worst case only, i.e., if all viewpoints are inconsistent, the performance of the heuristic converges to that of the brute force approach. Summarizing, our performance evaluation confirms, that the incremental consistency check is applicable for large feature models.

6 Related Work

Various authors propose to use feature model views in configuration processes [3,6,20,14,16,21,31,37]. They address MDSoC, where each view restricts the features visible to a stakeholder. Each stakeholder configures parts of the feature model in his view until the entire variability is bound. The authors of [21] use perspectives to address such a stakeholder's view. Acher et al. [5] use feature model slicing to create views similar to the previous approaches. Clarke et al. provide a formal framework for feature model views to reason about compatibility and reconciliation of separated views [13], thus addressing the integration of multiple SPLs. Another approach that addresses MDSoC is the concept of multi-dimensional hyperspaces [27]. They group all concerns of stakeholders in multiple dimensions, each encapsulating one concern. Relating hyperspaces to feature modeling, a hyperslice states a view on a feature model. In contrast to these approaches, we integrate multiple views in perspectives to restrict the variant space of a feature model, where we explicitly define which views form a valid perspective using viewpoints. Closely related to our work are approaches that compose views to create integrated views on multiple domain feature models of independent SPLs [4,7,12,30,35,29]. In these approaches, the set of derivable variants of the resulting feature model will contain all variants of the constituents, whereas in our approach, we create a perspective by composing views of the same feature model such that variants derivable from a perspective are subsets of the variants of the feature model. Furthermore, Zaid et al. [38] present a multi-perspective approach for multi-product lines. The authors state, that a feature belonging to one perspective may be related to features in other perspectives. This corresponds to our definition of views. In contrast, we consider perspectives as semantic refinements on feature models.

7 Conclusion

We developed an approach that extends SPL engineering with multi-perspectives on feature models. Our approach especially supports the customization on feature model level. We provided a formalization and implementation of the concepts and presented an efficient algorithm to check consistency properties and applied it to various case studies for evaluation purposes. Our experiences with case studies have shown perspectives to be a promising concept for tailoring the variant space of a domain feature model for various stakeholders' concerns (cf. [33]). We will use our concepts in model-based SPL testing to organize different testing concerns to derive reduced representative variant spaces under test. Furthermore, we plan to adapt the approach in a dynamic SPL to create customized, adaptable reconfiguration spaces for evolvable systems.

Acknowledgements. The presented work is co-funded by the European Social Fund, Federal State of Saxony and SAP AG in the research project #080949335.

References

1. Choco csp solver website. Internet (April 2012), `http://choco.sourceforge.net`
2. Software product line online tools (splot) website. Internet (April 2012), `http://www.splot-research.org`

3. Abbasi, E., Hubaux, A., Heymans, P.: A toolset for feature-based configuration workflows. In: Proceedings of SPLC 2011 (2011)
4. Acher, M., Collet, P., Lahire, P., France, R.: Composing Feature Models. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 62–81. Springer, Heidelberg (2010)
5. Acher, M., Collet, P., Lahire, P., France, R.: Slicing feature models. In: Proceedings of ASE 2011 (2011)
6. Acher, M., Collet, P., Lahire, P., France, R.: Separation of Concerns in Feature Modeling: Support and Applications. In: Proceedings of AOSD 2012 (2012)
7. Aydin, E.A., Oguztuzun, H., Dogru, A.H., Karatas, A.S.: Merging multi-view feature models by local rules. In: Proceedings of SERA 2011 (2011)
8. Batory, D.S.: Feature Models, Grammars, and Propositional Formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
9. Benavides, D., Segura, S., Martín-Arroyo, P.T., Cortés, A.R.: Using Java CSP Solvers in the Automated Analyses of Feature Models. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 399–408. Springer, Heidelberg (2006)
10. Benavides, D., Segura, S., Ruiz-Cortés, A.: Automated analysis of feature models 20 years later: A literature review. Information Systems 35 (2010)
11. Benavides, D., Trinidad, P., Ruiz-Cortés, A.: Automated Reasoning on Feature Models. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 491–503. Springer, Heidelberg (2005)
12. van den Broek, P., Galvão, I., Noppen, J.: Merging feature models. In: Proceedings of SPLC 2010 (2010)
13. Clarke, D., Proença, J.: Towards a theory of views for feature models. In: Proceedings of FMSPLE 2010 (2010)
14. Classen, A., Hubaux, A., Heymans, P.: A formal semantics for multi-level staged configuration. In: Proceedings of VaMoS 2009 (2009)
15. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison-Wesley (2000)
16. Czarnecki, K., Helsen, S., Ulrich, E.: Staged Configuration Using Feature Models. In: Nord, R.L. (ed.) SPLC 2004. LNCS, vol. 3154, pp. 266–283. Springer, Heidelberg (2004)
17. Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L., Goedicke, M.: Viewpoints: A framework for integrating multiple perspectives in system development. International Journal of Software Engineering and Knowledge Engineering (1992)
18. Heidenreich, F., Wende, C.: Bridging the gap between features and models. In: Proceedings of AOPLE 2007 (2007)
19. Heymans, P., Schobbens, P.Y., Trigaux, J.C., Bontemps, Y., Matulevicius, R., Classen, A.: Evaluating formal properties of feature diagram languages. IET Software (2008)
20. Hubaux, A., Heymans, P., Schobbens, P.Y., Deridder, D.: Towards Multi-view Feature-Based Configuration. In: Wieringa, R., Persson, A. (eds.) REFSQ 2010. LNCS, vol. 6182, pp. 106–112. Springer, Heidelberg (2010)
21. Hubaux, A., Heymans, P., Schobbens, P.Y., Deridder, D., Abbasi, E.: Supporting multiple perspectives in feature-based configuration. Software and Systems Modeling (2011)
22. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Tech. Rep. CMU/SEI-90-TR-21, Carnegie Mellon University Pittsburgh, Software Engineering Institute (1990)

23. Karataş, A.S., Oğuztüzün, H., Doğru, A.: Mapping Extended Feature Models to Constraint Logic Programming over Finite Domains. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 286–299. Springer, Heidelberg (2010)
24. Mazo, R., Salinesi, C., Diaz, D., Lora-Michiels, A.: Transforming attribute and clone-enabled feature models into constraint programs over finite domains. In: Proceedings of ENASE 2011 (2011)
25. Nagle, T.T., Holden, R.K.: The strategy and tactics of pricing. Prentice Hall (2002)
26. Nuseibeh, B., Kramer, J., Finkelstein, A.: Viewpoints: meaningful relationships are difficult. In: Proceedings of ICSE 2003 (2003)
27. Ossher, H., Tarr, P.: Multi-dimensional separation of concerns using hyperspaces. Tech. Rep. IBM Research Report 21452, IBM Research (1999)
28. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering - Foundations, Principles, and Techniques. Springer (2005)
29. Reiser, M.O., Weber, M.: Multi-level feature trees. Requirements Engineering 12, 57–75 (2007)
30. Rosenmüller, M., Siegmund, N.: Automating the configuration of multi software product lines. In: Proceedings of VaMoS 2010 (2010)
31. Rosenmüller, M., Siegmund, N., Thüm, T., Saake, G.: Multi-dimensional variability modeling. In: Proceedings of VaMoS 2011 (2011)
32. Schobbens, P.Y., Heymans, P., Trigaux, J.C.: Feature diagrams: A survey and a formal semantics. In: Proceedings of RE 2006 (2006)
33. Schroeter, J., Lochau, M., Winkelmann, T.: Extended version of multi-perspectives on feature models. Tech. Rep. TUD-FI11-07-Dezember 2011, TU Dresden (2011)
34. Schroeter, J., Lochau, M., Winkelmann, T.: Conper: Consistent perspectives on feature models. In: Proceedings of ACME 2012 (2012)
35. Segura, S., Benavides, D., Ruiz-Cortés, A., Trinidad, P.: Automated Merging of Feature Models Using Graph Transformations. In: Lämmel, R., Visser, J., Saraiva, J. (eds.)GTTSE 2008. LNCS, vol. 5235, pp. 489–505. Springer, Heidelberg (2008)
36. Weiss, D.M., Lai, C.T.R.: Software Product-Line Engineering: A Family-Based Software Development Process. Addison-Wesley Professional (1999)
37. White, J., Dougherty, B., Schmidt, D.C., Benavides, D.: Automated reasoning for multi-step feature model configuration problems. In: Proceedings of SPLC 2009 (2009)
38. Zaid, L.A., Kleinermann, F., Troyer, O.D.: Feature assembly framework: towards scalable and reusable feature models. In: Proccedings of VaMoS 2011 (2011)

Generating Better Partial Covering Arrays by Modeling Weights on Sub-product Lines

Martin Fagereng Johansen[1,2], Øystein Haugen[1], Franck Fleurey[1], Anne Grete Eldegard[3], and Torbjørn Syversen[3]

[1] SINTEF ICT, Pb. 124 Blindern, 0314 Oslo, Norway
{Martin.Fagereng.Johansen,Oystein.Haugen,Franck.Fleurey}@sintef.no
[2] Institute for Informatics, University of Oslo, Pb. 1080 Blindern, 0316 Oslo, Norway
[3] Tomra Systems ASA, Drengsrudhagen 2, 1385 Asker, Norway
{Anne.Grete.Eldegard,Torbjorn.Syversen}@tomra.no

Abstract. Combinatorial interaction testing is an approach for testing product lines. A set of products to test can be set up from the covering array generated from a feature model. The products occurring in a partial covering array, however, may not focus on the important feature interactions nor resemble any actual product in the market. Knowledge about which interactions are prevalent in the market can be modeled by assigning weights to sub-product lines. Such models enable a covering array generator to select important interactions to cover first for a partial covering array, enable it to construct products resembling those in the market and enable it to suggest simple changes to an existing set of products to test for incremental adaption to market changes. We report experiences from the application of weighted combinatorial interaction testing for test product selection on an industrial product line, TOMRA's Reverse Vending Machines.

Keywords: Product Lines, Software, Hardware, Testing, Combinatorial Interaction Testing, Evolution.

1 Introduction

A product line is a collection of systems with a considerable amount of software or hardware components in common [14]. The commonality and differences between the systems are usually modeled as a feature model [12]. Testing product lines is a challenge since the number of possible configurations generally grows exponentially with the number of features in the feature model. Yet, one has to ensure that any valid product will function correctly. There is no consensus yet on how to efficiently test product lines, but there are a number of suggested approaches [7].

A first level of product line testing is testing the software or hardware components in isolation to ensure that they function correctly on their own, a technique seen in industry [9]. Still there may be errors in the interaction between the features. Combinatorial interaction testing [4] is a promising approach for performing interaction testing between the features of a product line.

Based on this, we decided to try out combinatorial interaction testing on TOMRA's product line of reverse vending machines. Reverse vending machines handle the return of deposit beverage containers at retail stores such as supermarkets, convenience stores and gas stations. The feature model for the part of their product line we study has 68 features that potentially combine to 435,808 different configurations.

At TOMRA Verilab they are responsible for testing these machines. They already have a set of test products and were interested in applying new theory and techniques from product line testing research to understand the quality of their current test process and to improve it.

We found that their existing test lab covered a high percentage of the possible simple feature interactions. But, when we generated a new test lab from scratch of the same size as the current test lab, we encountered a problem. Even though the generated machines were valid machines that could be constructed, and even though they did test more of the simple interactions between features with the same number of products, they neither resembled any realistic machine that would be found in the market nor did any subset of products cover the most prevalent interactions.

A solution to these problems was to partition the machines in the market into sub-product lines, partially configured feature models; and to assign weights on them reflecting the number of products that are instances of this particular sub-product line. Modeling weights and sub-product lines proved to be a simple and intuitive way to capture relevant domain-knowledge in a feature model. It is close to the way the domain experts reason about the market and what is most important to be verified.

By generating covering arrays by prioritizing interactions according to their weight, we generated products that resemble the products in the market and that covered as many simple interactions as possible. This caused fewer interactions to be covered, but those interactions that were covered were more relevant according to the market situation.

The weighted sub-product line models also gave us an unexpected benefit. It enabled us to set up an evolution process for the test lab to incrementally adapt it to a continually changing market situation.

In addition to the application to TOMRA's product line, we briefly show how the technique can be used on the Eclipse IDE[1] software product line.

The generation, analysis and evolution based on weighted sub-product line models have been implemented in a fully functional tool freely available as open source on the paper's resource website[2]. The generation of covering arrays in the tool is done using the ICPL algorithm, an algorithm we have developed to generate covering arrays from large feature models [10,11].

[1] The Eclipse IDE is provided by the Eclipse Foundation and is independent from the TOMRA case.

[2] `http://heim.ifi.uio.no/martifag/models2012/`. Two example models, covering arrays and weighted sub-product line models are also available on this website.

This paper is structured as follows. In Section 2 we cover relevant background information and related work. In Section 3 we introduce models of weighted sub-product lines that enable covering array generation algorithms to select and evolve collections of products to test. In Section 4 we present the models and experiences from applying the techniques to an industrial product line at TOMRA and in Section 5 briefly describe the applicability to testing Eclipse IDEs. The paper ends with the conclusion, Section 6.

2 Background and Related Work

2.1 Product Lines

As stated in the introduction, a product line is a collection of systems with a considerable amount of hardware or software in common. The primary motivation for structuring one's systems as a product line is to allow customers to have a system tailored for their purpose and needs, while still avoiding redundancy of hardware or code. It is not uncommon for customers to have conflicting requirements. In that case, it is not even possible to ship one system for all customers. In the case of hardware, it would be uneconomical to ship unused components.

The Eclipse IDE products [2] can be seen as a software product line. Today, the Eclipse project lists 12 products on their download page[3]. The configurations of these products are shown in Table 1a[4]. These products share many components, but all components are not offered together as one single product. The reason is that the download would be unnecessary large, since, for example, a C++ systems programmer usually does not need to use the PHP-related features. It would also bloat the system by giving the user many unnecessary alternatives when, for example, creating a new project. Some products contain early developer releases of some components, such as Eclipse for modeling. Including these would compromise the stability for the other products. Thus, it should be clear why offering specialized products for different use cases is good.

One way to model the commonalities and differences in a product line is using a feature model [12]. A feature model sets up the commonalities and differences of a product line in a tree such that configuring the product line proceeds from the root of the tree. Please refer to an example of a feature model for a subset of Eclipse in Figure 1.

Proceeding from the root, configuring the product line consists of making a decision for each node in the tree. Each node represents a feature of the product line. The nature of this decision is modeled as a decoration on the edges going from a node to another. For example, in Figure 1, a filled circle means that the feature is mandatory, and an empty circle means that it is optional. A filled semi-circle on the outgoing edges means that at least one of the features underneath must be selected. An empty semi-circle means that one and only one must be

[3] `http://eclipse.org/downloads/` as of 2012-03-09.

[4] The original version of this table was found at `http://www.eclipse.org/downloads/compare.php`, 2012-03-28.

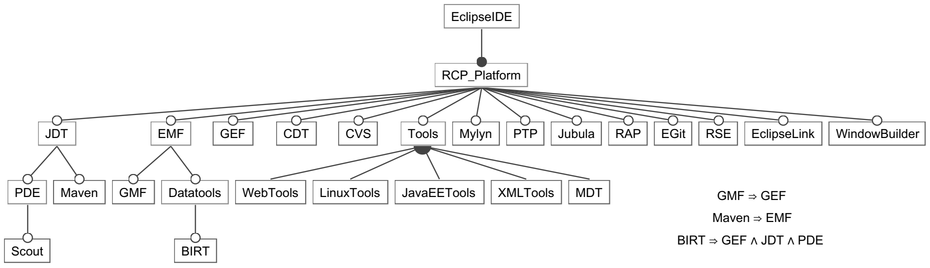


Fig. 1. Feature model for a significant part of the Eclipse IDE product line supported by the Eclipse Project

selected. In addition, constraints not effectively modeled on the tree are written underneath the model as propositional constraints; for example, when GMF is selected, it implies that GEF must also be selected.

The parts that can be different in the products of a product line are usually called its *variability*. One particular product in the product line is called a *variant* and is specified by a configuration of the feature model. A configuration consists of specifying whether each feature is included or not.

2.2 Product Line Testing

Testing a product line poses a number of new challenges compared to testing single systems. It has to be ensured that each possible configuration of the product line functions correctly. One way to verify a product line is through testing, but testing is done on a running system. The product line is simply a collection of many products. The number of possible configurations generally grows exponentially with the number of features in the feature model. For the feature model in Figure 1, the number of possible configurations is 1,900,544, and this is a relatively simple product line.

There is no single recommended approach available today for testing product lines efficiently [7], but there are many suggestions. Some of the more promising suggestions are combinatorial interaction testing [4], discussed below; reusable component testing, which we have seen in industry [9], but which does not test for interaction faults in the product line; a technique called ScenTED, where the idea is to express the commonalities and differences on the UML model of the product line and then derive concrete test cases by analyzing it [15]; and incremental testing, where the idea is to automatically adapt a test case from one product to the next using the specification of similarities and differences between the products [16].

2.3 Combinatorial Interaction Testing for Product Lines

Combinatorial interaction testing [4] is one of the most promising approaches. The benefits of this approach is that it deals directly with the feature model to derive a small subset of products which can then be tested using single system testing techniques, of which there are many good ones [3]. The idea is to select a small subset of products where the interaction faults are most likely to occur. For example, we can select the subset of all possible products where each pair of features is present. This includes the cases where both features are present; when one is present and when none of the two are present. Table 1b shows the 12 products that must be tested to ensure that every interaction between two features in the running example functions correctly, a 2-wise covering array. Each row represents one feature and every column one product. 'X' means that the feature is included for the product, '-' means that the feature is not included. Some features are included for every product because they are mandatory, and some pairs are not covered since they are invalid according to the feature model.

Table 1. Eclipse IDE Products, Instances of the Feature Model in Figure 1

(a) Eclipse IDE Products

Feature\Product	1	2	3	4	5	6	7	8	9	10	11	12
EclipseIDE	X	X	X	X	X	X	X	X	X	X	X	X
RCP_Platform	X	X	X	X	X	X	X	X	X	X	X	X
CVS	X	X	X	X	X	X	X	X	X	X	X	X
EGit	-	-	X	X	X	X	-	-	-	-	-	-
EMF	X	X	-	-	-	X	X	-	-	-	-	-
GEF	X	X	-	-	-	X	X	-	-	-	-	-
JDT	X	X	-	-	X	X	X	-	X	-	-	X
Mylyn	X	X	X	X	X	X	X	X	X	X	X	-
Tools	X	X	X	X	X	X	X	X	-	-	X	-
WebTools	-	X	-	-	-	-	X	-	-	-	X	-
LinuxTools	-	-	X	X	-	-	-	X	-	-	-	-
JavaEETools	-	X	-	-	-	-	X	-	-	-	-	-
XMLTools	X	X	-	-	X	-	X	X	-	-	-	-
RSE	-	X	X	X	-	-	X	X	-	-	-	-
EclipseLink	-	X	-	-	-	-	X	-	-	X	-	-
PDE	-	X	-	-	X	X	X	-	X	-	-	X
Datatools	-	X	-	-	-	-	X	-	-	-	-	-
CDT	-	-	X	X	-	-	-	X	-	-	-	-
BIRT	-	-	-	-	-	-	X	-	-	-	-	-
GMF	-	-	-	-	-	X	-	-	-	-	-	-
PTP	-	-	-	-	-	-	-	X	-	-	-	-
MDT	-	-	-	-	-	X	-	-	-	-	-	-
Scout	-	-	-	-	-	-	-	-	X	-	-	-
Jubula	-	-	-	-	-	-	-	-	-	X	-	-
RAP	-	-	-	-	X	-	-	-	-	-	-	-
WindowBuilder	X	-	-	-	-	-	-	-	-	-	-	-
Maven	X	-	-	-	-	-	-	-	-	-	-	-

(b) Complete 2-wise Covering Array

Feature\Product	1	2	3	4	5	6	7	8	9	10	11	12
EclipseIDE	X	X	X	X	X	X	X	X	X	X	X	X
RCP_Platform	X	X	X	X	X	X	X	X	X	X	X	X
CVS	X	X	-	-	X	-	-	X	-	X	-	-
EGit	X	X	X	-	-	-	X	X	-	-	-	-
EMF	X	X	X	-	X	-	-	X	X	-	X	X
GEF	X	X	-	-	X	-	X	X	X	-	X	-
JDT	X	-	X	X	X	-	-	X	X	-	-	-
Mylyn	-	X	X	X	-	-	-	X	-	-	-	-
Tools	X	X	X	X	X	X	X	X	X	-	-	-
WebTools	X	-	X	X	-	-	X	-	X	-	-	-
LinuxTools	X	-	X	-	-	X	X	-	X	-	-	-
JavaEETools	X	-	-	X	-	X	-	-	X	-	-	-
XMLTools	-	X	X	-	X	-	X	-	X	-	-	-
RSE	-	X	X	-	-	X	-	X	-	X	-	-
EclipseLink	-	X	X	-	-	X	-	X	X	-	-	-
PDE	X	-	-	X	X	-	-	X	X	-	-	-
Datatools	X	X	-	-	X	-	-	X	X	-	-	X
CDT	X	-	X	-	X	X	-	-	-	-	-	-
BIRT	X	-	-	-	X	-	-	X	X	-	-	-
GMF	X	X	-	-	X	-	-	-	X	-	X	-
PTP	-	-	X	-	X	X	X	-	-	-	-	-
MDT	X	-	X	X	-	-	X	X	-	-	-	-
Scout	-	-	-	X	X	-	-	X	X	-	-	-
Jubula	-	-	X	X	-	X	-	X	X	-	-	-
RAP	X	-	X	-	-	X	-	X	-	-	-	-
WindowBuilder	-	X	X	-	X	-	-	X	X	X	-	-
Maven	X	-	X	-	X	-	-	-	X	-	-	-

2-wise covering arrays are a special case of t-wise covering arrays where $t = 2$. 1-wise coverage means that every feature is at least included and excluded in at least one product. 3-wise coverage means that every combination of three features are present, etc. For our running example, 2, 12 and 37 products are sufficient to achieve 1, 2 and 3-wise coverage, respectively.

An important motivation for combinatorial interaction testing is a paper by Kuhn et al. 2004 [13]. They indicated empirically that most bugs are found for 6-wise coverage, and that for 1-wise one is likely to find on average around 50%, for 2-wise on average around 70%, and for 3-wise around 95%, etc. Kuhn et al.

2004 result, however is not about combinations of features, but about combinations of program input. A recent study by Garvin and Cohen 2011 [8] checked whether Kuhn et al. 2004's result also holds for feature interaction faults. They investigated 250 faults of two real-world, open source systems. Of these faults 28 were found to be configuration dependent and three to be true interaction faults. In addition, they conclude that exercising feature interactions traverses more of the product line's behavior. This indicates that Kuhn et al. 2004's result is also applicable for feature interaction faults.

There are three main stages in the application of combinatorial interaction testing to a product line. First, the feature model of the system must be made. Second, the t-wise subset of products must be generated. We have developed an algorithm that can generate such arrays from large features models [11]. These products must then be generated or physically built. Last, a single system testing technique must be selected and applied to each product in this covering array.

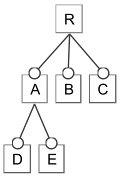
3 Weighted Combinatorial Interaction Testing

As explained earlier, in order to successfully apply combinatorial interaction testing at TOMRA, we had to extend the technique by developing and using weighted sub-product line models. In this section, we describe the models and how they are used on a simple example, before discussing the more complex details and evaluations for the application at TOMRA in the next section.

Ordinary Combinatorial Interaction Testing. Figure 2a shows a simple feature model with 6 features. When applying ordinary combinatorial interaction testing, we would, for example, generate a 2-wise covering array (as in Table 2b), build the products and test them individually. That way, we know that all interactions between pairs of features have been tested.

Table 2. A Simple Example

(a) Feature Model



(b) Complete 2-wise covering array

	1	2	3	4	5	6
R	X	X	X	X	X	X
A	X	X	X	X	-	-
B	-	X	-	X	-	X
C	-	X	-	X	X	-
D	X	-	-	X	-	-
E	X	X	-	-	-	-

(c) Weighted Sub-product lines

	1	2
#Weight	100	10
R	X	X
A	-	?
B	X	X
C	?	X
D	-	?
E	-	?

Initial Problems. At this stage of the application of combinatorial interaction testing at TOMRA we faced a few problems:

– There were no ways to select a subset of the products to test that included the most important interactions. One could of course select a subset that covered as many interactions as possible, but it might not include some important interactions.
– The covering array generation algorithm generates a different result each time it is run. Since there are many equally good products, why not select the ones that look like some of the larger classes of sold products?

A Solution. These experiences revealed to us that an assumption of ordinary combinatorial interaction testing, that all interactions are equal, is not entirely the case in practice.

For TOMRA, there were certain market segments where the products are similar. It is important for TOMRA that a product containing the commonalities of these segments are tested. Thus, we decided: Let us model the products in each market segment as a sub-product line, and assign a weight to it according to how many products of that kind are in the market.

This will enable us to assign weights to each interaction in the sub-product line. These weights can then be used by a covering array generator to select the interactions with the most weight to cover first. This would cause the results to be similar each time the covering array algorithm is run and the first products produced would contain the most important interactions.

Weighted Sub-product Lines. For our simple example, there are two major market segments, modeled in Table 2c. These two sub-product lines are as follows: The first has R and B included and A, D and E excluded. The feature C is optional within this segment. The second segment has features R, B and C included while A, D and E are optional within this segment. Now, in this second example, the limitations imposed on the products by the feature model in Figure 2a still apply, of course. So, even though A, D and E are marked with a question mark, the products with A excluded also has D and E excluded. The first segment have 100 instances in the market while the second segment has only 10.

Now, what about legal products that are not in any market segment? For this example, a product with R included and B excluded is not found in the market. As we experienced at TORMA, the reason that they are not found is that they do not make sense even though they are structurally valid. This is valuable domain knowledge, and is of great value to a covering array generator: It enables it not to focus on the interactions that are of no practical importance.

Algorithms. An example should clarify how the covering array generation algorithm can use the weighted sub-product line models. This example uses 1-wise covering arrays since an example with 2-wise covering arrays would fill up several pages due to the combinatorial explosion of interactions. This is not a problem for modern computers to deal with however.

We will use the terminology from a previous paper of ours [11]: An *assignment* is a pair with a feature name and a boolean. A *t-set* is a set of t assignments. A *configuration* is a set of assignments in which all features of the product line are given an assignment. The *universe* to cover is the set of all valid t-sets, U_t. Thus, a *t-wise covering array* is a set of configurations, C_t, such that $\forall e \in U_t, \exists c \in C_t : e \subseteq c$. A t-set can be written as for example $\{(F1, X), (F2, -)\}$, a 2-set with $F1$ included and $F2$ excluded.

For our simple example, the following 11 t-sets need to be in a product to achieve 1-wise coverage: $\{\{(R, X)\}, \{(A, X)\}, \{(A, -)\}, \{(B, X)\}, \{(B, -)\}, \{(C, X)\}, \{(C, -)\}, \{(D, X)\}, \{(D, -)\}, \{(E, X)\}, \{(E, -)\}\}$. Note that the assignment with R excluded is not present since in feature modeling the root must always be included.

Now, we can assign weights to each t-set. In Table 2c, whenever a t-set is present in a sub-product line, the weight is added to the t-set. If there is a question-mark, half the weight is given to each assignment. For example (A, X) gets 5 because it is not present in the first sub-product line and only as an option in the second. One t-set is not present in any sub-product line and therefore gets the weight zero: $\{ (\{(R, X)\}, 110), (\{(A, X)\}, 5), (\{(A, -)\}, 105), (\{(B, X)\}, 110), (\{(B, -)\}, 0), (\{(C, X)\}, 60), (\{(C, -)\}, 50), (\{(D, X)\}, 5), (\{(D, -)\}, 105), (\{(E, X)\}, 5), (\{(E, -)\}, 105)\}$.

These t-sets can now be ordered according to their weights: $\{ (\{(R, X)\}, 110), (\{(B, X)\}, 110), (\{(A, -)\}, 105), (\{(E, -)\}, 105), (\{(D, -)\}, 105), (\{(C, X)\}, 60), (\{(C, -)\}, 50), (\{(A, X)\}, 5), (\{(D, X)\}, 5), (\{(E, X)\}, 5), (\{(B, -)\}, 0)\}$.

Now, the circumstances warrants two kinds of coverages. The ordinary type of coverage is *t-set coverage*. The goal of combinatorial interaction testing is to cover as many simple interactions as possible, and ultimately a t-set coverage of 100%, that is, cover all t-sets. Since we have introduced weights for each t-set, talking about *weight coverage* makes sense. The goal of weight coverage is then to cover the most weight possible, and ultimately cover all the t-sets with weight, that is, achieve 100% weight coverage.

Just as t-set coverage is found by taking the number of covered t-sets and dividing it by the total number of valid t-sets, $|U_t|$, similarly weight coverage is found by taking the covered weight divided by the total weight.

The total weight of our example is the sum of all the weights of all the t-sets: 660. Now, if we were to generate a single product to test from the weighted t-sets, we would get the following: $\{\{(R, X)\}, \{(A, -)\}, \{(B, X)\}, \{(C, X)\}, \{(D, -)\}, \{(E, -)\}\}$. The weight covered by this product is the sum of all weights of the covered t-sets: 595. Thus, the weight coverage of this single product is $595/660 \approx 90\%$. This number can be contrasted with the t-set coverage of this product which is $6/11 \approx 55\%$. The high weight coverage indicates to us that we have most of the important t-sets (given the current market situation) are in the product. Any valid product would, however, give the same t-set coverage, but only that product would give such a high weight coverage.

Note that 100% weight coverage can mean that we have less than 100% t-set coverage since some t-sets can have zero weight. To ensure that 100% weight coverage means 100% t-set coverage, include a sub-product line with all question marks and a non-zero weight.

Evolution of Test Products. A goal of testing is to gain confidence in the products that are sold to or used by the customers. Weights on sub-product lines can be set up to reflect the market situation, but could also include expected sales. When the market situation or the expectations change, the weights and the sub-product lines can change. This does not mean that the test lab or the feature model is changed. It will, however, mean that the weight coverage of the products that are currently being tested changes.

A simple algorithm can suggest simple changes to a set of test products. By calculating the coverage of the current test products, and then the new coverage given 1, 2 or 3 (or even more) changes of it, a list of possible changes can be made[5]. If the best changes are applied to the lab incrementally, the test products can evolve over time to converge on the current market situation, even if it changes during the evolution.

A special case of this is the introduction of a new feature in the feature model. The expected sales of this new feature can be added to the weighted sub-product line models. Including it for at least one of the products in the set of test products will probably be the best way to increase weight coverage. Thus, this decision is automated by our approach.

Sub-Product Lines, Related Work. Czarnecki et al. 2004 [5] introduced the idea of staged configuration. The stages are the production of a new sub-product line from a previous one. The difference between their work and ours is that we apply sub-product lines to modeling the market situation for use in testing, while they use it during product line development.

Their ideas are further developed in Czarnecki et al. 2005 [6]. Our view is similar to theirs in that the sub-product lines are specializations of the complete feature model to certain market segments. It is on the basis of these specializations that domain experts do their daily work.

Batory 2005 [1] integrates the idea of staged configurations with the formalization of feature models as propositional constraints. He formalizes feature models as propositional formulas. In his work, a sub-product line is a propositional formula where some variables, representing features, set to 'true' for included, some set to 'false' for excluded, and the unset features set to 'unknown'. The 'unknown' classification is the same as the questions-marks in our sub-product line models.

4 Industrial Application: TOMRA

In this section, we report from an industrial application of our technique at TOMRA Verilab.

[5] An implementation of this algorithm is available on the paper's resource website. It supports searching for 1–3 changes for improving 1–3 wise coverage.

About TOMRA Verilab. TOMRA Verilab is the part of TOMRA that is responsible for testing TOMRA's reverse vending machines (RVMs). Reverse vending machines handle the return of deposit beverage containers at retail stores such as supermarkets, convenience stores and gas stations. In Norway, customers are required by law to pay an amount for each container they buy which is given back to them if they decide to return the container.

RVMs. The RVMs are delivered all over the world and the market is expanding. However, individual market requirements and the needs of TOMRA's customers within the different markets can very significantly. TOMRA's reverse vending portfolio therefore offers a high degree of flexibility in terms of how a specific installation is configured.

Figure 2 shows a part of the feature model for TOMRA RVMs. The feature model has 68 features, and a huge number of possible configurations (435,808, to be exact). Variation can include such things as the quality of the display used (e.g. black and white, color, touch-screen interface), the type of storing and sorting facilities the system has, as well as different container recognition technologies utilized for identifying container security marks, material and other characteristics.

Test Lab. In order to test the RVMs, TOMRA Verilab has set up a test lab of machines configured by hand to ensure the quality of the machines in the market. Parts of these product configurations are shown in Table 3. These products are both automatically and manually tested. The software is partly tested automatically by installing and running test suites on the machines. The manual tests are run by, for example, inserting bottles of various kinds in various ways, orders and magnitudes.

Sub-Product Lines and Weights. As discussed earlier, the results produced by ordinary covering array generation was not suited for TOMRA for various reasons. To solve these, we modelled the weighted sub-product lines as partly shown in Table 4.

Existing Coverages. The first experiment was to measure the t-set and weight coverage of the existing test lab, shown in part in Table 3. Recall that coverage is measured by taking the covered valid t-sets and then dividing either their number or their weight by the total number or t-sets or the total weight, respectively.

Figure 3a shows the 1–3-wise, t-set and weight coverage for the existing test lab. The weight coverage is consistently higher for the weight coverage than for the t-set coverage. This is consistent with the fact that the developers paid attention to the market situation when designing the test lab manually. It also suggests that weighted sub-product line models are a guide to test product selection for TOMRA Verilab.

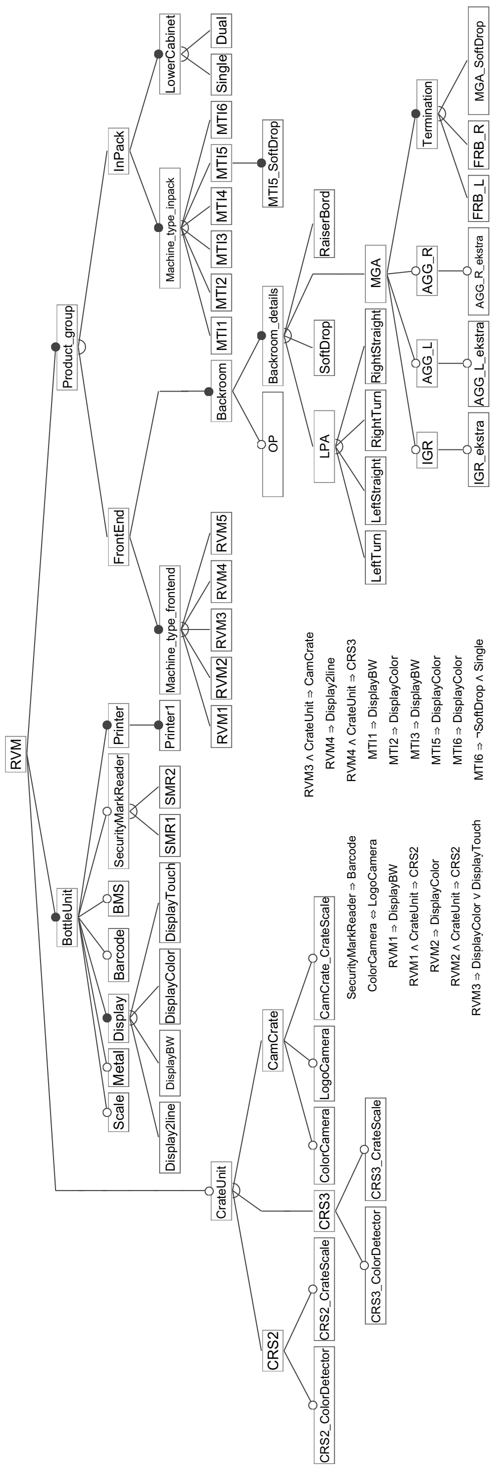


Fig. 2. Part of the Feature model of TOMRA's Product Line of Reverse Vending Machines

Table 3. Part of TOMRA's Actual Test Lab

Feature\Product	1	2	3	4	5	6	7	8	9	10	11	12
RVM	X	X	X	X	X	X	X	X	X	X	X	X
CrateUnit	X	-	X	-	-	-	X	X	X	-	-	X
...												
BottleUnit	X	X	X	X	X	X	X	X	X	X	X	X
Display	X	X	X	X	X	X	X	X	X	X	X	X
Display2line	-	-	X	-	-	-	-	-	-	-	-	-
DisplayBW	X	X	-	X	X	-	-	-	-	-	-	-
DisplayColor	-	-	-	-	-	X	X	X	X	X	X	-
DisplayTouch	-	-	-	-	-	-	-	-	-	-	-	X
Scale	-	X	X	X	X	X	X	X	X	X	X	X
Metal	X	X	-	X	X	X	X	X	X	X	X	X
Barcode	X	X	-	X	X	X	X	X	X	X	X	X
BMS	-	-	-	-	X	-	-	-	-	-	-	-
SecurityMarkReader	-	X	-	X	-	-	X	-	-	-	X	X
SMR1	-	X	-	-	-	-	-	-	-	-	-	-
SMR2	-	-	-	X	-	-	X	-	-	-	X	X
Printer	X	X	X	X	X	X	X	X	X	X	X	X
Printer1	X	X	X	X	X	X	X	X	X	X	X	X
Product_group	X	X	X	X	X	X	X	X	X	X	X	X
FrontEnd	X	X	X	-	-	-	X	X	X	-	-	X
...												
Backroom	X	X	X	-	-	-	X	X	X	-	-	X
Backroom_details	X	X	X	-	-	-	X	X	X	-	-	X
OP	-	-	-	-	-	-	-	X	-	-	-	-
LPA	-	X	-	-	-	-	-	-	-	-	-	-
...												
SoftDrop	-	-	-	-	-	-	-	-	-	-	-	-
...												
RaiserBord	X	-	X	-	-	-	-	X	X	-	-	-
...												

Generated Coverages. The second experiment was to generate a partial covering array from scratch using both t-set and weight coverage and compare them to each other. The results are shown in Figure 3b. We can see that for 1–3-wise covering arrays, the weighted covering arrays are consistently smaller than t-set-based covering arrays. This suggests that either it would have been beneficial to used weighted covering array generation from the start, or that the current test lab is outdated with respect to the current market situation.

Suggesting Improvements. The third experiment was to find small modifications that can be done on the existing test lab at TOMRA to increase the weight coverage. Some such suggestions are shown in Table 5.

The search for improvements is done by flipping a set of assignments and, if the new configuration is valid, recalculating the new weight coverage. If the coverage is better than the original one, the changes and the new coverage are recorded.

In Table 5, we have recorded some suggestions for improving the 2-wise weight coverage of the existing test lab. The original 2-wise weight coverage was 95.8% (Table 3). We can achieve an increase of 0.2 pp by excluding feature *Metal* for product 11, an increase of 0.7 pp by including feature *SoftDrop* and excluding feature *RaiserBord* for product 1. Finally, we achieve an increase of 0.8 pp by excluding *Metal*, including *SoftDrop* and excluding *RaiserBord* for product 1.

Table 4. Part of TOMRA's Sub-Product Lines and Their Weights

Feature\Product	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
#Weight	478	478	1140	500	50	33	33	25	75	120	58	1	81	81	525	100	1500	1225	125
RVM	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
CrateUnit	-	X	?	X	X	-	X	-	X	?	?	?	-	X	-	-	-	-	-
...																			
BottleUnit	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Display	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Display2line	-	-	-	-	-	-	-	-	-	-	-	-	X	X	-	-	-	-	-
DisplayBW	?	?	?	?	?	-	-	-	-	-	-	-	-	-	-	?	?	?	-
DisplayColor	?	?	?	?	?	X	X	?	?	?	?	?	-	-	X	?	?	?	X
DisplayTouch	-	-	-	-	-	-	-	?	?	?	?	?	-	-	-	-	-	-	-
Scale	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
Metal	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?
Barcode	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	X
BMS	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	?	-	-	-
SecurityMarkReader	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	X
SMR1	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	-
SMR2	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	?	X
Printer	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Printer1	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
Product_group	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
FrontEnd	X	X	X	X	X	X	X	X	X	X	X	X	X	X	-	-	-	-	-
...																			
Backroom	X	X	X	X	X	X	X	X	X	X	X	X	X	X	-	-	-	-	-
Backroom_details	X	X	X	X	X	X	X	X	X	X	X	X	X	X	-	-	-	-	-
OP	?	?	-	-	-	?	?	?	?	-	-	-	?	?	-	-	-	-	-
LPA	-	-	X	-	-	?	?	-	-	X	-	-	?	?	-	-	-	-	-
...																			
SoftDrop	?	?	-	-	-	?	?	?	?	-	-	-	?	?	-	?	?	?	-
...																			
RaiserBord	?	?	-	-	-	?	?	?	?	-	-	-	?	?	-	-	-	-	-
...																			

Generating the suggestions on our machine[6] took 49s, 141s and 1,090s for an improvement in 2-wise covering of 1, 2 and 3 suggestions of changes respectively.

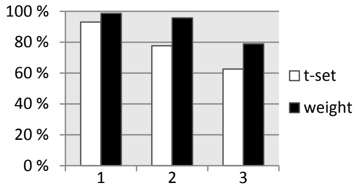
The next set of runs (4–6) is on an improved version of the original test lab. We chose to apply suggestion 1, to exclude feature *Metal* for product 11, and got a new test lab of 2-wise weight coverage of 96.0%. The improvements can be read in the same way as the previous set of suggestions.

The next set of runs (7–9) is on another improved version of the original test lab. We chose to apply suggestion 2 to the original lab, to include feature *SoftDrop* and exclude feature *RaiserBord* for product 1, and got a new test lab of 2-wise weight coverage of 96.5%.

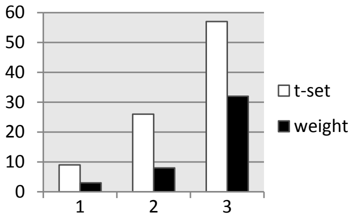
Finally, we applied suggestions 1 and 2 to the original test lab to get a coverage of 96.3%. This produces suggestions 10–12.

The best coverage was achieved by changing four features by applying suggestions 2 and 9 to get a new weight coverage of 97.2%, up 1.4pp from 95.8%.

[6] Its specifications: Intel Q9300 CPU @2.53GHz and 8 GiB, 400MHz RAM. Each execution ran in parallel in 4 threads, as the computer had 4 logical processors.



(a) t-set and Weight Coverage of Original Lab



(b) Size of Labs with 95% t-set and Weight Coverage

Fig. 3. Results of Two Experiments

Table 5. Simple changes to TOMRA's test lab, Table 3, that produce higher coverage of the product line, Figure 2, based on the current market situation, Table 4

Change Suggestion	New Coverage	Product	Feature 1	Set	Feature 2	Set	Feature 3	Set
Starting from the lab machines with coverage 95.8%								
1	96.0%	11	Metal	-				
2	96.5%	1	SoftDrop	X	RaiserBord	-		
3	96.6%	1	Metal	-	SoftDrop	X	RaiserBord	-
Starting from lab machines with suggestion 1, with coverage 96.0%								
4	96.3%	10	Barcode	-				
5	96.7%	1	SoftDrop	X	RaiserBord	-		
6	96.9%	1	Barcode	-	SoftDrop	X	RaiserBord	-
Starting from lab machines with suggestion 2, with coverage 96.5%								
7	96.7%	11	Metal	-				
8	97.0%	11	Metal	-	Scale	-		
9	97.2%	10	Metal	-	Scale	-	Barcode	-
Starting from lab machines with suggestion 1 and 4, with coverage 96.3%								
10	96.5%	11	Scale	-				
11	97.0%	1	SoftDrop	X	RaiserBord	-		
12	96.5%	3	Scale	-	SoftDrop	X	RaiserBord	-

5 Applicability to the Eclipse IDEs

As an indication of the generality of our approach, we did an experiment to see if it also made sense for the product line of Eclipse IDEs. The Eclipse IDE can be seen as a product line; it was introduced in Section 2. The actual products offered on the Eclipse website was shown in Table 1a.

One source of information about what the users of the Eclipse IDE have is the download statistics reported on the Eclipse project's download pages[7]. These can be used as weights[8]. The weights can be assigned to the product configurations themselves, which were previously shown in Table 1a. Table 6 shows the downloads as weights linked to each of the Eclipse products in Table 1a. In this table we can clearly see that some products are more downloaded than others.

[7] `http://eclipse.org/downloads/` as of 2012-03-09.

[8] A better source of weights and sub-product lines is the data aquired by the Eclipse Usage Data Collector (UDC) that "[...] collects information about how individuals are using the Eclipse platform," available online at `eclipse.org/org/usagedata/`.

Table 6. Eclipse IDE Product Configurations and Their Downloads as of 2012-03-09

Product	Name	Weight	Product	Name	Weight
1	Java	282,220	7	Reporting	33,813
2	JavaEE	856,493	8	Parallel	10,441
3	C/C++	58,720	9	Scout	1,130
4	C/C++ Linux	58,720	10	Testers	8,953
5	RCP/RAP	16,610	11	JavaScript	35,750
6	Modeling	22,060	12	Classic	651,616

By generating a 2-wise covering array using the feature model in Figure 1 with the weights from Table 6 on the product configurations in Table 1a, we found that just 4 products give more that 95% weight coverage. This clearly shows that our approach is most likely applicable outside the scope of the TOMRA industrial case.

6 Conclusion

In this paper we showed how an additional type of model was needed in order to effectively apply combinatorial interaction testing to an industrial product line. The new model captures relevant domain knowledge in a form that is close to the way domain experts reason about their domain and enables additional benefits to be derived from combinatorial interaction testing:

- Cover the interactions found in the market or planned to be in future products first.
- Generate products that both cover many simple interactions and resemble products found in the market.
- Incrementally evolve the test products for the continually changing market situation.
- Covering array generation is more deterministic.

We described our experiences of applying this to a product line of industrial size and complexity, the TOMRA RVMs.

The algorithms that implement these features in addition to the ordinary combinatorial interaction testing features are available on the paper's resource website as free and open source software.

Acknowledgments. The work presented here has been developed within the VERDE project ITEA 2 - ip8020. VERDE is a project within the ITEA 2 - Eureka framework.

References

1. Batory, D.: Feature Models, Grammars, and Propositional Formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)

2. Beaton, W., Rivieres, J.: Eclipse platform technical overview. Tech. rep., The Eclipse Foundation (2006)
3. Binder, R.V.: Testing object-oriented systems: models, patterns, and tools. Addison-Wesley Longman Publishing Co., Inc., Boston (1999)
4. Cohen, M.B., Dwyer, M.B., Shi, J.: Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. IEEE Transactions on Software Engineering 34, 633–650 (2008)
5. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged Configuration Using Feature Models. In: Nord, R.L. (ed.) SPLC 2004. LNCS, vol. 3154, pp. 266–283. Springer, Heidelberg (2004)
6. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Staged configuration through specialization and multilevel configuration of feature models. Software Process: Improvement and Practice 10(2), 143–169 (2005)
7. Engström, E., Runeson, P.: Software product line testing - A systematic mapping study. Information and Software Technology 53(1), 2–13 (2011)
8. Garvin, B., Cohen, M.: Feature interaction faults revisited: An exploratory study. In: IEEE 22nd International Symposium on Software Reliability Engineering (ISSRE), pp. 90–99 (29 2011-December 2 2011)
9. Johansen, M.F., Haugen, Ø., Fleurey, F.: A Survey of Empirics of Strategies for Software Product Line Testing. In: O'Conner, L. (ed.) Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2011, pp. 266–269. IEEE Computer Society, Washington, DC (2011)
10. Johansen, M.F., Haugen, Ø., Fleurey, F.: Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 638–652. Springer, Heidelberg (2011)
11. Johansen, M.F., Haugen, Ø., Fleurey, F.: An Algorithm for Generating t-wise Covering Arrays from Large Feature Models. In: Alves, V., Santos, A. (eds.) Proceedings of the 16th International Software Product Line Conference (SPLC 2012), ACM (2012)
12. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (foda) feasibility study. Tech. rep., Carnegie-Mellon University Software Engineering Institute (November 1990)
13. Kuhn, D.R., Wallace, D.R., Gallo, A.M.: Software fault interactions and implications for software testing. IEEE Transactions on Software Engineering 30(6), 418–421 (2004)
14. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus (2005)
15. Reuys, A., Reis, S., Kamsties, E., Pohl, K.: The scented method for testing software product lines. In: Käköla, T., Duenas, J.C. (eds.) Software Product Lines, pp. 479–520. Springer, Heidelberg (2006), doi:10.1007/978-3-540-33253-4_13
16. Uzuncaova, E., Khurshid, S., Batory, D.: Incremental test generation for software product lines. IEEE Transactions on Software Engineering 36(3), 309–322 (2010)

Towards Business Application Product Lines

Vinay Kulkarni, Souvik Barat, and Suman Roychoudhury

Tata Research Development and Design Centre
54B, Industrial Estate, Hadapsar
Pune, India
{vinay.vkulkarni,souvik.barat,suman.roychoudhury}@tcs.com

Abstract. With continued increase in business dynamics, it is becoming increasingly harder to deliver purpose-specific business systems in the ever-shrinking window of opportunity. Code-centric software product line engineering (SPLE) techniques show unacceptable responsiveness as business applications are subjected to changes along multiple dimensions that continue to evolve simultaneously. Through clear separation of functional concerns from technology, model-driven approaches enable easy delivery of the same functionality into multiple technology platforms. However, business systems for same functional intent tend to have similar but non-identical functionality. This makes a strong case for bringing in SPLE ideas i.e., *what* can change *where* and *when*, to models. We propose an abstraction that aims to address composition, variability and resolution in a unified manner; describe its model-based realization; and outline the key enablers necessary for raising business application product lines. Early experience of our approach and issues that remain to be addressed for industry acceptance are highlighted.

Keywords: software product lines, model driven engineering.

1 Introduction

We are in the business of developing business-critical software systems, typically for large enterprises. These systems are characterized by low algorithmic complexity, database intensive operation, large size, and distributed architecture. The large size of a typical business application leads to large development team that needs to work in a coordinated manner. Choice of distributed architecture paradigm necessitates multiple technologies to be managed effectively, and moreover, many times the customer has non-negotiable technology platform preferences. Being business critical in nature, the solution needs to be delivered quickly and is expected to be in use for a long time. Given the increased business and technology dynamics, the latter poses a significant architectural challenge. Our experience is no two solutions, even for the same business intent such as straight-through-processing of trade orders, back-office automation of a bank, automation of insurance policies administration etc, are identical [22]. Though there exists a significant overlap across functional requirements for a given business intent, the variations are manifold too. Moreover, our management expects delivery of subsequent solutions for the same business intent to be significantly faster, better and cheaper.

We have witnessed that business applications tend to vary along three dimensions:

- Functionality dimension which can be further divided into Business rules and Business logic sub-dimensions
- Business process dimension which can be further divided into Process tasks, Organizational policies, and Organizational structure sub-dimensions
- Solution architecture dimension which can be further divided into Design decisions, Technology platform, and Implementation architecture sub-dimensions

Each [sub]dimension can be seen as a set of multiple choice questions. Arriving at the desired implementation is essentially the task of selecting suitable answers for the relevant questions, identifying / implementing the code fragments, and stitching them together in a consistent manner. Some dimensions tend to vary more frequently over time than others. For instance, changes along the Technology platform dimension are more rapid than changes along the Business logic dimension. A given set of choices along Functionality dimension are needed to be delivered with a different set of choices along Business process and Solution architecture dimensions. There is a natural interplay between the set of choices wherein a choice along a dimension eliminates (or forces) a set of choices along other dimensions. For instance, choice of 'rural India' geography for a banking system may force 'hosted services platform' choice of Implementation architecture; Choice of Java programming language and Oracle database as persistent store may force 'Object relational mapping' choice for design strategy etc. We witnessed that a choice along a dimension can impact multiple program locations (i.e. scattering) and choices along a set of dimensions can impact the same program location (i.e. tangling). For instance, choices for a set of strategies such as concurrency of a database table row (corresponding to a persistent object), object-relational mapping, and preserving audit trail all impact the *create()* method implementation for a persistent class. And object-relational mapping strategy impacts definitions of all classes in the hierarchy.

Based on these observations, we claim the following to be the key tenets towards raising business application product lines:

T1 - Generative development whereby application specification is kept independent of technology platform concerns thus enabling application developers to focus solely on specifying functionality in a manner that is intuitive and closer to the problem domain.

T2 - Generation of code generators whereby technical architects can specify the desired [set of] code generator[s] as a hierarchical composition wherefrom code generator generator delivers implementation of the desired [set of] code generator[s]. Moreover, elements of the hierarchical composition can be seen as reusable artefacts.

T3 – The key issue in supporting product lines is to specify *what* can change *where* and *when*. This enables specification of a product line as a cross-product of a set non-changing artefacts and a set of variation artefacts for each placeholder in a non-changing artifact. A resolution mechanism is required to project out meaningful members of the cross-product such that each member is a valid product. This calls for an abstraction that can address composition, variability and resolution in a unified manner. Since business applications are typically implemented conforming to a layered architecture wherein an architectural layer encapsulates a specific [set of] concern[s],

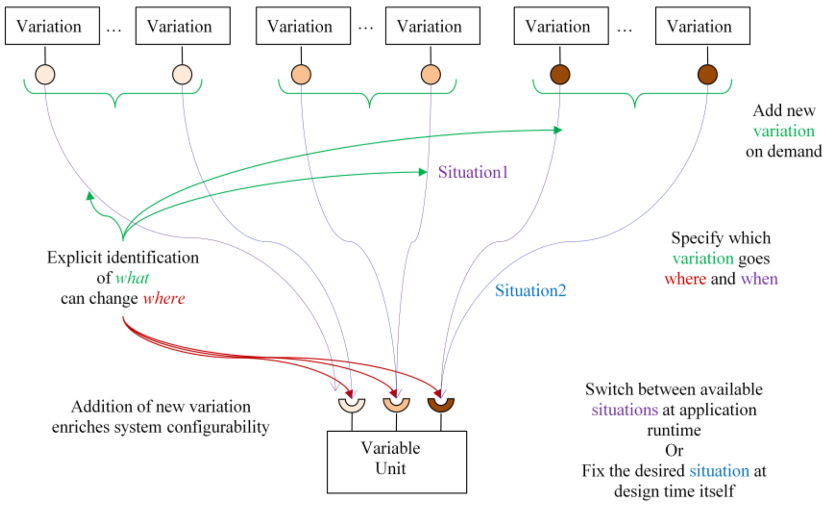

Variation
...
Variation
Variation
...
Variation
Variation
...
Variation
Add new variation on demand
Situation1
Explicit identification of what can change where
Specify which variation goes where and when
Situation2
Addition of new variation enriches system configurability
Switch between available situations at application runtime Or Fix the desired situation at design time itself
Variable Unit

Fig. 1. Architecting for configuration and extension

the abstraction should be amenable for use within and across the architectural layers. Moreover, this abstraction may need to be used at different levels of granularity, for instance, at Class level, Module level, and Application level. Also, there is a temporal dimension of variability that needs to be addressed. For instance, is the variability fixed/bound /determined at application design time or installation time or run-time?

A model-driven development approach can help in separation of technology concerns from functionality, and automatic derivation of suitable implementation therefrom thus addressing tenet T1[19]. Applying model-driven techniques to specify model-based code generators as a hierarchical composition of parameterized model-to-text transformation templates makes it possible to generate the desired implementation therefrom thus addressing T2[20]. This further enables a set of cohesive model-based code generators to be visualized as a product line. In this paper, we present: i) the core abstraction required for addressing tenet T3, ii) its realization in terms of a set of meta models, iii) an architecture to support composition and variability management of business applications, and iv) a method for managed evolution of the product line. Rest of the paper is organized as follows – Section 2 describes the core solution addressing tenet T3. Section 3 illustrates the proposed solution through a pragmatic example. Related work is described in section 4. Early experience and unaddressed challenges are highlighted in Section 5 with section 6 concluding.

2 Proposed Solution

2.1 Architecting Business Applications for Configuration and Extension

Idea is to visualize the system under consideration as a set of composable Variable Units each having a set of well-defined Variation Points (VPs) as shown in Fig. 1.

The variation points of a variable unit denote the places *where* changes are expected to occur thus reflecting current level of understanding of the domain. A Variation (V) denotes *what* can change at a variation point so as to cater to a specific *Situation*. A *situation* helps to describe the context, i.e. *when* a specific change can occur. Addition of a new variation enriches *system configurability* i.e. ability to address more situations. Also, the variation being added can have variation points of its own thus introducing new paths for extension and configuration. Thus, the system begins to take the shape of a product line wherein a member corresponds to a set of variable units and variations such that all variation points are bound to variations, and the variations are *consistent* among themselves. Some situations may require application designer to take a relook at a variable unit whereby new variation points can be introduced or old variation points discarded.

2.2 A Model-Driven Architecture for Managing Variability and Configuration

Figure 2 depicts a model-driven architecture for managing variability and configuration. Key components of the architecture are: i) a generic Base Metamodel (BM) that enables specification of any domain model, in our case, model of business application pertaining to any business vertical such as Banking, Financial Services, Insurance, etc. ii) a generic Variability Metamodel that helps to specify a priori known variability in domain-independent manner, iii) a Target Metamodel (TM) that enables specification of the resolved domain model i.e. situation-specific model, and iv) a configuration engine that resolves variability and delivers the base model with each variation point plugged with the desired variation.

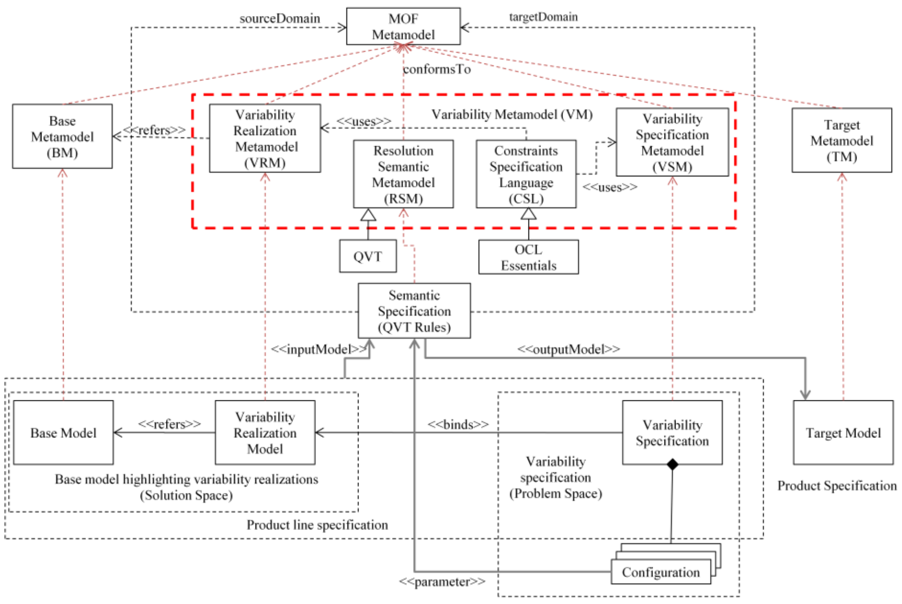

sourceDomain
MOF Metamodel
targetDomain
conformsTo
Base Metamodel (BM)
<<refers>>
Variability Realization Metamodel (VRM)
<<uses>>
Variability Metamodel (VM)
Variability Specification Metamodel (VSM)
Target Metamodel (TM)
Resolution Semantic Metamodel (RSM)
Constraints Specification Language (CSL)
<<uses>>
QVT
OCL Essentials
Semantic Specification (QVT Rules)
<<inputModel>>
<<outputModel>>
Base Model
<<refers>>
Variability Realization Model
<<binds>>
Variability Specification
Target Model
Base model highlighting variability realizations (Solution Space)
Variability specification (Problem Space)
Product Specification
Product line specification
Configuration
<<parameter>>

Fig. 2. Overview of Variability Modeling and Configuration Approach

Variability Metamodel (VM) describes variability of a domain model (base model) using two metamodels namely, variability realization metamodel (VRM), and variability specification metamodel (VSM). VRM describes the concrete variability of a variable unit in terms of Variation Points (VP) and Variations (V). VSM describes variability in abstract form such as features [10, 17] and their consistent configurations each describing an a priori known situation. We also provide for specification of semantics of a variation point, i.e. how a variation point is to be interpreted for a given variation. Resolution Semantic Metamodel (RSM) describes such semantic interpretations. In addition, a declarative constraint specification language is used for specifying constraints on variability model.

In our realization of business application product line, BM and TM are MOF [28] describable metamodels and VM is aligned with MOF standards. For instance, we use operational Query View Transformations (QVT) [30] as a RSM and OCL [27] as constraint specification language. This unification, i.e. all metamodels are MOF-describable, helps in specifying relationships between different meta models, for instance, between VRM and BM, between VRM and VSM etc. The steps for modeling variability and a process for resolving variability are summarized below:

A) The concrete variability specification begins with highlighting or annotating the base model. This results in instantiating the VRM with appropriate references (i.e., `<<refers>>`), to the base model. This separates out the base model and variability realization model.
B) The abstract variability specification begins with the variability specification model (similar to feature tree specification). Steps A and B may be carried out in parallel, and once completed, appropriate bindings (i.e., `<<binds>>`) are provided from the realization model to the specification model. In addition, a set of valid configurations can be specified on variability specification model.
C) Once steps A and B are complete (along with bindings, configurations, and reference), the configuration or materialization process can begin. In this step, the semantic specification (QVT rules) is defined using the product line specification model[1] and a valid configuration (shown as parameter in Fig 2) as input. The semantic transformation rules (QVT rules) generate the target resolved model as output.

2.3 A Generic Variability Metamodel

Fig 3 depicts the variability metamodel whose key elements are described below:

Variation Point: A Variation Point (VP) is a placeholder in the VRM where variations can be plugged in. A VP is derived from the variability class reference (VClassRef), which is an instance of the MOF class. Also, VPs refers to base model elements via a reference handler. It is assumed that any base model element is an instance of the MOF class. A VP must have a variation point type (VPType) that captures the behavior of the variation point. In other words, VPType determines how the variation point will be

[1] Dotted line in Figure 2 showing VRM, BM, VSM along with corresponding bindings and references.

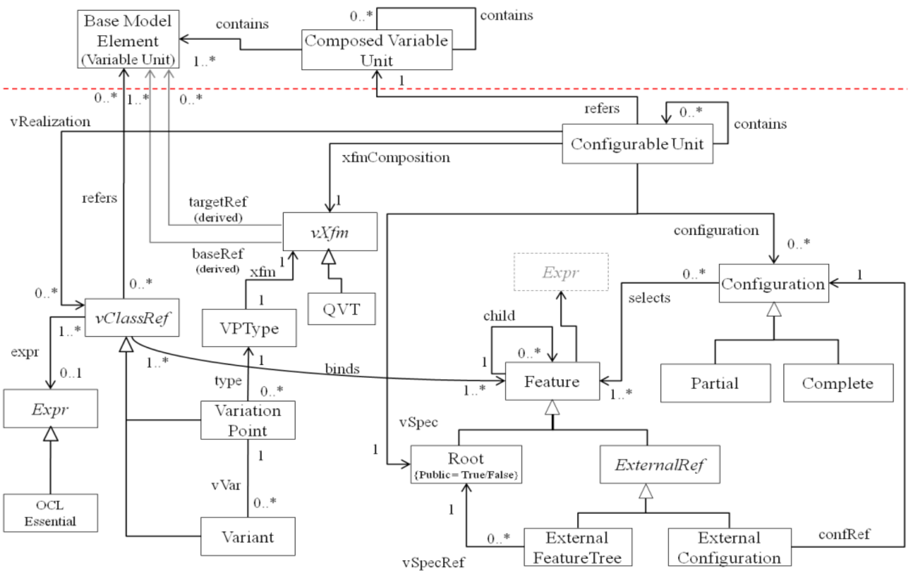


Fig. 3. Variability Metamodel

handled by resolution semantics. The metamodel does not make explicit definition of VPType, instead the semantics is specified using QVT transformation rules.

Variation: Variations can be considered as individual parts that can be plugged into a variation point (with type safety). Variations are the second key component of VRM. Similar to VPs, variations are also derived from VClassRef and conform to MOF class. Constraint expressions on variation points and variations can be defined using OCL. Similar to VP, variations also refer to base model elements via a reference handler.

vXfm: Variability transformation or vXfm signifies transformation applied on a variability class reference (i.e., variations points and variations). They capture the resolution semantics of VM and are expressed in QVT. The QVT rules are used to resolve a target model from unresolved product line input specification.

Feature: A primary constituent of the VSM is a feature or vSpec tree. The top of the tree is denoted by a Root that facilitates in the composition of the tree. A feature tree can be composed of external references, i.e., external feature tree or external configurations (i.e., pre-configured). A feature is an abstract representation and is realized via bindings to concrete concepts like variation points and variations.

Configuration and Resolution: A variability configuration is a set of all valid resolutions from a variability specification tree (i.e., feature tree) whereas variability resolution is the process of resolving a single feature (VP) to a distinct choice (variation) from a set of possible choices (variations). A configuration can be either partial (unresolved resolutions) or complete when all resolutions are resolved.

Configurable Unit: A configurable unit is a reusable entity that can be composed of other configurable units. It refers to a composite Variable Unit (of Base Model) via a reference handler. A CU can be either preconfigured when it contains valid configurations (i.e. a CU without any feature tree) or a CU can be partially configured/ unconfigured when it contains a set of valid configurations and a feature tree. A CU also guides in the composition of vXfms (resolution semantics). This is shown in Fig 3 by the xfmComposition association.

2.4 Variability and Multi-dimensional Separation of Concerns

We make use of modeling and model-based code generation techniques to enable multi-dimensional separation of concerns to the extent possible. The use of high level specification languages enables separation of functional concerns (i.e. Business functionality and Business process dimensions) from technical concerns (i.e. Solution architecture dimension). Moreover, separating business functionality concern from business process concern enables multiple orchestrations of a given set of application services.

Business Functionality: Business functionality is implemented in a layered architecture comprising of three architectural layers – user interface, application functionality and database. A user interacts with an application through its user interface. The user feeds in information through *forms* to be processed either for *querying* and / or *computations* and / or *updates* to the application state, and browses over the information returned. Forms are implemented in the desired graphical user interface platform such as Struts-JSP, Flex-PHP, Winforms-ASP etc using standard graphical user interface primitives such as windows, controls, buttons, event handling logic etc. Window is a unit of interaction between the user and the application, and is composed of controls and buttons. A control accepts or presents data in a specific format. The user can perform a specific task by clicking on a button. A task identifies an application service, i.e. query or computation or update to application state, being exposed by Application functionality layer. The functionality is modeled using classes, attributes, methods and associations between classes. Class is set of attributes and methods implementing cohesive business functionality. Some methods are application services. Some classes need to be persisted using, typically, RDBMSs. The database layer provides persistence for application objects using RDBMS tables, primary key and query based access to these tables, and an object oriented view of these accesses to the application layer.

Fig. 4 shows unified meta model, architectural layer specific meta models as views of the unified meta model, and associations spanning across the architectural layer models. The architectural layer specific metamodels are the *Base Models*, and all elements of these metamodels such as Class, Property, Key, and Button are the *Base*

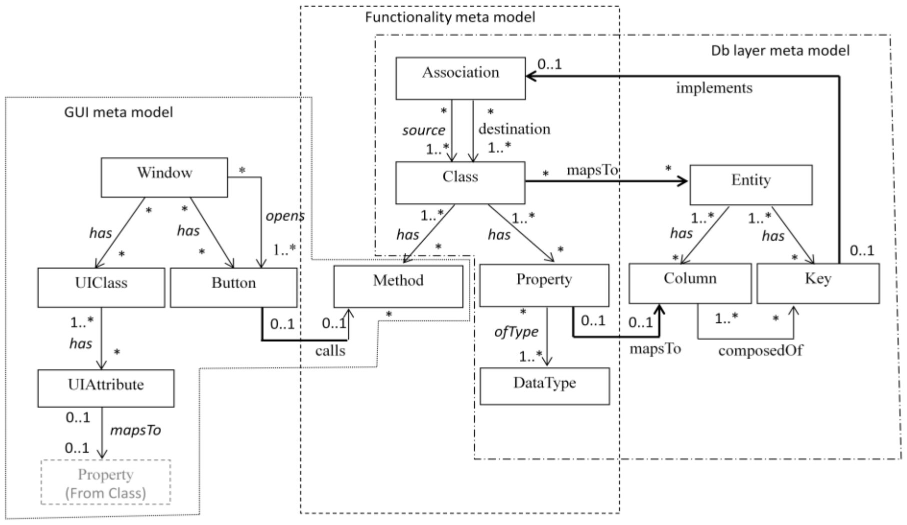


Fig. 4. Unified meta model for business functionality

Model Element of variability metamodel described in Fig. 3. By establishing references (i.e., <<refers>>) to the desired base model elements (i.e. instance of class, property, etc) variation points (VP) and variations (V) can be made explicit. Thus, each architectural layer specification is extended to model a priori known variability. Additionally, architectural layer specific feature model specifies valid combinations of variations possible within the architectural layer. Thus, we allow modeling of a set of well-formed variants for each architectural layer. Since business functionality is composition of various architectural layers, variations in architectural layers lead to variations in business functionality. Business functionality level feature model specifies valid combinations of architectural layer variants. This composition is achieved through the composable configurable units (refer Fig.3 and Metamodel description).

Business Process: Business process dimension comprises of operational tasks, organizational structure, and organizational policies concerns. At present, organizational policies are not modeled explicitly but get encoded through specification of the other two concerns. Organizational structure concerns are typically externalized through workflow specification. Industry practice is to use BPMN 2.0 [29] and EPC [33] to model business processes. We have come up with a MOF describable business process metamodel that conforms to BPMN 2.0 standard. We enable variability in business processes to be made explicit through *Activity*, *Event* and *Gateway* model elements as *Base Model Elements*. Details can be found in [4, 21].

Solution Architecture: Solution architecture dimension comprises of Design decisions (D), Architecture (A) and Technology platform (T) concerns. Solution architect is faced with multiple choices along these dimensions in order to arrive at the most appropriate solution architecture for the specific situation. These choices get

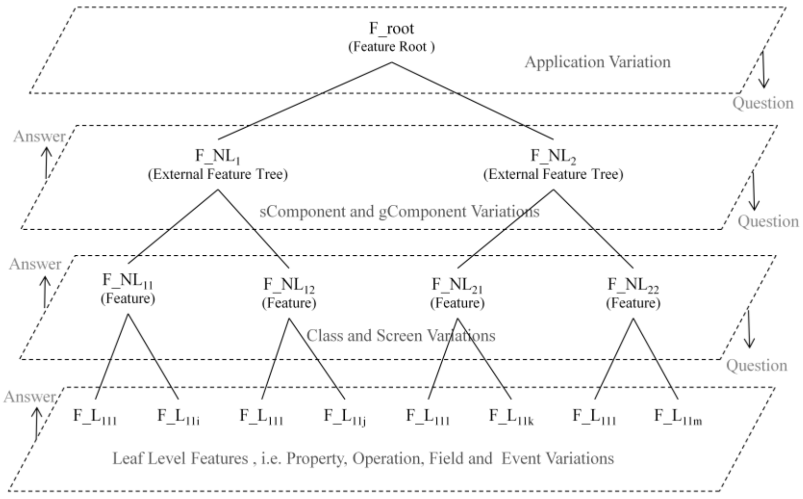


Fig. 5. Variability resolution across multiple level of granularity

encoded in the implementation of the code generators [19, 20]. As a result, retargetting the same functional specification into different solution architecture entails implementation of a fresh set of code generators. Instead, we devised a way of model-based generation of model-based code generators [3, 20]. As a result, known solution architecture strategies are externalized into a repository of *building blocks*. A building block encapsulates contribution of a given choice along D / A / T dimension to the eventual implementation i.e. solution architecture [20]. Thus, supporting new solution architecture is either novel composition of existing building blocks or addition of a new building block. We unify variability description and resolution by using VSM and VRM (described in section 3 B) with building block metamodel described in [3 and 20].

2.5 Variability and Multiple Levels of Granularity

As discussed earlier, variability in business applications needs to be managed along multiple dimensions where each dimension comprises of multiple concerns or sub-dimensions thus leading to hierarchical decomposition structure. We showed how to enhance a concern-specific metamodel so as to make concern-specific variability explicit and to specify valid variations for a concern-specific model. Hierarchical decomposition structure demands variability resolution mechanism at n+1th level such that it specifies valid compositions of variants at nth level. The *configurable unit* of metamodel depicted in Fig 3 addresses this need with *Expr* being the expression language for describing influence of selecting choice of a concern onto other concerns.

For example, business applications demonstrate variability at different levels of granularity as shown in Fig. 5. Variations in Operations and Properties lead to variations in Class. Variations in Fields and Event code lead to variations in Screen. Variations in Class lead to variations in sComponent (i.e. server functionality). Variations in Screens lead to variations in gComponent (i.e. client functionality). Variations in sComponents and gComponents lead to variations in Application. The features at level n+1 are answers to the questions their parent features at level n correspond to. Dependence between features helps impart further well-formedness to the variations across multiple levels of granularity. Input to the configuration engine is a transform of the feature diagram shown in Fig. 5 to a series of multiple choice questions.

2.6 Variability Resolution at Various Phases of Application Development

The proposed approach helps visualize a business application product line wherein application is specified in terms of a set of MOF describable base metamodels, and a code generators product line wherein code generator is specified in terms of building block metamodel. The proposed generic variability metamodel is capable of specifying variability along multiple dimensions of separation of concerns namely, business functionality and solution architecture. A purpose-specific business application can be derived through application of two operators namely, resolution and code generation, to the product line specification. Order of application of the two operators results in three ways of variability resolution namely, at application design-time, at application installation time, and at application run time.

Design-time resolution involves resolution of variability in business functionality first followed by the resolution of variability in solution architecture next. The latter leads to code generator specifications for the given situation. The resolved business functionality specifications are then transformed using the desired code generators to deliver the desired business application implementation. Installation-time resolution involves resolution of solution architecture variability first. The resultant code generator specifications are used to transform business functionality product line specifications into implementation and metadata encoding the unresolved variability. Application installer makes use of the metadata to resolve functional variability at installation time. Run-time resolution differs from Installation-time resolution in that the metadata is used for resolving application variability at application run-time. Run-time resolution demands multi-tenant architecture and relies on metadata interpretation.

3 Illustrative Example

We illustrate the proposed approach with a very small subset of functionality from banking domain. UML class model is the base model used. All as-yet-known variations are modeled. We show how the variability is resolved to derive a situation-specific UML class model. Only two classes namely, Customer and Address, are modeled due to space constraint.

The customer class has properties customer name, uid (universal identification no.) and dob (date of birth), address and an operation called getCustomerCreditHistory. Typically, the customer identification number and the address differ with operational context. For example, a customer located in US is identified by a 10 digit numeric SSN

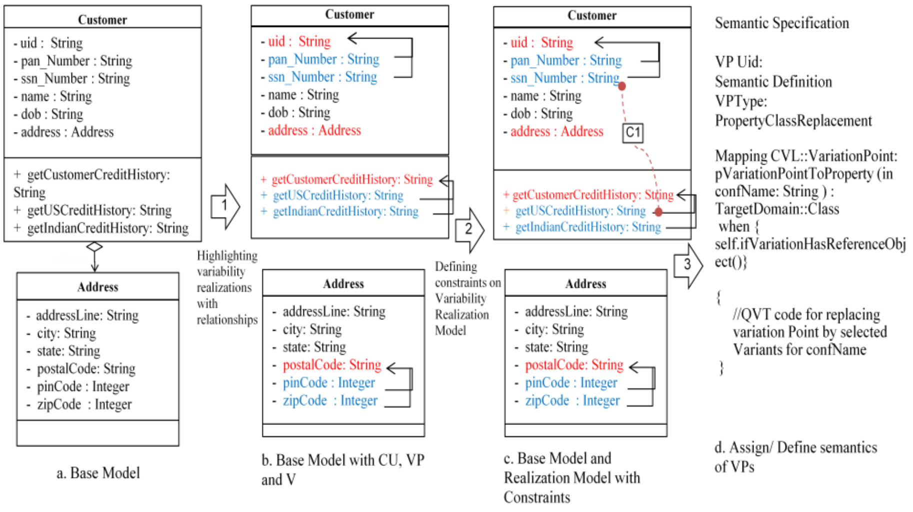

Customer
- uid : String
- pan_Number : String
- ssn_Number : String
- name : String
- dob : String
- address : Address
+ getCustomerCreditHistory: String
+ getUSCreditHistory: String
+ getIndianCreditHistory: String
Address
- addressLine: String
- city: String
- state: String
- postalCode: String
- pinCode : Integer
- zipCode : Integer
a. Base Model
1
Highlighting variability realizations with relationships
b. Base Model with CU, VP and V
2
Defining constraints on Variability Realization Model
C1
c. Base Model and Realization Model with Constraints
3
Semantic Specification
VP Uid:
Semantic Definition
VPType:
PropertyClassReplacement
Mapping CVL::VariationPoint:
pVariationPointToProperty (in confName: String) :
TargetDomain::Class
when {
self.ifVariationHasReferenceObject()}
{
//QVT code for replacing variation Point by selected Variants for confName
}
d. Assign/ Define semantics of VPs

Fig. 6. Defining Variability Realization Model (VRM)

(social security number) and an address field is described by a ZIP code, whereas a customer in India is identified by a PAN number (Personal Account Number) and an address represented by a PIN code. The operation getCustomerCreditHistory of a customer can also differ in different contexts – US-based banks use Credit Bureau Report to determine credit history whereas Indian Banks use CIBIL agency. Thus the following variability requirements emerge:

1. **For US Customer**: unique identification number is SSN based, address is ZIP code based, and getCustomerCreditHistory is based on Credit Bureau Report.
2. **For Indian Customer:** PAN based unique identification number, PIN code based address, and getCustomerCreditHistory is based on CIBIL.

Fig 6.a depicts Customer and Address class model for the banking product line. The class model, i.e. base model, captures the common as well as the entire variability requirements as an instance of UML metamodel. For example, the common properties of a Customer class are name and dob, while the variable properties are uid, pan number and ssn. Operation getCustomerCreditHistory is the only variable operation of Customer class, and it maps onto two variations, getUSCreditHistory and getIndianCreditHistory. The address property of Customer class refers to the Address class that has its own variability requirements as shown in Fig 6.a.

The process of realizing variability from a given unresolved base model, as described in section 3.B, is illustrated in Figures 6-9. It begins with highlighting or marking the variation points, variations and the relationship between variation points and variations in the unresolved base model. Fig 6.b shows: i) the variation points of Customer class (i.e. uid, address, and getCreditHistory) and Address class (i.e. postalCode) highlighted in red colour, ii) the variations in Customer class (i.e. pan_Number, ssn_Number, getUSCreditHistory, getIndianCreditHistory) and Address (i.e. pinCode,

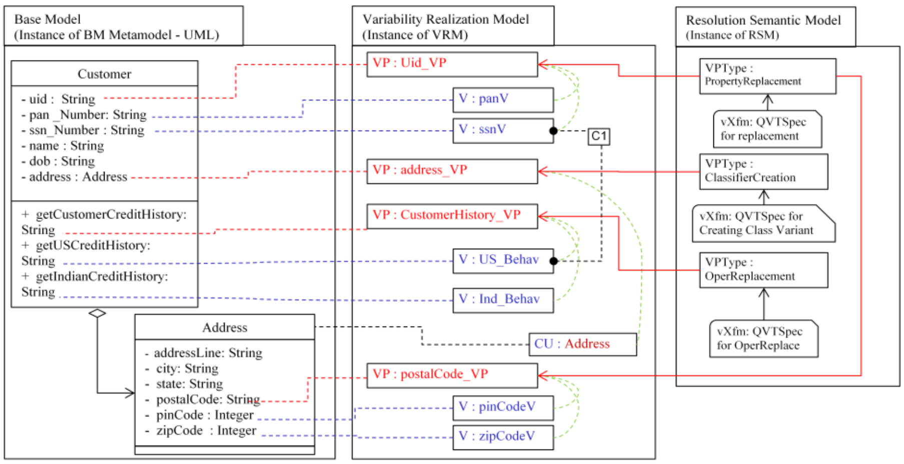


Fig. 7. Underlying variability realization model and resolution semantic model

zipCode) in blue colour, and iii) relationships between variation points and variations shown by special multi-tail arrows where the head points to VP and the tail points to variations. Fig 6.c depicts constraint C1 which specifies choice of variation ssn_Number leads to the choice of variation getUSCreditHistory. Fig 6.d shows specification of semantic interpretation for the variation point using QVT.

Fig 7 shows the variability realization model along with its appropriate references to the base model. As stated earlier, VRM is independent of the base (meta-) model and refers to the base model elements via reference handlers (dotted red and blue lines). C1 is a constraint defined in the realization model. In addition, Fig 7 also shows the semantics model that defines how the VPs would be handled by corresponding variation point types (VPTypes) and QVT rules. The model depicted in Fig 7 describes the model of solution space of the variability requirements for the banking product.

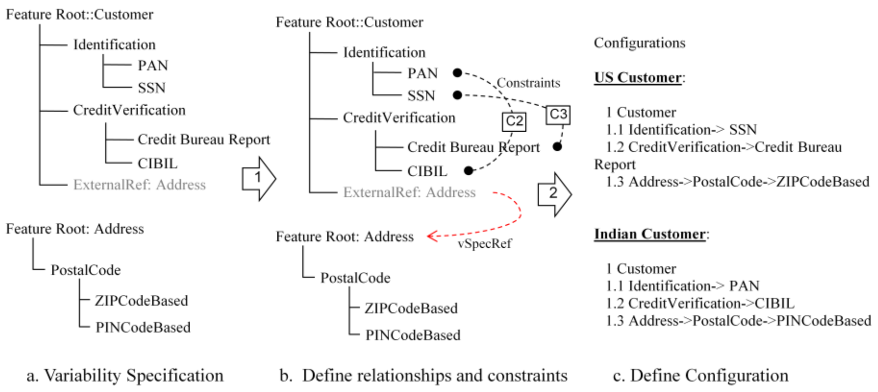


Fig. 8. Defining Variability Specification (VSM)

The process of defining variability specification, (see Fig 8) starts with identifying configurable units or CUs. In our example the two CUs are the Customer CU and the Address CU (Fig 8.a). Note that Customer CU contains Address CU via the external reference as shown in Fig 8.a. Fig 8.b describes the complete variability specification for the Customer feature along with various constraints. Configuration criteria for specifying a US Customer or an Indian Customer are depicted in Fig 8.c. Once the variability specification model is defined, bindings from the abstract specification model to concrete realization model must be accomplished. The binding process is illustrated in Fig 9 that shows how VPs and variations from the realization model are bound to various choices or features in the feature tree. Once all the above steps are completed, the configuration process can derive a purpose specific base model by applying appropriate M2M transformations on the input product line specification. This completes demonstration of: a) how variability and consistent choices (configuration) can be specified in an intuitive manner, b) how variability definitions of variable units can be composed to derive a larger unit using configurable unit, and c) how semantics of variability resolution can be specified using model transformation specification language.

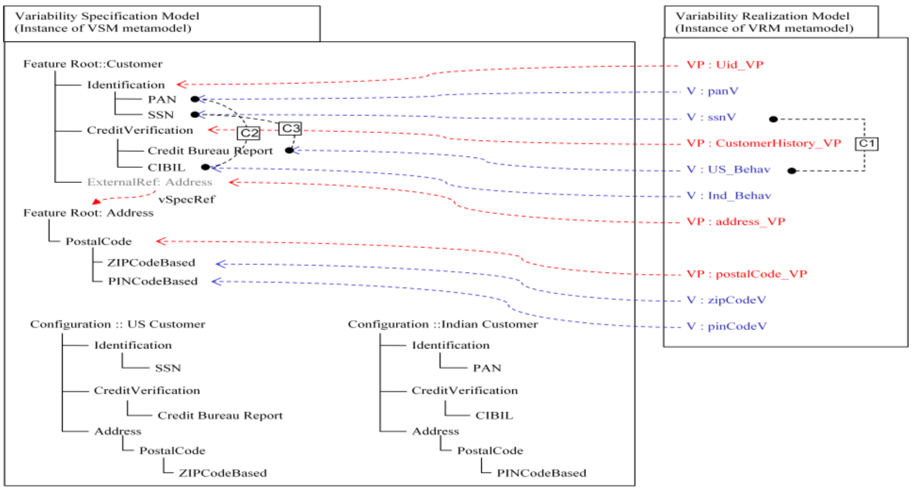


Fig. 9. Complete Variability Model (VM)

4 Related Work

There are several approaches addressing variability management of software product lines with varying degree of success. They can be classified into two broad categories – code centric and model-driven. Code-centric approaches visualize product line as a set of common and situation-specific variable code fragments such that situation-specific implementation can be derived by composing the valid set of variation fragments with the common code fragments. This is aligned with the intuition described in sub-section 3.A. Early techniques used #ifdef directive to make variations explicit and relied on pre-processors for composition. Though simple to specify and implement, these techniques could not even guarantee that resultant composed code will compile. Also, use of pre-processors eliminated resolution of variability at runtime. Advanced modularization and composition techniques such as Aspect/J [18], Hyper/J [32], AHEAD [5], Mixin [7], Jiazzi [24], Scala[26], ClassBoxes [1], Composition Filters [14], Caesar [25], and Framed Aspects [23] fare better in comparison though with some limitations still. For example, Aspect/J can compose cross-cutting concerns at design time and runtime as long as the composition lies within the fixed join-point model supported. On the other hand, ClassBoxes provide support for defining new composition semantics but do not support composition of class fields/attributes. Apart from these specific limitations, being tied to a specific programming language severely hampers efficacy, even applicability, of code centric approaches for addressing variability along multiple dimensions of separation of concerns. This is so because one programming language might not be the best suited for specifying all dimensions of concerns.

Though around for a while, model-centric approach is not common for managing software product lines. Most common use is of feature models for describing (rather, documenting) product lines, typically from end-user perspective [5, 10, 17]. However, traceability from features to product line implementation is missing which leads to inadequate support for variability resolution. In congruence with our approach, other model-centric approaches to product lines exist [2, 15, 16]. They also use a set of meta models to specify the common part, the variations, and the concrete as well as abstract variability specification. Like our approach, they too depend on model transformation techniques to handle resolution requirements. However, our approach differs with them in three key aspects.

1. Describing concrete variability specification: Most of the existing approaches use Base Model extension mechanism (using stereotype) to describe concrete variability. For instance, a proposal for modeling variability in software families with UML using the standardized extension-mechanisms of UML (using stereotype) is presented in [8]. On the similar line, the extension of base metamodel using UML stereotype is presented in [15, 16, 22]. Whereas, we use a generic variability metamodel with MOF meta-meta level unification to establish interoperability with any MOF describable metamodel. This approach also conforms to the OMG's RfP for Common Variability Language [31] and [9].
2. Describing abstract variability specification: In congruence with other approaches, our metamodel is based on existing feature modeling technique. Our VSM is aligned with a specification described in [10]. We unify the key concepts of variability specification metamodel with variability realization metamodel to establish bindings between them, which is similar to a concise representation of variability specification for different kinds of models as presented in [12]. However applying variability at various levels of granularity and along different dimensions are unique to our proposed approach.
3. Resolution semantics and approach: Existing resolution approaches [13, 16] are based on model transformation techniques but they are based on pre-defined M2M transformation rules. Instead, our approach uses the concept of transformation based semantic composition. This enables customized semantics for each variation point to be composed by any M2M transformation language like QVT.

5 Early Experience and Evaluation

We are in the final step of realizing our objective. The core infrastructure namely, meta models, model processors, method etc are in place. The central idea is vetted by raising model-based code generators product line [20]. A near real-life example in Banking domain is also implemented to illustrate all the concepts in a laboratory setting. We are about to start with a real-life product-line implementation through restructuring and refactoring of a set of existing purpose-specific solutions. Completion of this exercise will give better feedback on robustness and usability of the proposed approach.

Still, early experience tells that models, through better separation of multi-dimensional concerns, clearly provide a better handle than code for implementing business application product lines. Separation of solution architecture from business functionality concerns enables business domain experts to focus solely on specifying the variations in business functionality and technology architects to focus solely on specifying the variations in technology platform, design strategies and architecture. Independent resolution of variability along multiple dimensions and model-based generation of model-based generators enable application variability resolution at design-time or installation-time or run-time. The Variation Points also double up as extension points for introduction of as-yet-unforeseen changes. And unforeseeable changes are a reality for business applications.

Though early signs are encouraging, several significant issues remain to be addressed:

- Multi-level resolution seems to suffice but is posing usability challenge even for the small application we implemented. In the least, more intuitive GUI seems a must for resolving variability.
- Business-critical applications need to evolve with time through extension and mutation. The proposed variability model is adequate to address extension i.e. *add as-yet-unseen variant part* or *add as-yet-uncalled-for common part*. But, mutation would need refactoring of common part to add a new variation point, and commensurate fusion of a set of variant parts etc. Intuitive refactoring support is essential.
- Maintenance / evolution effort far exceeds the development effort for a successful business application [6]. Precise computation of impact of a change and optimal testing (i.e. *what* to test *when*) is a must.
- Effective management of a product line demands coordination of multiple stakeholders such as Domain experts, Solution architects, Developers, Testers, Product line managers etc across the various SDLC phases. Should there be a feature model for every stakeholder? But a stakeholder might be interested in a set of [sub] dimensions leading to overlap of feature models and feature dependency. It calls for a method (and the relevant tooling) to help *what* to do *when* and by *whom*. There is a need to build further on the proposed multi-level resolution model and [11].
- Definition of a new mutual fund offering or an insurance policy or a financial product varies from the existing ones in a well-defined manner even though the variations need to be introduced at many places. A declarative mechanism aided by suitable [de]composition architecture is missing.

6 Conclusion

To raise business application product lines, we argued, the need for: i) use of models and model-driven techniques to implement SPLE concepts, ii) an abstraction that addresses composition, variability and resolution in a unified manner, iii) a mechanism to resolve variability at several levels of granularity, and iv) abstraction and method support to address unforeseeable changes. We presented a model-driven solution that addresses these needs and shared early experience with the solution in practice. We also outlined some of the significant issues that need to be overcome for ready adoption of the proposed solution by industry practice.

References

1. Bergel, A., Ducasse, S., Wuyts, R.: Classboxes: A Minimal Module Model Supporting Local Rebinding. In: Böszörményi, L., Schojer, P. (eds.) JMLC 2003. LNCS, vol. 2789, pp. 122–131. Springer, Heidelberg (2003)
2. Bachmann, F., Goedicke, M., Leite, J., Nord, R., Pohl, K., Ramesh, B., Vilbig, A.: A Meta-model for Representing Variability in Product Family Development. In: van der Linden, F.J. (ed.) PFE 2003. LNCS, vol. 3014, pp. 66–80. Springer, Heidelberg (2004)
3. Barat, S., Kulkarni, V.: Developing configurable extensible code generators for model-driven development approach. In: SEKE 2010, pp. 577–582 (2010)
4. Barat, S., Kulkarni, V.: A Component Abstraction for Business Processes. In: Business Process Management Workshops, vol. (2), pp. 301–313 (2011)
5. Batory, D.: Feature-oriented programming and the ahead tool suite. In: ICSE 2004: Proceedings of the 26th International Conference on Software Engineering, pp. 702–703. IEEE Computer Society, Washington, DC (2004) ISBN 0-7695-2163-0
6. Boehm, B.: A Spiral Model of Software Development and Enhancement. ACM SIGSOFT Software Engineering Notes, "ACM" 11(4), 14–24 (1986)
7. Bracha, G., Cook, W.: Mixin-based inheritance. In: OOPSLA (1990)
8. Clauß, M., Jena, I.: Modeling variability with UML. In: GCSE 2001Young Researchers Workshop (2001)
9. Common Variability Language Initiative, `http://www.omgwiki.org/variability/doku.php`
10. Czarnecki, K., Eisenecker, U.: Generative programming methods, tools and applications. Addison-Wesley (2000)
11. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged Configuration Using Feature Models. In: Nord, R.L. (ed.) SPLC 2004. LNCS, vol. 3154, pp. 266–283. Springer, Heidelberg (2004)
12. Czarnecki, K., Antkiewicz, M.: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 422–437. Springer, Heidelberg (2005)
13. Deelstra, S., Sinnema, M., Jilles, V.G., Bosch, J.: Product derivation in software product families: a case study. Journal of Systems and Software 74(2), 173–194 (2005)
14. Filman, R.E., Elrad, T., Clarke, S., Aksit, M.: Aspect-Oriented Software Development. Addison-Wesley (2004)
15. Gomaa, H., Webber, D.L.: Modeling Adaptive and Evolvable Software Product Lines Using the Variation Point Model. In: 37th Annual Hawaii International Conference on System Sciences (HICSS 2004), vol. 9, p. 90268.3 (2004)

16. Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Solberg, A.: An MDA®-based framework for model-driven product derivation. In: IASTED Conf. on Software Engineering and Applications 2004, pp. 709–714 (2004)
17. Kang, K., et al.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. CMU/SEI-90-TR-21. Carnegie Mellon Univ., Pittsburgh, PA (November 1990)
18. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Lee, S.H. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–354. Springer, Heidelberg (2001)
19. Kulkarni, V., Reddy, S.: Model-Driven Development of Enterprise Applications. In: UML Satellite Activities 2004, pp. 118–128 (2004)
20. Kulkarni, V., Reddy, S.: An abstraction for reusable MDD components: model-based generation of model-based code generators. In: GPCE 2008, pp. 181–184 (2008)
21. Kulkarni, V., Barat, S.: Business Process Families Using Model-driven Techniques. In: 1st International Workshop on Reuse in Business Process Management, rBPM 2010 (2010)
22. Kulkarni, V.: Raising family is a good practice. In: FOSD 2010, pp. 72–79 (2010)
23. Loughran, N., Rashid, A., Zhang, W., Jarzabek, S.: Supporting Product Line Evolution with Framed Aspects. In: ACP4IS Workshop, AOSD (2004)
24. McDirmid, S., Flatt, M., Hsieh, W.C.: Jiazzi: New age components for old-fashioned Java. In: OOPSLA (2001)
25. Mezini, M., Ostermann, K.: Conquering Aspects with Caesar. In: AOSD (2003)
26. Odersky, M., et al.: An Overview of the Scala Programming Language (September 2004), http://scala.epfl.ch
27. OMG Document (OMG document number - formal/2010-02-01): Object Constraint Language (OCL), Version 2.2, http://www.omg.org/spec/OCL/2.2/
28. OMG Document (OMG document number formal/2006-01-01): Meta Object Fa-cility (MOF) - Version 2.0, http://www.omg.org/spec/MOF/2.0/
29. OMG. BPMN 2.0, OMG document - dtc/10-06-04 (2010), http://www.bpmn.org
30. OMG Document (OMG document number - formal/2011-01-01): Meta Object Facility (MOF) 2.0 Query/View/Transformation, v1.1, http://www.omg.org/spec/QVT/1.1/
31. OMG RFP – Common Variability Language (CVL) RFP, http://www.omg.org/techprocess/meetings/schedule/Common_Variability_Language_%28CVL%29_RFP.html
32. Tarr, P.L., Ossher, H., Sutton Jr., S.M.: Hyper/J: multi-dimensional separation of concerns for Java. In: ICSE 2002, pp. 689–690 (2002)
33. Van der Aalst, W.M.P.: Formalization and verification of event-driven process chains. Information & Software Technology 41(10), 639–650 (1999)

Inter-association Constraints in UML2: Comparative Analysis, Usage Recommendations, and Modeling Guidelines

Azzam Maraee[1,2] and Mira Balaban[2,⋆]

[1] Deutsche Telekom Laboratories, Ben-Gurion University of the Negev, ISRAEL
[2] Computer Science Department, Ben-Gurion University of the Negev, ISRAEL
{mari,mira}@cs.bgu.ac.il

Abstract. UML specification is verbal and imprecise, the exact meaning of many class diagram constructs and their interaction is still obscure. There are major problems with the inter-association constraints subsets, union, redefinition, association specialization, association-class specialization. Although their standard semantics is ambiguous and their interaction unclear, the UML meta-model intensively uses these constraints.

The paper investigates the semantic implications of the above inter-association constraints, their interaction with other constraints, and implied correctness problems. Based on this study, we present a comparative analysis of these constraints, that includes characterization, and refers to complexity factors, and usefulness aspect. This analysis yields recommendations concerning the semantics and usefulness of the constraints. In addition, we present modeling guidelines for users. To the best of our knowledge, this is the first all inclusive analysis of the inter-association constraints in UML2.

1 Introduction

UML 2 [1] introduces several new constraints, *property subsetting, property union and property redefinition* for inter-property relationships, and widely uses them in the definition of its meta-model. The new constraints add up to the previous constraints *association specialization and association-class hierarchies*, to form the set of inter-association constraints. These constraints are important due to their role in the meta-model, in defining new modeling languages [2] and in real domains such as configuration management.

Inter-association constraints present a computational complexity problem. Artale et al. [3] shows that reasoning about consistency of class diagrams with inter-association constraints is an *ExpTime* problem. Dropping the *complete*, association-class hierarchy and association specialization constraints decreases the complexity to *NP*.

⋆ Supported in part by the Paul Ivanir Center for Robotics and Production Management at Ben- Gurion University of the Negev.

The semantics of inter-association constraints as described in the meta-model is ambiguous, their inter-relationships are obscure, and some constraints have contradictory interpretations. The subsetting, redefinition, association specialization and union have been studied in [2, 4–11]. These works try to settle various semantic issues, but did not reach a semantic agreement. An overall consideration of all constraints, including association-class hierarchy, is still missing.

This paper presents an *observable-based* approach for investigating the inter-association constraints in UML2. The observables are properties of the constraints, for which there is a general agreement among the researchers. We study possible interpretations of the inter-association constraints, following three criteria: (1) preservation of observables; (2) minimizing contradiction with the meta-model; (3) consistency with the selected semantics of other UML constraints. In addition, the paper investigates the semantic implications of the inter-association constraints, their interaction with other constraints, and implied correctness problems. The results of the investigation have been added to our ongoing anti-pattern catalog of class diagram correctness patterns [12], and for extending the ***FiniteSat*** [13] tool to apply to these constraints.

Based on this study, we present a comparative analysis of these constraints, referring to complexity factors, and usefulness aspect. This analysis yields recommendations for standard semantics, and modeling guidelines. To the best of our knowledge, this is the first all inclusive analysis of the inter-association constraints in UML2.

Section 2 formally defines the UML class diagram model. Sections 3-5 study the inter-association constraints, subsetting, redefinition, association specialisation and association class hierarchy respectively; Section 7 presents a comparative analysis of these constraints; Section 8 presents recommendations and Section 9 concludes the paper. Due to space limitation, we omit the union constraint from this paper.

2 Background

A class diagram is a structural abstraction of a real world phenomenon. The model consists of basic elements, descriptors and constraints. This section defines the abstract syntax and the semantics of the class diagram without the inter-association constraints. Our definitions follow the works of [2, 14]. We use a *property oriented* abstract syntax, since it enables a better formulation of the semantics of inter-association constraints. Inter-association constraints will be addressed separately in the latter sections.

Abstract Syntax. A *class diagram* is a tuple $\langle \mathcal{C}, \mathcal{P}, \mathcal{A}, props, props_{ac}, source, target, \mathcal{C}onstraint\rangle$ where

- $\mathcal{C}$ is a set of *class* symbols.
- $\mathcal{P}$ is a set of *property* symbols (sometimes called *association ends*). Property symbols denote mappings derived from their associations. *Ordering* and *uniquness* constraints are not discussed in this paper [15].

- $\mathcal{A}$ is a set of *association* symbols.
- $props \colon \mathcal{A} \to \mathcal{P} \times \mathcal{P}$ is a 1:1 assignment of different properties to association symbols. For an association a, $props(a) \neq \langle p, p \rangle$. For a property p, there is a unique $a \in \mathcal{A}$, such that $props(a) = (p, *)$ or $props(a) = (*, p)$, where * is a wild card. We write $assoc(p)$ or $assoc(p_1, p_2)$ for the association of p or of (p_1, p_2), and $props_1(a), props_2(a)$ for the two properties of a.
- $source \colon \mathcal{P} \to \mathcal{C}$ and $target \colon \mathcal{P} \to \mathcal{C}$ are 1:1 mappings of properties to classes such that for an association a with $props(a) = (p_1, p_2)$, $target(p_1) = source(p_2)$ and $target(p_2) = source(p_1)$. In Figure 1, $target(p_1) = source(p_2) = A$ and $source(p_1) = target(p_2) = B$.

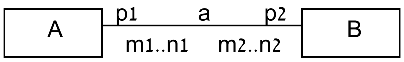


Fig. 1. Visualization of a binary association

- $\mathcal{AC}$ is a set of association-class symbols.
- $assoc_{AC} \colon \mathcal{AC} \to \mathcal{A}$ is a 1:1 assignment of association symbols to association class symbols. For an association class C, $pairs_C$ is a mapping between objects of C to object-pairs of $assoc_{AC}(C)$.
- $\mathcal{C}onstraint$ is a set of *constraints* as follows:
  1. *Multiplicity constraints on properties: $mul \colon \mathcal{P} \to (\mathbb{N} \cup \{0\}) \times (\mathbb{N} \cup \{*\})$* assigns multiplicity constraints to property symbols. For simplicity we use a compact symbolic representation, where association a in Figure 1 is denoted $a(p_1 : A[\, m_1,\ n_1],\ p_2 : B[\, m_2, n_2])$. The functions $minMul \colon \mathcal{P} \to \{\mathbb{N} \cup \{0\}\}$ and $maxMul \colon \mathcal{P} \to \{\mathbb{N} \cup \{*\}\}$ give the minimum and maximum multiplicities assigned to a property, respectively.
  2. *Aggregation and Composition:* Two unary relationships on the set of property symbols denoted by p^a and p^c respectively. Visually, composition is shown by a filled diamond adornment on the composite association end while aggregation is shown as an open diamond.
  3. *Class-hierarchy:* A non-circular binary relationship $\prec$ on the set of class symbols: $\prec\ \subseteq \mathcal{C} \times \mathcal{C}$. Henceforth we use the notation $C_2 \prec C_1$, where C_1 is the superclass and C_2 is the subclass (also called *direct-descendant*). The weak version of $\prec$ is denoted $\preceq$, which is "$\prec$ or equal". The reflexive transitive closure of $\prec$ is called the *descendant* relation, and denoted $\prec^*$. Its irreflexive version is denoted $\prec^+$. .
  4. *Generalization-set* (GS) *constraints:* GS is an $(n+1)$-ary relationship on $\mathcal{C}$, for $n \geq 2$. An element $\langle C, C_1, \ldots, C_n \rangle$ in GS must satisfy: For $i, j = 1..n, i \neq j$, (1) $C \neq C_i$; (2) $C_i \neq C_j$; (3) $C_i \prec C$. C is called the *superclass* and the C_i-s are called the *subclasses*. Elements of GS maybe associated with a *constraint* $const \in \{\langle disjoint \rangle, \langle overlapping \rangle, \langle complete \rangle, \langle incomplete \rangle, \langle disjoint, complete \rangle, \langle disjoint, incomplete \rangle, \langle overlapping, complete \rangle, \langle overlapping, incomplete \rangle\}$. We use the symbolic representation $GS(C, C_1, \ldots, C_n; const)$ for GS constraints. Note

that an unconstrained *GS* is redundant, as it specifies only class hierarchy constraints.

Semantics: The standard set theoretic semantics of class diagrams associates a class diagram with *instances* I, that have a semantic domain and an *extension mapping*, that associates syntactic symbols with elements over the semantic domain. For a symbol x, x^I is its denotation in I.

Symbol denotation:

- **Class:** For a class C, C^I, the extension of C in I, is a set of elements in the semantic domain. The elements of class extensions are called *objects*.
- **Property:** For a property p, p^I is a multi-valued function from its source class to its target class: $p^I : source(p)^I \rightarrow target(p)^I$.
- **Association:** For an association a, a^I, the extension of a in I, is a binary relationship on the extensions of the classes of a. If $props(a) = (p_1, p_2)$, then p_1^I and p_2^I are restricted to be inverse functions of each other: $p_1^I = (p_2^I)^{-1}$. The association denotes all object pairs that are related by its properties: $a^I = \{(e, e') \mid e \in target(p_1)^I,\ e' \in target(p_2)^I,\ e' \in p_2^I(e)\}$. The elements of association extensions are called *links*.
- **Association class**: For an association-class C, C^I is a set of elements in I and $pairs_C{}^I : C^I \rightarrow (assoc_{AC}(C))^I$ is a 1:1 and onto mapping. Furthermore, an object $e \in C^I$ is mapped to at most a single pair of objects in an association.

The semantics of the constraints with respect to an instance I:

1. **Multiplicity constraints on properties:** For a property p, p^I is restricted by the multiplicity constraints. For every $\mathsf{e} \in source(p)^I$, $minMul(p) \leq |p^I(\mathsf{e})| \leq maxMul(p)$. The upper bound is ignored if $maxMul(p) = *$.
2. **Aggregation**: An aggregation denotes two constraints:
   - *Irreflexivity*: For $\mathsf{e} \in source(p^a)^I$, $p^{a\,I}(\mathsf{e}) \neq \mathsf{e}$;
   - *Transitivity on the aggregation relation.*

   Together, these constraints imply *antisymmetry*:

   For aggregation properties $p_1^a, \ldots, p_n^a$, such that $target(p_i^a) = source(p_{i+1}^a), i = 1, n-1$, if $\mathsf{e} \in source(p_1^a)^I$, then $p_n^{a\,I}(p_{n-1}{}^{a\,I}(\ldots(p_1^{a\,I}(\mathsf{e})))) \neq \mathsf{e}$.
3. A **composition** is an aggregation with two additional constraints:
   - A composition property p^c is not multi-valued;
   - *multi-composition* constraint:
     For composite properties $p_1^c,\ p_2^c$ such that $source(p_1^c) = source(p_2^c)$, if $\mathsf{e} \in source(p_1^c)^I$, then $p_1^c(\mathsf{e}) = p_2^c(\mathsf{e})$.
4. **Class-hierarchy constraints:** Class hierarchy constraints denote subset relations between the extensions of the involved classes. That is, for $C_1 \prec C_2$, $C_1{}^I \subseteq C_2{}^I$.
5. **GS constraints** have the following meaning:
   (a) *disjoint:* $C_i^I \cap C_j^I = \emptyset, \forall i, j$
   (b) *overlapping:* For some i, j, it might be $C_i^I \cap C_j^I \neq \emptyset$

(c) *complete:* $C^I = \bigcup_{i=1}^{n} C_i^I$

(d) *incomplete*: $\bigcup_{i=1}^{n} C_i^I \subseteq C^I$

A *legal instance* of a class diagram is an instance that satisfies all constraints. A class diagram is *consistent* or *satisfiable* if it has a legal instance with non-empty class extensions, and is *finitely satisfiable* if it has a finite non-empty instance [13, 14, 16–20].

3 Subsetting Constraint

Subsetting is a basic inter-association constraint, quite popular in the UML meta-model. It is defined between two properties (association ends), constraining one to be a sub-mapping of the other. Figure 2 includes two subsetting constraints: The *presentation* property subsets the *paper* property, and the *presenter* property subsets the *author* property. The constraints state that a paper presented by an author must be a paper of the author, and the presenter of an accepted paper is also the author of that paper.

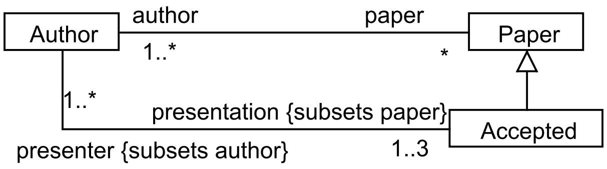


Fig. 2. A class diagram with subsetting constraints (Based on [21])

Subsetting is syntactically denoted by a binary relation on the set of property symbols: $\prec \ \subseteq \ \mathcal{P} \times \mathcal{P}$. $p_1 \prec p_2$, stands for "p_1 subsets p_2". p_1 is termed *the subsetting property*, and p_2 is termed *the subsetted end*. In Figure 2, *presentation* $\prec$ *paper* and *presenter* $\prec$ *author*. The UML specification imposes the following constraints on $p_1 \prec p_2$: $source(p_1) \prec^* source(p_2)$, $target(p_1) \prec^* target(p_2)$ and $maxMul(p_1) \leq maxMul(p_2)$.

Subsetting is the only inter-association constraint for which there is an overall agreement about its semantics. For $p_1, p_2 \in \mathcal{P}$, $p_1 \prec p_2$ states that p_1 is a sub-mapping of p_2, i.e., for a legal instance I, $\mathsf{e} \in source(p_1)^I$ implies ${p_1}^I(\mathsf{e}) \subseteq {p_2}^I(\mathsf{e})$.

Semantic properties of the subsetting constraint: Let $p_1,\ p_2 \in \mathcal{P}, p_1 \prec p_2$, and I a legal instance. Then:

1. **Symmetry:** [2, 5] $p_1^{-1} \prec p_2^{-1}$.
2. **Association inclusion:** [5] $assoc(p_1)^I \subseteq assoc(p_2)^I$. Furthermore, the meta-model [1] states that subsetting implies association specialization (see section 5).
3. [5]: If $minMul(p_1) > minMul(p_2)$, there exist an object $\mathsf{e} \in source(p_2)^I$ such that $|p_2^I(\mathsf{e})| \geq minMul(p_1)$.

3.1 Interaction with Other Constraints

The interaction of the subsetting constraint with other constraints can cause various correctness problems. Below, we analyze some problems, and point the reader to our catalog [12] for other problems.

A finite satisfiability problem due to conflicts with multiplicity constraints: Consider Figure 3. In every legal instance I, for $\mathtt{e} \in C^I$, $|b_3^I(\mathtt{e})| = 2$, and also $b_3^I(\mathtt{e}) \subseteq b_1^I(\mathtt{e})$. But, the multiplicity constraints on association r dictate $|A^I| = |B^I|$, implying that either I is an infinite instance or $|b_1^I(\mathtt{e})| = 1$. Therefore, the diagram does not have a finite non-empty legal instance, implying that the diagram is not finitely satisfiable[1]. This example shows how the surrounding context of properties can affect the correctness of the diagram. For other conflicts of subsetting with multiplicity constraints, such as in presence of *qualifier* constraints, consult [12].

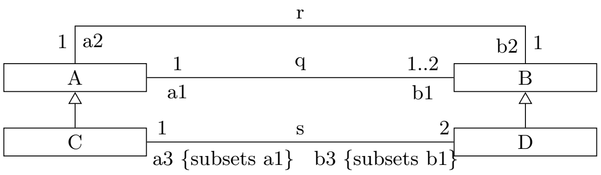


Fig. 3. A finite satisfiability problem due to interaction between subsetting and multiplicity constraints

Correctness problems due to conflicts with aggregation/composition constraints: Aggregation/Composition constraints are frequently used in the metamodel and in conceptual modeling. We analyze here only consistency problems caused by interaction with subsetting. For more finite satisfiability problems consult [12].

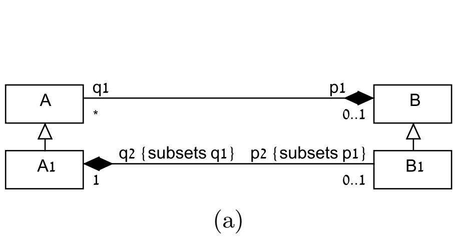
Figure 4a presents a consistency problem due to *composition cycles* (noticed by [2]): In every legal instance I, for $\mathtt{e} \in B_1^I$, $p_1^{c\,I}(q_2^{c\,I}(\mathtt{e})) = \mathtt{e}$. Figure 4b has the same consistency problem, but the composition cycle is caused indirectly, due to interaction with the subsetting constraints.

Claim 1 generalizes the result of [2] for *indirect composition cycles.*

Claim 1. *Let $p, q, r \in \mathcal{P}$, where $r \prec^* p$ and $r^{-1} \prec^* q$. Then, p and q are not both composition properties.*

Proof. (Sketched) Assume by contradiction that both p and q are composition properties (denoted, for clarity p^c, q^c). Let I be a legal instance and $\mathtt{e} \in source(r)^I$, such that $p^{c\,I}(\mathtt{e}) \neq \emptyset$. Then, $r^I(\mathtt{e}) = p^{c\,I}(\mathtt{e})$, and ${r^{-1}}^I(r^I(\mathtt{e})) = q^{c\,I}(r^I(\mathtt{e}))$ (due to subsetting and composition), imply ${r^{-1}}^I(r^I(\mathtt{e})) = \mathtt{e}$. Therefore, $q^{c\,I}(p^{c\,I}(\mathtt{e})) = \mathtt{e}$, i.e., a composition cycle. Consequently, at most one of p or q can be a composition property.

[1] A different way to describe this problem is noting that the r, q association cycle implies that the maximum multiplicity 2 on b_1 is not realized in any legal finite instance, i.e., the multiplicity constraint on b_1 is not *tight*. But, tightening $maxMul(b_1)$ to be 1 creates a consistency problem in finite legal instances.



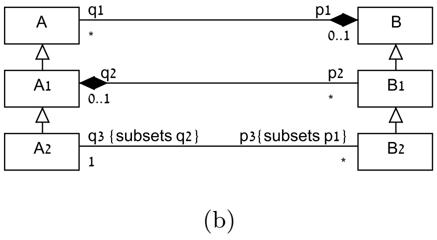


Fig. 4. Consistency problems due to subsetting and (indirect) composition cycles

Induced constraints, due to interaction with disjoint GS constraints: The subsetting constraint affects disjoint GS constraints on parallel hierarchies, as demonstrated in Figure 5. The subsetting constraints induce a disjoint constraint on $GS(A, A_1.A_2)$, and therefore the diagram is incomplete. In order to see that, let I be a legal instance, and assume that there exists $\mathsf{e} \in A_1^I \cap A_2^I$. Then, $|p_1^I(\mathsf{e})| = 1$, $|p_2^I(\mathsf{e})| = 1$ and $p_1^I(\mathsf{e}) \cap p_2^I(\mathsf{e}) = \emptyset$. The subsetting constraints imply $(p_1^I(\mathsf{e}) \cup p_2^I(\mathsf{e})) \subseteq p^I(\mathsf{e})$, which further implies $|p^I(\mathsf{e})| \geq 2$, in contradiction to the maximum multiplicity constraint on p. Therefore, A_1^I and A_1^I are disjoint in any legal instance. Claim 2 generalizes the result.

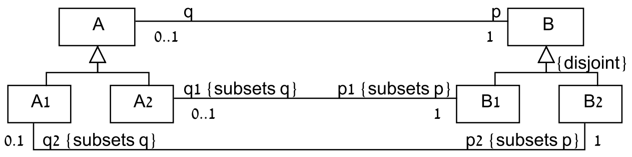


Fig. 5. Interaction of *subsetting* with disjoint constraint

Claim 2. *Assume that a class diagram includes a GS constraint $GS(B, B_1 \ldots, B_n; disjoint)$, and properties $p_i \prec p$ such that $target(p_i) \prec^* B_i$, for $i = 1, n$. Then, if for every i, j, $i \neq j$, $minMul(p_i) + minMul(p_j) > maxMul(p)$, the class diagram entails the induced GS constraint $GS(source(p), source(p_1), \ldots, source(p_n); disjoint)$.*

Proof. Assume by contradiction that there exists a legal instance I with an object e such that for some $i \neq j$, $\mathsf{e} \in source(p_i)^I \cap source(p_j)^I$. Then, since $target(p_i)^I$ and $target(p_j)^I$ are disjoint, $|p_i^I(\mathsf{e}) \cup p_j^I(\mathsf{e})| \geq minMul(p_i) + minMul(p_j) > maxMul(p)$. But, $p_i^I(\mathsf{e}) \cup p_j^I(\mathsf{e}) \subseteq p^I(\mathsf{e})$, implying $|p^I(\mathsf{e})| > maxMul(p)$, in contradiction to the multiplicity constraint. A special case of this claim involves properties p_i for which $minMul(p_i) = maxMul(p)$.

4 Property Redefinition Constraint

Property redefinition, like subsetting, is a constraint that is imposed between properties, and is frequently used in the meta-model. It can refer to several

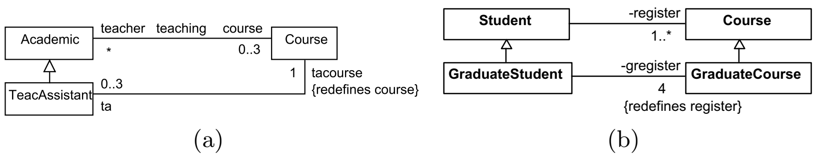


Fig. 6. Class diagrams with redefinition constraints

property characteristics like *name, time, visibility, multiplicity, type* [1, 22], of which we discuss only type and multiplicity constraints.

Figure 6 demonstrates type and multiplicity redefinition constraints. In Figure 6a, property *tacourse* redefines the multiplicity constraint of property *course.* It restricts a teaching assistant to teach exactly one course, although Academic employees can teach up to three courses. In Figure 6b, property *gregister* redefines the type and multiplicity of property *register.* It restricts a graduate student to register to exactly four graduate courses instead of to any number of general courses.

Redefinition is syntactically denoted by a binary relation on the set of property symbols: $\rhd \subseteq \ \mathcal{P} \times \mathcal{P}$. $p_1 \rhd p_2$, stands for "p_1 redefines p_2". p_1 is termed *the redefining property* and p_2 is termed *the redefined property.* In Figure 6a, *tacourse* $\rhd$ *course.*

The UML specification imposes the following constraints on $p_1 \rhd p_2$: $source(p_1)$
$\prec^+ source(p_2)$, $target(p_1) \prec^* target(p_2)$, $minMul(p_1) \geq minMul(p_2)$ and $maxMul(p_1) \leq maxMul(p_2)$. In addition, since the class hierarchy relation is acyclic (constraint 2 in [1, p. 52]), redefinition is also acyclic.

The semantics of redefinition, as explained in the metamodel [1, 23], is unclear and contains contradictory statements. Three different interpretations have been discussed in the literature [2, 4, 7–10], with no single agreement. We consider each separately, and then present criteria for selecting one. Let $p_1, p_2 \in \mathcal{P}$, $p_1 \rhd p_2$. The interpretations are:

1. **Single association (refinement) semantics:** [5, 9, 24] p_2 values on $source(p_1)$ are restricted to be in $target(p_1)$, i.e., for a legal instance I, $\mathsf{e} \in source(p_1)^I$ implies ${p_2}^I(\mathsf{e}) \in target(p_1)^I$ and $minMul(p_1) \leq |{p_2}^I(\mathsf{e})| \leq maxMul(p_1)$. The redefining association $aassoc(p_1)$ is ignored, and is not treated as an independent association.
2. **Overriding (covariant) semantics:** [2, 7] The redefined property p_2 is not defined on $source(p_1)$, i.e., for a legal instance I, $p_2^I : source(p_2)^I - source(p_1)^I \to target(p_2)^I$.
3. **Subsetting semantics:** [6, 8, 10] The redefined property p_2 is restricted to be the redefining property p_2 on $source(p_1)$, i.e., for a legal instance I, $\mathsf{e} \in source(p_1)^I$ implies ${p_2}^I(\mathsf{e}) = {p_1}^I(\mathsf{e})$. According to this interpretation, redefinition strengths the subsetting constraint with additional constraints.

In order to recommend a semantics for adoption, we single out *observables* of redefinition, i.e., characteristics that are agreed by most interpretations. Our criteria for semantics selection involve (1) preservation of observables; (2) minimizing contradiction with the meta-model; (3) consistency with the selected semantics of other UML constraints.

Redefinition Observables: Let $p_1 \rhd p_2$, and let I be a legal instance. The following two properties seem to characterize most texts that discuss redefinition:

1. **Type redefinition**: For object $\mathtt{e} \in source(p_1)^I$, ${p_2}^I(\mathtt{e}) \in target(p_1)^I$.
2. **Multiplicity redefinition**: For object $\mathtt{e} \in source(p_1)^I$, $maxMul(p_1) \leq |{p_2}^I(\mathtt{e})| \leq maxMul(p_1)$

The single association semantics satisfies the observables. Therefore, we consider the other two criteria. We suggest to reject this semantics since it ignores the redefining association, practically treating it as removed. The problem is that the meta-model [1], states that redefinition entail association specialization, implying that the redefining association has a role in the diagram. Besides, this semantics ignores the p_1^{-1} property, including its possible associated constraints, like multiplicity, qualifier, composition and subsetting constraints.

The overriding semantics does not satisfy the observables. Besides, under this semantics, subsetting and redefinition contradict each other. But, the combination of redefinition and subsetting on the same property occurs in the meta-model (e.g., Figure 12.21 on [1, p. 307]). In addition, covariant subtyping is problematic since a client of $source(p_2)$, when actually holding an object from $source(p_1)$, expects the p_2 property to be applicable [4, 25].

The subsetting semantics satisfies the observable, and is consistent with the UML meta-model and with other inter-relationship constraints. In particular, it entails subsetting. Therefore, we suggest adopting it as the semantics of the redefinition constraint.
note: Unlike subsetting, redefinition is not necessarily symmetric. That is, of p has a redefinition constraints, then p^{-1} is not necessarily a redefinition property.

5 Association Specialisation

Association is a *classifier*, and therefore can be specialized by another association. Like class hierarchy, association specialization appears in early versions of UML, with an inclusion semantics. Figure 7 show an association specialization between the *special authorization* and the *authorization* associations. It enforces an external user with a special authorization access to some wireless network, to have a regular user authorization to the same network. Association specialization is cognitively complex, and is not frequently used.

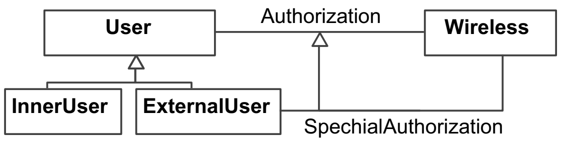


Fig. 7. Association Specialisation

Association specialization is syntactically denoted by a binary relation on the set of association symbols: $\prec \subseteq \mathcal{A} \times \mathcal{A}$. $a_1 \prec a_2$ stands for "a_1 specializes a_2". a_1 is the *super-association* and a_2 is the *sub-association*. The UML specification constrains association specialization to be acyclic and requires hierarchy among the involved classes. That is, for $a_1 \prec a_2$, with $props(a_i) = (p'_i, p''_i), i = 1, 2$, $source(p'_1) \prec^* source(p'_2)$ and $source(p'_1) \prec^* source(p'_2)$. This causes the properties of the associations to be covariant, which is problematic for subtyping. In addition, see below for problems of *property correspondence*.

Although association specialization is an earlier concept, its semantics is still unsettled. The UML specification includes two references to its semantics. In one place, association specialization is given a *generalization semantics*, which implies an *inclusion relation* between the instances of the involved associations. In another place, it is given a *specialization semantics* which implies intentional relationship between the related associations.

Observable of Association Specialisation: Let $a_1 \prec a_2$ and let I be a legal instance. Then, $a_1^I \subseteq a_2^I$.

The inclusion observable reveals an inherent problem in the syntactic specification of association specialization: It neglects the correspondence between the properties of the association. Therefore, it does not provide sufficient information as to the order of the objects in included pairs. Figure 8 presents two legal instance diagrams for the class diagram in Figure 8a. In both instances, sp_1 is a *manager* and sp_2 is a *special*. The difference is that in Figure 8b, sp_1 is a grantor and sp_2 is a grantee, while in Figure 8c sp_1 is a grantee and sp_2 is a grantor. The problem is that the association specialization syntax is lacking – it does not specify correspondences between the properties of the sub-association and the super-association. We return to this point after the discussion of the semantics.

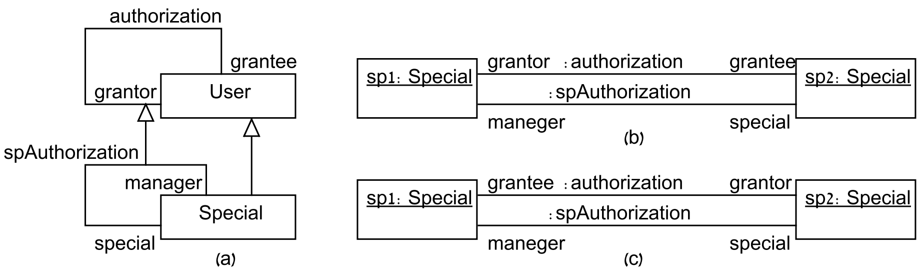


Fig. 8. An association hierarchy between two reflexive assiciations

Four possible interpretations for association specialization appear in the literature:

1. **Inclusion semantics:** [3, 21] Association specialization is interpreted as the inclusion observable.
2. **Covariant semantics:** [26, 27] This approach considers association specialization as redefinition on the two properties of the sub-association, with the overriding (covariant) semantics.

3. **Redefinition semantics** [22, 28, 29]: Association specialization is equivalent to redefinition constraints on both properties of the sub-association.
4. **Subsetting semantics:** [6, 8, 10] Association specialization is equivalent to subsetting constraints on the properties of the sub-association.

The inclusion semantics has the property correspondence problem. The covariant semantics relies on a redefinition semantics that does not satisfy the observables of redefinition, and therefore also does not satisfy the observable of association specialization. The redefinition semantics implies that association specialization entails subsetting. But, since the meta-model specifies that subsetting entails association specialization [1], the conclusion is that association specialization is equivalent to subsetting, which is the third and most popular semantics.

In order to fully specify the subsetting semantics of association specialization, the syntax should be strengthened to include specification of property correspondence as subsetting constraints. For example: $specialAuthorization \prec authorization$ with $manager \prec grantor$ and $sp \prec grantee$.

Association hierarchies with generalization set constraints: The metamodel allows defining GS constraints with association specializations. Generalization set constraints over association specialization or association class hierarchies have been used in the translation of description logic into UML [18].

The syntax is similar to the syntax of the generalization set above classes: $GS(a, a_1, \ldots, a_n; const)$, $a_i \prec a$ where $const$ is one of the GS constraints defined for classes. The semantics of GS constraints over associations is similar to that of GSs over classes. While association specialization without GS constraint can be replaced by subsetting constraints, GS constraints require the explicit presence of association specialization.

The interaction of association specialization with association GS constraints and with other constraints yields correctness and quality problems that are analyzed in [12].

6 Association-Class Hierarchy

Association class is defined in [1] as a sub-class of the classifiers *Association* and *Class*, and therefore it can be specialized by another association class. Association classes appear frequently in the translation of description logics to class diagrams, but in other domains its usage is rare. Syntactically, it is a binary relation on the set of association class symbols: $\prec \subseteq \mathcal{AC} \times \mathcal{AC}$.

The semantics of association-class hierarchy has received little attention in the literature and its definition in the metamodel is unclear. Its intuitive semantics is a mixture of the semantics of class generalization and association specialization, and can be interpreted in different ways. Three criteria, from which we derive the observables, seem to characterize association-class hierarchy:

1. **Class generalization:** The sub-association-class denotes a subset of the super-association-class denotation.
2. **Association specialization:** The sub-association is a specialization of the super-association.

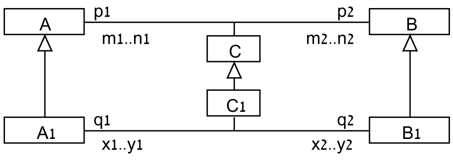
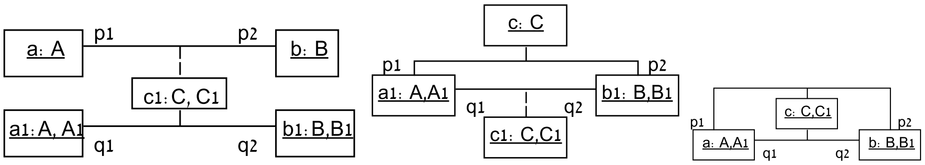


Fig. 9. Association class hierarchy

3. **Association class:** The association-class nature of the association-classes is not violated.

Observables of association-class hierarchy:

1. **No-double-pair:** An object of the sub-association-class C does not identify (through the $pairs_C$ mapping) two different pairs, one in the sub-association and one in the super-association (as demonstrated in Figure 10a).
2. **No-double-object:** A pair of objects in the sub-association is not identified by two different objects of the sub-association-class and the super-association-class (as demonstrated in Figure 10b).
3. **Association subsetting:** The sub-association satisfies subsetting constraints with respect to the super-association. Note, that this observable implies that property correspondence between the sub and super associations is settled, and the sub-association inherits the multiplicity constraints of the super-association[2].



(a) An object identifis 2 pairs (b) 2 objects identify a pair (c) Recommended

Fig. 10. Instances of Figure 9 according to three possible interpretations

Possible interpretations for association-class hierarchy can be considered:

1. **Class-hierarchy semantics**: The hierarchy constraint affects only the associated classes, with no reference to the association of the association-class.
2. **Class-hierarchy + subsetting semantics**: The class hierarchy constraint is strengthened with subsetting between the involved associations.
3. **Class-hierarchy + subsetting + single-mapping semantics:** The class hierarchy and subsetting constraints are strengthened by a requirement for a single pair identification for objects in the sub-association-class: For association classes $C_1 \prec C_c$, and a legal instance I, $(pairs^I_{C_2})_{/C^I_1} = pairs^I_{C_1}$.

[2] Some versions of the *USE* system [30] do not satisfy this observable.

The class-hierarchy semantics does not satisfy the no-double-pair observable, and the subsetting observables. The class-hierarchy + Subsetting semantics does not satisfy the no-double-object observable. The class-hierarchy + subsetting + single-mapping semantics satisfies all three observables (Figure 10c demonstrates this semantics), and therefore is recommended as most appropriate.

Interaction with other constraints is omitted, for lack of space.

7 Comparative analysis of Inter-association Constraints

Characterization: Syntax and Semantics. Subsetting and redefinition are constraints between properties, association specialization is a constraint between associations and association-class hierarchy is a constraint between the associated-classes. The semantic characterization covers several aspects:

1. **Expressive power:** Subsetting and association specialization are equivalent; redefinition entails subsetting and association-class hierarchy entails association specialization.
2. **Symmetry:** Subsetting is symmetric with respect to the association on which it is imposed, while redefinition is not.
3. **Semantics variation status:** Subsetting has an overall agreement about its semantics. The semantics of redefinition is still controversial. There is disagreement between the three interpretations, termed “single association”, “covariant” and “subsetting”. We have shown that the first two are not consistent with the meta-model. The semantics that we recommend for association specialization, i.e., subsetting, is widely accepted, although its intentional aspect is still unclear. The semantics of association-class hierarchies is hardly addressed.

Complexity Factors

Inter-association constraints raise several complexity issues with respect to model evaluation and modeling activity. *Cognitive complexity* is a modeling activity factor that refers to the mental burden that people (e.g. analysts, designers, developers, etc.) experience when building, validating, verifying or using models [31, 32]. *Design problems* and *structural complexity* are model evaluation factors. Design problems refer to correctness and quality of models, and structural complexity refers to the inter-relationships between model elements [31, 32].

Design Problems. All inter-association constraints cause correctness and quality problems due to their complex interaction with other constraints. The complex interaction can be local or global, and involves the associated classes, which are already constrained among themselves.

Structural and Cognitive Complexity. Structural and cognitive complexity [31] are widely recognized as factors that affect model understandability. Structural complexity affects cognitive complexity, which reduces understandability [32]. Empirical studies show that class diagram complexity is an indication for cognitive complexity and external quality attributes such as maintainability and modifiability [33]. For example, deep class hierarchies decrease understandability.

The effect of inter-association constraints on the structural properties of class diagrams has been not studied yet. We hypothesize that they increase cognitive complexity and reduce model understandability, for the following reasons:

1. **Global inter-relationships:** The association ends that are involved in a constraint can appear in a far distance from each other (high hierarchy depth in association class/association specialization). This occurs frequently in the meta-model, where finding a subsetted property might require high navigation skills through multiple meta-model diagrams.
2. **Induced constraints:** The amount of induced constraints due to interaction with other constraints is quite high. Identification of induced constraints requires understanding of indirect relationships between constraints, sometimes within a large scope (e.g. identifying indirect composition cycles).

Visual Aspect. The visual representation of model element affects its cognitive complexity. Good visual representations enjoy the *cognitive effectiveness* property, i.e., the ability to directly clarify cognitive to visual translations [34].

– **Subsetting and Redefinition:** Their visual notations of are textual, and do not rely on efficient cognitive processes [34]. Their major problem is that their notations have a global scope, that might be the entire class diagram. Besides, the syntax $\{subsets\ x\}$ or $\{redefines\ x\}$ does not indicate which class is related to property x. additional issues are that subsetting and redefinition notations do not show the relationship between the super and sub-associations, and subsetting does not reflect its *symmetric nature.*
– **Association Specialization and Association-Class Hierarchy:** The visual notation of association specialization and of association-class hierarchy is intuitive because it directly specifies the involved associations. Its visual notation indicates its semantic properties of the constraint: *transitivity*, *irreflexivity*, and *asymmetry*[34, 35]. The visual notation, however, does not show the subsetting relationship between the association ends, which increases clarity but creates redundancy. In complex structures, association specialization might create edge crossings, that decreases understandability.

Usefulness. Subsetting, union, and redefinition are frequently used by the meta-model, in that frequency order. Association specialization and association-class hierarchy are not used. Description logics use association specialization and association-class hierarchy in translations to UML.

8 Recommendations and Guidelines for Using Inter-association Constraints

Semantics Decisions

1. **Redefinition:** We suggest strengthening the meta-model to include the entailment relationship between the redefinition and subsetting constraints, and the symmetry of subsetting. Furthermore, the meta-model can be simplified, by removing combined specification of subsetting and redefinition.

2. **Association-class hierarchy and Association specialization:** We recommend (1) enforcing specification of subsetting constraints; (2) modifying the meta-model so that association specialization do not entail class hierarchy between the associated classes.
3. **Visual decisions:** We suggest updating the visual representation of subsetting and redefinition to include information about the classes. A possible representation is $\{subsets\ C.p\}$, where $C = source(p)$.

Modeling Guidelines and Recommendations

1. Overuse of inter-association constraints, as in the meta-model, increases structure complexity, cognitive complexity, and the number of induced constraints and correctness problems.
2. (a) Since subsetting is symmetric, define subsetting on both properties of an association, unless there is a redefinition constraint. This can decrease ambiguity, where a reader may conclude that one property includes more constraints than the other.
   (b) In case of redefinition on both properties of an association, subsetting is redundant since redefinition entails subsetting. Combining it with subsetting can confuse.
   (c) If only one property of an association includes a redefinition constraint, adding subsetting on the other property may increase understandability.
3. Avoid subsetting/redefinition of properties in far distance. This can help in discovering the design problems caused by these constraints. We recommend defining subsetting/redefinition constraints on a property in an ascending order according to their distance in the hierarchy of the associated classes.
4. Involving inter-association constraints with *GS* constraints is a potential of design problems.
5. Association-class hierarchy and association specialization: The semantics of association-class hierarchy is not intuitive, while association specialization is equivalent to subsetting. Both constraints require subsetting constraints and their visual notation in complex hierarchy structures is obscure.
   Therefore, we recommend using these constraints only if they are part of *GS* constraints (on classes or on associations).

9 Conclusion and Future Work

We presented a coherent approach for choosing the semantics of inter-association constraints in UML2, with a maximum compatibility with the Meta-Model, and based on common observables. The paper addressed correctness problems that result from these constraints which we added to our catalog [12]. The paper presented a comparative analysis of these constraints, semantics recommendations and modeling guidelines.

The inter-association constraints contribute markedly to the complexity of class diagram constraints. Modelers cannot be expected to master the complex interactions among constraints. Therefore, the well-formedness rules should be automated and embedded in model-level IDEs. Another research direction involves the study of metrics for *model complexity.*

References

1. OMG: UML 2.4 Superstructure Specification. Specification Version 2.4.1, Object Management Group (2011)
2. Alanen, M., Porres, I.: A Metamodeling Language Supporting Subset and Union Properties. Software and Systems Modeling 7, 103–124 (2008)
3. Artale, A., Calvanese, D., Ibanez-Garcia, A.: Full Satisfiability of UML Class Diagrams. In: Parsons, J., Saeki, M., Shoval, P., Woo, C., Wand, Y. (eds.) ER 2010. LNCS, vol. 6412, pp. 317–331. Springer, Heidelberg (2010)
4. Costal, D., Gómez, C.: On the Use of Association Redefinition in UML Class Diagrams. In: Embley, D.W., Olivé, A., Ram, S. (eds.) ER 2006. LNCS, vol. 4215, pp. 513–527. Springer, Heidelberg (2006)
5. Costal, C., Gómez, C., Nieto, P.: On the Semantics of Redefinition, Specialization and Subsetting of Associations in UML (Extended Version). Technical report, Universitat Politcnica de Catalunya (2010)
6. Costal, D., Gómez, C., Guizzardi, G.: Formal Semantics and Ontological Analysis for Understanding Subsetting, Specialization and Redefinition of Associations in UML. In: Jeusfeld, M., Delcambre, L., Ling, T.-W. (eds.) ER 2011. LNCS, vol. 6998, pp. 189–203. Springer, Heidelberg (2011)
7. Kleppe, A., Rensink, A.: On a Graph-Based Semantics for UML Class and Object Diagrams. In: Graph Transformation and Visual Modelling Techniques, EASST, vol. 10 (2008)
8. Amelunxen, C., Schürr, A.: Formalising Model Transformation Rules for UML/-MOF 2. IET Software 2(3), 204–222 (2008)
9. Nieto, P., Costal, D., Gomez, C.: Enhancing the Semantics of UML Association Redefinition. Data & Knowledge Engineering 70(2), 182–207 (2011)
10. Bildhauer, D.: On the Relationships Between Subsetting, Redefinition and Association Specialization. In: Ninth Conference on Databases and Information Systems (2010)
11. Maraee, A., Balaban, M.: On the Interaction of Inter-Relationship Constraints. In: Workshop on Model-Driven Engineering, Verification and Validation (MoDeVVA 2011). MoDELS 2011 (2011)
12. BGU Modeling Group: UML Class Diagram Pattern Catalog (2010), http://www.cs.bgu.ac.il/~cd-patterns/
13. BGU Modeling Group: FiniteSatUSE – A Class Diagram Correctness Tool (2011), http://sourceforge.net/projects/usefsverif/
14. Balaban, M., Maraee, A.: Finite Satisfiability of UML Class Diagrams with Constrained Class Hierarchy. ACM Transactions on Software Engineering and Methodology (TOSEM) (to appear)
15. Milicev, D.: On the Semantics of Associations and Association Ends in UML. IEEE Transactions on Software Engineering 33, 238–251 (2007)
16. Cadoli, M., Calvanese, D., De Giacomo, G., Mancini, T.: Finite Satisfiability of UML Class Diagrams by Constraint Programming. In: The Workshop on CSP Techniques with Immediate Application (2004)
17. Boufares, F., Bennaceur, H.: Consistency Problems in ER-schemas for Database Systems. Information Sciences, 263–274 (2004)
18. Berardi, D., Calvanese, D., Giacomo, D.: Reasoning on UML Class Diagrams. Artificial Intelligence 168, 70–118 (2005)

19. Maraee, A., Makarenkov, V., Balaban, B.: Efficient Recognition and Detection of Finite Satisfiability Problems in UML Class Diagrams: Handling Constrained Generalization Sets, Qualifiers and Association Class Constraints. In: MCCM 2008 (2008)
20. Queralt, A., Teniente, E.: Verification and Validation of UML Conceptual Schemas with OCL Constraints. ACM Transactions on Software Engineering and Methodology (TOSEM) 21, 13:1–13:41 (2012)
21. Szlenk, M.: UML Static Models in Formal Approach. In: Meyer, B., Nawrocki, J.R., Walter, B. (eds.) CEE-SET 2007. LNCS, vol. 5082, pp. 129–142. Springer, Heidelberg (2008)
22. Rumbaugh, J., Jacobson, G., Booch, G.: The Unified Modeling Language Reference Manual, 2nd edn. Adison Wesley (2004)
23. OMG: UML 2.4 Infrastructure Specification. Specification Version 2.4, Object Management Group (2011)
24. Olivé, A.: Conceptual Modeling of Information Systems. Springer (2007)
25. Buttner, F., Gogolla, M.: On Generalization and Overriding in UML 2.0. In: UML Modeling Languages and Applications. Springer (2004)
26. Snoeck, M., Lemahieu, W.: Specializing Associations. Technical Report 0329, Katholieke Universiteit Leuven (2003)
27. Varro, D., Pataricza, A.: VPM: A Visual, Precise and Multilevel Metamodeling Framework for Describing Mathematical Domains and Metamodeling Framework for Describing Mathematical Domains and UML. Softw. Syst. Model 2, 180–210 (2003)
28. Pons, C.: Generalization Relation in UML Model Elements. In: Inheritance Workshop at European Conference for Object-Oriented Programming, ECOOP (2002)
29. Monperrus, M., Beugnard, A., Champeau, J.: A Definition of "Abstraction Level" for Metamodels. In: 16th Annual IEEE International Conference and Workshop on the Engineering of Computer Based System (2009)
30. Bremen Database Systems Group: A UML-based Specification Environment (2012), http://www.db.informatik.uni-bremen.de/projects/USE/
31. Btiand, L., Lounis, H., Wuest, J.: A Comprehensive Investigation of Quality Factors in Object-oriented Designs: An Industrial Case Study. In: The 21st International Conference on Software Engineering, pp. 345–354 (1999)
32. Cruz-Lemus, J., Maes, A., Genero, M., Poels, G., Piattini, M.: The Impact of Structural Complexity on the Understandability of UML Statechart Diagrams. Information Sciences 180, 2209–2220 (2010)
33. Genero, M., Manso, E., Visaggio, A., Canfora, G., Piattini, M.: Building measure-based prediction models for uml class diagram maintainability. Empirical Software Engineering 12, 517–549 (2007)
34. Moody, D.: The "Physics" of Notations: Towards a Scientific Basis for Constructing Visual Notations in Software Engineering. IEEE Transactions on Software Engineering 35, 756–779 (2009)
35. Gurr, C.: Effective Diagrammatic Communication: Syntactic, Semantic and Pragmatic Issues. Journal of Visual Languages and Computing 10, 317–342 (1999)

The Coroutine Model of Computation

Chris Shaver and Edward A. Lee

EECS Department, University of California Berkeley
{shaver,eal}@eecs.berkeley.edu

Abstract. This paper presents a general denotational formalism called the *Coroutine Model of Computation* for control-oriented computational models. This formalism characterizes atomic elements with control behavior as *Continuation Actors*, giving them a static semantics with a functional interface. *Coroutine Models* are then defined as networks of Continuation Actors, representing a set of control locations between which control traverses during execution. This paper gives both a strict and non-strict denotational semantics for Coroutine Models in terms of compositions of Continuation Actors and their interfaces. In the strict form, the traversal of control locations forms a control path producing output values, whereas in the non-strict form, execution traverses a tree of potential control locations producing partial information about output values. Furthermore, the given non-strict form of these semantics is claimed to have useful monotonicity properties.

1 Introduction

Let a *control-oriented model* describe a system characterized by a network of control locations traversed sequentially during execution. At each location visited during execution, an action may be performed that produces outputs or manipulates the state of the system. However, at each location there is also a determination of how the traversal through the network of locations will subsequently progress. This determination can depend on both the inputs to the system, as well as its state, and is often represented as a conditional or guard. In some control-oriented models this determination can also include the possibility of the model either *suspending* or *terminating* its thread of control in the context of a larger model or execution environment.

Examples of control-oriented models include traditional imperative programming models, control flow graphs, and automata-based models such as state machines or labeled transition systems. Languages such as Esterel [4], Reactive C [5], and StateCharts [10] are also control-oriented in this sense. In the case of Esterel or Reactive C, the control locations correspond to individual imperative statements in the language, with the traversal of these locations corresponding to the control flow of the language. In StateCharts control locations are simply states with traversal governed by the guards of transitions. These three examples have the additional feature of suspending and resuming control over a series of *reactions*, as defined by Boussinot [4]. In the **Ptolemy II** environment, Modal

Models [14] give a hierarchical layer of control-oriented behavior to heterogeneous models. Modal Models are guarded state machines where at each state there is an associated actor, known as a refinement, that is fired when the model is in that state.

Although this characterization of control-oriented behavior is very general, it stands in contrast with Dataflow models where the behavior of a system is structured in terms of the movement of data rather than control. Counterexamples then include Kahn Process Networks [11], Dataflow as described by Lee and Matsikoudis [12], and Stream models such as those of Broy et al. [6]. However, it can be the case that a control-oriented model can participate in a heterogeneous system where a dataflow actor is internally control-oriented or a control-oriented model contains control locations at which a dataflow process represents the associated action taken, as can be the case in Modal Models for instance. Additionally, it must be emphasized that the monotonicity discussed in this paper is not the same as that of stream functions addressed in papers such as [6].

When these control-oriented models represent isolated models of computation, the formal treatment of their meaning in terms of operational semantics provides a clear way to reason about them, and gives a way to determine how to correctly implement them. However, particularly in the context of heterogeneous models, it is difficult to reason about compositional properties of these models given there is no clear general way to compose operational semantics such as those given by Boussinot and de Simone for Esterel [4], Berry for Constructive Esterel [3], and Andre for SyncCharts [1]. Since these languages are all both control-oriented and synchronous, a motivating kind of heterogeneity arises when these languages are decomposed into control-oriented fragments in synchronous compositions, as well as in systems with dataflow components, as was mentioned above.

In the case of SyncCharts, these two components are the control-oriented *State Transition Graphs*, which are similar to StateCharts [10], and synchronous compositions of *Macrostates* [1]. While in the operational semantics given by Andre [1] these two components are entangled, a denotational semantics would allow each of these parts to be treated separately, and the full model to arise out of their heterogeneous composition. Such a denotational formalism for synchronous composition exists in the Synchronous Reactive (SR) model of computation given by Edwards [8]. In this model, the semantics of a step in execution is given by the least fixed point of the function derived from composing the functional representations of each component in the model. So long as these components can be represented as monotonic functions, this least fixed point is guaranteed to uniquely exist.

Using the SR model to express synchronous composition, one should be able to achieve a model similar to that of SyncCharts or Constructive Esterel as a heterogeneous, hierarchical composition of control-oriented models and SR models. But, reasoning about this composition requires a general denotational semantics for control-oriented models. In particular, with this kind of semantics the conditions can be determined under which such a model is monotonic. Having such a denotational semantics for control-oriented models facilitates the analysis of other compositional properties of these models as well. Like the SR model, there are other models of computation that can similarly be described in a compositional way, as is done by Tripakis et al. [16]. With the semantics given in this paper, meaningful compositions can be formed between control-oriented models and these other models.

1.1 Contributions

In order to reason about control-oriented models in a compositional manner, this paper presents the *Coroutine Model of Computation*, a general denotation formalism for control-oriented models. This model consists of atomic elements called *Continuation Actors*, and defines *Coroutine Models* as networks of these Continuation Actors. A Coroutine Model composes Continuation Actors to form itself a Continuation Actor. Taking influence from the idea of *stars* and *Reactive Cells* from Andre's SyncCharts [1,2], the decisions to take control transitions in Coroutine Models are treated as part of the individual Continuation Actors in the network. This choice avoids having to settle on a particular language and semantics for transition guards and actions, and leads to a simple compositional semantics for Coroutine Models, defined in terms of the behavior of their constituting Continuation Actors.

Moreover, a meaning is given to non-strict Continuation Actors, which can make partial control decisions given partial inputs. Correspondingly, a non-strict dynamic semantics is defined for Coroutine Models containing these non-strict Continuation Actors. Further, it is argued that these semantics, in fact, form monotonic functions when the constituting Continuation Actors of a model are monotonic. Thus, Coroutine Models can be meaningfully put into synchronous compositions such as the that of SR models. What we give here is an abstract semantics [13] for concurrent composition of sequential processes. Our semantics focuses on the control behavior, and hence complements a semantic that focuses on concurrency, such as SR [8] or KPN [11].

2 Continuation Actor

A *Continuation Actor* describes a process that has a set of programmatic *entry locations* starting from which execution can be *entered*, concluding by either *terminating*, *suspending*, or *exiting* with an *exit label*. A Continuation Actor *exiting* represents the control leaving the Continuation Actor and moving to some external location referred to by the *exit label*. A Continuation Actor *terminating* represents the end of control flow, whereas a continuation *suspending* denotes a pause taken in control flow, yielding control to a containing model or an execution environment. A suspended Continuation Actor can be *resumed*, which can be thought of as *entering* with a special, relative entry location that is set internally to the location at which the Continuation Actor was last suspended. Finally, continuations can be *initialized*, which too can be thought of as a special entry location.

2.1 Continuation Actor Semantics

Formally, a *Continuation Actor* **C** is defined by the tuple

$$\mathbf{C} = (\mathbb{I}, \mathbb{O}, \mathbb{S}, s_0, \mathcal{L}, \mathcal{G}, \mathbf{enter}, \mathbf{fire}, \mathbf{postfire}) \tag{1}$$

similar to an *Actor* in *Modular Actor Interface* semantics [16]. The first three types represent the input $\mathbb{I}$, output $\mathbb{O}$, and state $\mathbb{S}$ of the Continuation Actor, with the initial state $s_0 \in \mathbb{S}$. The subsequent two components, $\mathcal{L}$ and $\mathcal{G}$, are finite sets containing *entry locations* and *exit labels.* Together, these six components specify the static semantics of the Continuation Actor. With the addition of special elements to $\mathcal{L}$ and $\mathcal{G}$, entry and exit *control actions* for **C** are defined as follows

$$\mathbb{L} = \mathcal{L} + \mathit{initialize}_u + \mathit{resume}_u \tag{2}$$

$$\mathbb{G} = \mathcal{G} + \mathit{terminate}_u + \mathit{suspend}_u \tag{3}$$

where T_u is the singleton type containing T and $+$ is a disjoint union. Actions *initialize* and *resume* in $\mathbb{L}$ denote the initialization and resumption of a Continuation Actor, whereas actions in $\mathcal{L}$ denote entrance of a Continuation Actor at the corresponding location. Similarly, *terminate* and *suspend* denote the result of a Continuation Actor terminating and suspending. Actions in $\mathcal{G}$ denote exiting a Continuation Actor via the corresponding exit labels.

The last three components form the *interface* of a Continuation Actor, and define its dynamic semantics. They have the following types:

$$\mathbf{enter} : \mathbb{S} \times \mathbb{I} \times \mathbb{L} \to \mathbb{G} \tag{4}$$

$$\mathbf{fire} : \mathbb{S} \times \mathbb{I} \times \mathbb{L} \to \mathbb{O} \tag{5}$$

$$\mathbf{postfire} : \mathbb{S} \times \mathbb{I} \times \mathbb{L} \to \mathbb{S} \tag{6}$$

The **fire** and **postfire** function are similar to those in [16] and [14], only differing in their additional input of an entry action. The **fire** function specifies the outputs produced by the Continuation Actor with the given state, input, and entry action. The **postfire** likewise specifies the change in state consequent the execution from a given entry. The **enter** function specifies the control behavior of the Continuation Actor, and is the extension of the interface beyond that of an *Actor* [16]. In particular, this function specifies the concluding *control decision* made by the execution in the form of an exit action.

The role these interface functions play in execution depends on the model of computation in which the Continuation Actor is contained. In the case of an SR model, for instance, a typical execution is constituted of a series of discrete steps. In each step n, there will be several iterations, indexed by k, computing a least fixed point of the relation determined by the contained elements. A Continuation Actor would, in a particular state $s_n \in \mathbb{S}$, be *entered* and *fired* for each iteration with a particular entry action l_n^k and input value i_n^k, producing a exit action g_n^k and output value o_n^k. The Continuation Actor would then be *postfired* at the end of the step updating its internal state from s_n to s_{n+1} in terms of the final values for the input and entry action, denoted i_n^M and l_n^M. Such an execution would fulfill the relations

$$g_n^k = \mathbf{enter}(s_n,\, i_n^k,\, l_n^k) \tag{7}$$

$$o_n^k = \mathbf{fire}(s_n,\, i_n^k,\, l_n^k) \tag{8}$$

$$s_{n+1} = \mathbf{postfire}(s_n,\, i_n^M,\, l_n^M) \tag{9}$$

Note that the state is not superscripted by an iterative step since it is maintained over iterations of a fixed point computation. Later, the semantics of the particular case of a Coroutine Model will be described in detail.

2.2 Non-strict Continuation Actors

In certain models of computation, input and output values can be partially known during execution. Examples of this include Synchronous Reactive models [8] and models in synchronous languages such as Constructive Esterel [3] and SyncCharts [1]. The input and output types of components in these models are extended to represent this partial information by being lifted into pointed Complete Partial Orders (pCPOs), which are partially ordered sets containing a bottom element $\perp$ and the least upper bound of each chain.

In the case where partial information simply means that a variable may either be known to have a particular value or not known, the corresponding pCPO often used is constructed by adjoining a bottom element to the set of values associated with the type of the variable. In this case, all particular values are incomparable in the order, and all greater than the adjoined bottom element. This pCPO is known as a flat CPO. For tuples of variables, which often characterize input and output spaces, the corresponding pCPOs are typically the pointwise products of the flat CPOs for each constituting variable. Nevertheless, for generality it is not assumed that any of these particular pCPOs is used.

Given that the spaces $\mathbb{I}$ and $\mathbb{O}$ are lifted into pCPOs a Continuation Actor can be specified on these lifted types. Consequently, the **fire** function can be defined so that partial information about the outputs can be determined from partial information about the inputs. A function is known as *strict* if it maps all input values that are not maximal in the input pCPO to bottom. A function is otherwise *non-strict*, and intuitively can be understood as able to determine some information about the output without total information about the input. Non-strict functions play an important role in models of computation such SR [8] where constructive methods are used to iteratively determine consistent valuations of input and output variables which can have cyclic dependencies. Note that the state here is not lifted into a pCPO, and thus the **postfire** function has no non-strict version analogous to that of **fire**. The **enter** function can similarly extended to operate over partial information about inputs producing partial information about exit actions in $\mathbb{G}$. These partial control choices can be represented as sets of possible exit actions given the partial information about the input. Hence the **enter** function in such a Continuation Actor has a lifted codomain of type $2^{\mathbb{G}}$.

If this representation of partial information about control actions, $2^{\mathbb{G}}$, is ordered by reverse-inclusion, where

$$a \leq b \;=\; a \supseteq b$$

a pCPO is formed with $\bot$ being the whole set $\mathbb{G}$. The motivation behind this ordering is that a strict increase in this order corresponds to making more specific control decisions, with the singleton elements representing a unique and thus total decision. Monotonicity of the **enter** function, as required in certain domains with constructive semantics, therefore corresponds to the requirement that

$$\forall\, s \in \mathbb{S},\, l \in \mathcal{L} \bullet a \leq b \;\Rightarrow\; \mathbf{enter}(s,\, a,\, l) \supseteq \mathbf{enter}(s,\, b,\, l)$$

Intuitively, this monotonicity property means that as more is known about the input, the control choices at the least do not become less known, and may become more known. In particular, for the **enter** function this means that as more is known about the input additional control choices can never be added, and some may be removed potentially narrowing down the control behavior to a single choice.

2.3 Counter Example

An example of a Continuation Actor is a **Counter** that increments an internal state s_c each time it is resumed, and subsequently suspends. This **Counter** also has a threshold stored in an internal state s_t, which can be set by an input i_t. If the **Counter** is resumed and $s_c \geq s_t$, instead of suspending the **Counter** exits with exit label g_t. Let this **Counter** also have an output o_c that is set to the current count during each execution. In order to set the value of the threshold s_t to input i_t, suppose there is also an explicit entry location l_t at which s_t is set before performing the resume action. Assume that i_t, s_c, s_t, and o_c are all natural numbers (of type $\mathbb{N}$).

This **Counter** can defined formally as follows. Let the static semantics be

$$\mathbf{Counter} \;=\; (\overbrace{\underbrace{\mathbb{N}}_{i_t}}^{\mathbb{I}},\, \overbrace{\underbrace{\mathbb{N}}_{o_c}}^{\mathbb{O}},\, \overbrace{\underbrace{\mathcal{L} \times \mathbb{N} \times \mathbb{N}}_{(s_l,\, s_c,\, s_t)}}^{\mathbb{S}},\, \overbrace{(l_0,\, 0,\, 0)}^{s_0},\, \overbrace{\{l_t,\, l_0,\, l_1\}}^{\mathcal{L}},\, \overbrace{\{g_t\}}^{\mathcal{G}})$$

The state here is a triple $(s_l,\, s_c,\, s_t) \in \mathcal{L} \times \mathbb{N} \times \mathbb{N}$, where $s_l \in \mathcal{L}$ holds the entry location to resume at after a suspension, $s_c \in \mathbb{N}$ is the current counter value, and $s_t \in \mathbb{N}$ is the current threshold value. In addition to the entry location l_t, which sets the threshold, there are two internal entry locations l_0 and l_1. s_l is set to l_0 initially, and in this state the counter is reset under a resumption, but upon the completion of any entry s_l is set by the **postfire** function to l_1, indicating that the **Counter** is counting when it is resumed. The interface functions are then

$$\mathbf{enter}((s_l,\, s_c,\, s_t),\, i_t,\, l) \;=\; \begin{cases} \mathit{suspend} & l = l_0 \text{ or } \mathit{initialize} \\ \mathbf{enter}((s_l,\, s_c,\, s_t),\, i_t,\, s_l) & l = \mathit{resume} \\ \textbf{if } s_c \geq s_t \textbf{ then } g_t \textbf{ else } \mathit{suspend} & l = l_1 \\ \mathbf{enter}((s_l,\, s_c,\, i_t),\, i_t,\, l_r) & l = l_t \end{cases}$$

$$\mathbf{fire}((s_l,\, s_c,\, s_t),\, i_t,\, l) \;=\; \begin{cases} 0 & l = l_0 \text{ or } \mathit{initialize} \\ \mathbf{fire}((s_l,\, s_c,\, s_t),\, i,\, s_l) & l = \mathit{resume} \\ s_c + 1 & l = l_1 \text{ or } l_t \end{cases}$$

$$\mathbf{postfire}((s_l,\, s_c,\, s_t),\, i_t,\, l) \;=\; \begin{cases} (l_1,\, 0,\, s_t) & l = l_0 \text{ or } \mathit{initialize} \\ \mathbf{postfire}((s_l,\, s_c,\, s_t),\, i_t,\, s_l) & l = \mathit{resume} \\ (l_1,\, s_c + 1,\, s_t) & l = l_1 \\ \mathbf{postfire}((s_l,\, s_c,\, i_t),\, i_t,\, l_r) & l = l_t \end{cases}$$

Note that here there is a difference between internal location l_1 and entry action *resume*, and likewise between l_0 and *initialize*, and that these cannot be conflated. If the **Counter** were entered with *resume* in its initial state, it would be map to l_0 rather than l_1. Although *initialize* is always the same case as l_0, *initialize* is maintained as separate as a matter of satisfying the interface obligation of providing such an entry action.

2.4 State Example

Another example of a Continuation Actor would be one that represents a state in a state machine, where the state evaluates outgoing guard expressions as part of its **enter** function and performs corresponding transition actions as part of its **fire** function. Let this formulation of a state, called **StateCA**, be parameterized by a finite set of transitions $\mathcal{T}$ and a default action $q_{\mathbf{def}}$. Each transition $\tau_k \in \mathcal{T}$ is defined by the tuple $\tau_k = (p_k,\, q_k,\, g_k)$, where $p_k : \mathbb{I} \rightarrow 2$ are transition predicates, $q_k : \mathbb{I} \rightarrow \mathbb{O}$ are transition actions ($q_{\mathbf{def}}$ is of the same type), and g_k are exit labels, referring to the remote destination of control upon taking the corresponding transition. Let π_p, π_q, and π_g be the projection functions for these components.

Given these parameters, characterizing the local behavior of the state, such a **StateCA** A can be given the following static semantics:

$$A(\mathcal{T},\, q_{\mathbf{def}}) = (\mathbb{I},\, \mathbb{O},\, \overbrace{\mathbf{1}}^{\mathbb{S}},\, \overbrace{\mathbf{u}}^{s_0},\, \overbrace{\emptyset}^{\mathcal{L}},\, \overbrace{\{\pi_g(\tau) \,|\, \tau \in \mathcal{T}\}}^{\mathcal{G}})$$

Here, there are no explicit entry locations and the exit labels for A are the locations $\pi_g(\tau_k)$ corresponding to each transition τ_k. There is only one state, denoted $\mathbf{u}$. In addition to the given transitions let the set of transitions be adjoined with an additional default transition defined

$$\tau_{\mathbf{def}} = (\forall\, \tau \in \mathcal{T} \bullet \neg\pi_p(\tau),\, q_{\mathbf{def}},\, \mathit{suspend})$$

to form $\mathcal{T}'$. This predicate of this default transition is true if those of all other transitions are false, the action is the given default action, and instead of an exit label the third component denotes suspension. Assume that there also exists a function $\mathbf{choose}_{\mathcal{T}'}$ of type $\mathbb{I} \to \mathcal{T}'$ that, given an input, chooses a transition τ for which the predicate $\pi_p(\tau)(i)$ is true.[1] If none are true, let it return default transition. The interface functions for A are simply:

$$\begin{aligned} \mathbf{enter}(s, i, l) &= \pi_g(\mathbf{choose}_{\mathcal{T}'}(i)) \\ \mathbf{fire}(s, i, l) &= \pi_q(\mathbf{choose}_{\mathcal{T}'}(i))(i) \\ \mathbf{postfire}(s, i, l) &= \mathbf{u} \end{aligned}$$

3 The Coroutine Model of Computation

Models in the *Coroutine* Model of Computation describe networks of *Continuation Actors*, connected to each other such that the exit labels of one Continuation Actor refer to entry locations of another. The referent can either be an explicit entry location, the *initialize* action, or the *resume* action on a target Continuation Actor. Given this structure, when one Continuation Actor in the network *exits*, control can proceed to another Continuation Actor following these connections. Furthermore, when a Continuation Actor *suspends* or *terminates* during execution the containing Coroutine Model does as well. An execution of a Coroutine Model is thus a sequence of executions of the contained Continuation Actors forming a control path through the structure and terminating with either suspension or termination. The Coroutine Model is also given its own entry locations and exit labels that can connect internally to the respective exit labels and entry locations of its contained Continuation Actors.

In this manner, a Coroutine Model can also be *entered* by entering one of its entry locations, as well as be *resumed* by resuming the Continuation Actor in which the execution of the model had been previously suspended. The model can also be *initialized* by initializing a particular initial Continuation Actor. It can *exit* with one of its exit labels, and also *suspend* or *terminate* if one of its contained Continuation Actors does. It follows that a Coroutine Model is itself a Continuation Actor. This compositionality property allows for Coroutine Models to form hierarchies, and likewise for specified Continuation Actors to be built out of other Continuation Actors.

3.1 Coroutine Models

Formally, a *Coroutine Model* $\mathcal{M}$ is described by the following tuple

$$\mathcal{M} = (\mathbf{Q}, q_0, m_{\mathbb{I}}, m_{\mathbb{O}}, \oplus, \kappa, \eta) \tag{10}$$

Here, $\mathbf{Q}$ is a finite set of Continuation Actors that constitute the model, and $q_0 \in \mathbf{Q}$ is an initial Continuation Actor. The two components $m_{\mathbb{I}}$ and $m_{\mathbb{O}}$ map

[1] This allows for the possibility that the predicates are not mutually exclusive in which case **choose** determines a means to select a unique transition.

between the inputs and outputs of the whole model and those specific inputs and outputs of particular Continuation Actors in $\mathbf{Q}$.

Let $\mathbb{I}_{\mathcal{M}}$ and $\mathbb{O}_{\mathcal{M}}$ be the input and output types of $\mathcal{M}$. The input and output maps then have the following types:

$$m_{\mathbb{I}} : \Pi\, q \in \mathbf{Q} \bullet \mathbb{I}_{\mathcal{M}} \to \mathbb{I}_q \tag{11}$$

$$m_{\mathbb{O}} : \Pi\, q \in \mathbf{Q} \bullet \mathbb{O}_q \to \mathbb{O}_{\mathcal{M}} \tag{12}$$

where the operator Π here denotes a dependent type product, and the types $\mathbb{I}_q$ and $\mathbb{O}_q$ denote the input and output types for Continuation Actor q. By composition with $m_{\mathbb{I}}$ and $m_{\mathbb{O}}$, the input and output types of each Continuation Actor are made identical. The binary operator $\oplus : \mathbb{O}_{\mathcal{M}} \times \mathbb{O}_{\mathcal{M}} \to \mathbb{O}_{\mathcal{M}}$ is then used to combine the mapped output values produced by different Continuation Actors.

Let two sets of internal entry locations and exit labels be defined for the model

$$\mathcal{L} \;=\; \Sigma\, q \in \mathbf{Q} \bullet \mathbb{L}_q \tag{13}$$

$$\mathcal{G} \;=\; \Sigma\, q \in \mathbf{Q} \bullet \mathcal{G}_q \tag{14}$$

where the operator Σ here denotes a dependent type sum. In other words, members of $\mathcal{L}$ are of the form $(q,\, x)$ where $q \in \mathbf{Q}$ and $x \in \mathbb{L}_q$, and likewise for $\mathcal{G}$ with respect to $\mathcal{G}_q$. Let the finite sets $\mathcal{L}_{\mathcal{M}}$ and $\mathcal{G}_{\mathcal{M}}$ be the entry locations and exit labels of the model, distinct from their internal counterparts.

The functions κ and η give the structure to the model. The former maps locations in $\mathcal{L}_{\mathcal{M}}$ to internal locations in $\mathcal{L}$. The latter maps each exit labels of each Continuation Actor to either entry actions of another, including to the *initialize* and *resume* special locations, or to exit labels in $\mathcal{G}_{\mathcal{M}}$. They can therefore be given the following types:

$$\kappa : \mathcal{L}_{\mathcal{M}} \to \mathcal{L} \tag{15}$$

$$\eta : \mathcal{G} \to \mathcal{L} + \mathcal{G}_{\mathcal{M}} \tag{16}$$

When the conclusion of the execution of q is to exit with exit label g, and $(q',\, k) = \eta(q,\, g)$, control proceeds with entry action k performed on Continuation Actor q'. When instead $\eta(q,\, g) \in \mathcal{G}_{\mathcal{M}}$, control exits the model.

The state space of model $\mathcal{M}$ is constructed from a product of the state spaces of the Continuation Actors in $\mathbf{Q}$ along with the internal entry location corresponding to the entry action to be taken when the model resumes from a suspension. That is

$$\mathbb{S}_{\mathcal{M}} = \mathcal{L} \times \prod_{q \in \mathbf{Q}} \mathbb{S}_q \tag{17}$$

Correspondingly, the initial state of $\mathcal{M}$ is

$$s_{0\mathcal{M}} = ((q_0,\, \mathit{initialize}),\, s_{0\,q_1},\, \ldots,\, s_{0\,q_n}),\ \text{ where } q_k \in \mathbf{Q}, 1 \le k \le n \tag{18}$$

so that calling *resume* on model in its initial state has the effect of initializing q_0.

The above, in total, give the static semantics for Coroutine Model $\mathcal{M}$ as a Continuation Actor:

$$\mathbf{C}_{\mathcal{M}} = (\mathbb{I}_{\mathcal{M}},\, \mathbb{O}_{\mathcal{M}},\, \mathbb{S}_{\mathcal{M}},\, s_{0\mathcal{M}},\, \mathcal{L}_{\mathcal{M}},\, \mathcal{G}_{\mathcal{M}}) \tag{19}$$

3.2 Strict Dynamic Semantics

For a *Coroutine Model* $\mathcal{M}$, specified as in (19), **enter**, **fire**, and **postfire** functions can be defined compositionally, in terms of the specifications and corresponding interfaces of the contained Continuation Actors in **Q**. The definitions of these functions constitute a denotational dynamic semantics for Coroutine Models as Continuation Actors.

In order to describe the traversal of control in the model, the interface functions are augmented using the input and output maps to functions that have types corresponding to the model:

$$\mathbf{enter}_U(s,\, i,\, (q,\, l)) \;=\; (q,\, \mathbf{enter}_q(s_q,\, m_{\mathbb{I}}(q,\, i),\, l)) \tag{20}$$

$$\mathbf{fire}_U(s,\, i,\, (q,\, l)) \;=\; m_{\mathbb{O}}(q,\, \mathbf{fire}_q(s_q,\, m_{\mathbb{I}}(q,\, i),\, l)) \tag{21}$$

$$\mathbf{postfire}_U(s,\, i,\, (q,\, l)) \;=\; r_q(s,\, \mathbf{postfire}_q(s_q,\, m_{\mathbb{I}}(q,\, i),\, l)) \tag{22}$$

where the function $r_q(s,\, v)$ replaces the element in s corresponding to q with value v.

The process of traversing a control path through the model, following exit labels of Continuation Actors to entry locations of subsequent Continuation Actors, ultimately reaching suspension, termination, or the exiting of the model, is described by the $\mathbf{enter}_U$ function in conjunction with the structural map η. In order to put these two pieces together, first it should be noted that the type of $\mathbf{enter}_U$ function is

$$\mathbf{enter}_U : \mathbb{S}_{\mathcal{M}} \times \mathbb{I}_{\mathcal{M}} \times \mathcal{L} \to \mathcal{G} + \mathcal{Z} \tag{23}$$

$$\text{where } \mathcal{Z} \;=\; Q \times (\mathit{suspend}_u + \mathit{terminate}_u)$$

When the image of $\mathbf{enter}_U$ is in $\mathcal{G}$, control then can continue to another Continuation Actor determined by η, whereas if the image is in $\mathcal{Z}$ the control ends in the model with a suspension or termination. To connect this with η, an augmented version of the function is defined as follows:

$$\nu : \mathcal{G} + Q \times \{\mathit{terminate},\, \mathit{suspend}\} \to \mathcal{L} + \mathcal{G}_{\mathcal{M}} + \mathcal{Z} \tag{24}$$

$$\nu(g) = \begin{cases} \eta(g) & g \in \mathcal{G} \\ g & g \in \mathcal{Z} \end{cases} \tag{25}$$

This function can then be composed with $\mathbf{enter}_U$ to form the *traversal function*, which describes the control traversal through the model:

$$\epsilon : \mathbb{S} \times \mathbb{I}_{\mathcal{M}} \times \mathcal{L} \to \mathcal{L} + \mathcal{G}_{\mathcal{M}} + \mathcal{Z} \tag{26}$$

$$\epsilon(s,\, i) \;=\; \nu \circ \mathbf{enter}_U(s,\, i) \tag{27}$$

Since $\mathcal{L}$ is in both the domain and codomain of ϵ, it can be iterated over, starting with an initial location l_0, forming a series

$$(l_0,\, \epsilon(s,\, i)(l_0),\, \epsilon(s,\, i)^2(l_0),\, \ldots)$$

possibly ending with a terminating value in either $\mathcal{G}_{\mathcal{M}}$ or $\mathcal{Z}$. This series generated by ϵ is the *control path* of the model for state s and input i, generated by location l_0.[2]

In addition to defining the traversal function with η, the map κ can be augmented to handle the whole set of entry actions $\mathbb{L}_{\mathcal{M}}$. To this end, let this augmentation be defined

$$\theta : \mathbb{S}_{\mathcal{M}} \times \mathbb{L}_{\mathcal{M}} \to \mathcal{L} \tag{28}$$

$$\theta(s,\, h) = \begin{cases} (q_0,\, \mathit{initialize}) & h \in \mathit{initialize}_u \\ (s_{\mathcal{L}},\, \mathit{resume}) & h \in \mathit{resume}_u \\ \kappa(h) & h \in \mathcal{L}_{\mathcal{M}} \end{cases} \tag{29}$$

$$\mathbf{enter}(s,\, i,\, h) = \mathbf{e}(s,\, i,\, \theta(s,\, h)) \tag{30}$$

$$\mathbf{e}(s,\, i,\, l) = \begin{cases} z, \text{ where } (q,\, z) = l & l \in \mathcal{Z} \\ l & l \in \mathcal{G}_{\mathcal{M}} \\ \mathbf{e}(s,\, i,\, \epsilon(s,\, i,\, l)) & l \in \mathcal{L} \end{cases} \tag{31}$$

$$\mathbf{fire}(s,\, i,\, h) = \mathbf{f}(s,\, i,\, \theta(s,\, h),\, 0_{\oplus}) \tag{32}$$

$$\mathbf{f}(s,\, i,\, l,\, o) = \begin{cases} o & l \in \mathcal{Z} + \mathcal{G}_{\mathcal{M}} \\ \mathbf{f}(s,\, i,\, \epsilon(s,\, i,\, l),\, o \oplus \mathbf{fire}_U(s,\, i,\, l)) & l \in \mathcal{L} \end{cases} \tag{33}$$

$$\mathbf{postfire}(s,\, i,\, h) = \mathbf{p}(s,\, i,\, \theta(s,\, h)) \tag{34}$$

$$\mathbf{p}(s,\, i,\, l) = \begin{cases} r_{\mathcal{L}}(s,\, l) & l \in \mathcal{Z} \\ r_{\mathcal{L}}(s,\, (q_0,\, \mathit{initialize})) & l \in \mathcal{G}_{\mathcal{M}} \\ \mathbf{p}(\mathbf{postfire}_U(s,\, i,\, l),\, i,\, \epsilon(s,\, i,\, l)) & l \in \mathcal{L} \end{cases} \tag{35}$$

Fig. 1. Strict dynamic semantics for Coroutine Models

The **enter**, **fire**, and **postfire** functions for a coroutine model can then be recursively defined as in figure 1, where the functions **e**, **f**, and **p** are the recursive kernels of the respective **enter**, **fire**, and **postfire**. Here, the **enter** function simply follows the control path of the traversal function. The **fire** function makes

[2] If this path has no terminating value, it is possible that iterating ϵ can diverge. Depending on the context, this can either be left as a possibility or restricted in some fashion as for instance is done in Esterel **loop** constructs in [3].

the same traversal, but accumulates outputs from the firing of each Continuation Actor with $\oplus$. The **postfire** function similarly traverse the control path, updating the state of each Continuation Actor along the path.

It should be noted that the state update over the traversal is independent from the traversal itself and the firings, hence it represents a set of changes that are only committed to after the traversal and output values are established. Nevertheless, if the model is suspended and resumed, the state changes take effect when it is resumed. Amongst these state changes is certainly, in particular, the change in the location at which to resume.

3.3 Non-strict Semantics

In a Coroutine Model where the constituting Continuation Actors are defined over pCPOs, representing partial information about inputs, outputs, and control decisions, a non-strict semantics can be given. Rather than determining a single control path through the Continuation Actors, given partial input information several control actions may be possible at each Continuation Actor, thereby generating instead a *tree* of control paths. If the **enter** function of each Continuation Actor is monotonic, then as the input information becomes more specific the control choices at each Continuation Actor in the tree become no greater, and possibly fewer, thereby pruning the *control tree* (or at least making it no larger).

Given the **enter** function for a Continuation Actor defined over pCPOs has a codomain of $2^{\mathcal{G}_q}$, the correspondingly lifted version of $\mathbf{enter}_U$ is defined:

$$\mathbf{enter}_U(s,\, i,\, (q,\, l)) \;=\; \{(q,\, g) \,|\, g \in \mathbf{enter}_q(s_q,\, m_{\mathbb{I}}(q,\, i),\, l)\} \tag{36}$$

The $\mathbf{fire}_U$ function on the other hand has essentially the same definition. Given this change in $\mathbf{enter}_U$, the function η can also be lifted:

$$\tilde{\nu}(G) \;=\; \{\eta(g) \,|\, g \in G \cap \mathcal{G}\} \cup (G \cap (\mathcal{G}_{\mathcal{M}} + \mathcal{Z}_{\mathcal{M}})) \tag{37}$$

Combining these two parts, a non-strict traversal function $\hat{\epsilon}$ can then be defined

$$\hat{\epsilon} : \mathbb{S} \times \mathbb{I}_{\mathcal{M}} \times \mathcal{L} \to 2^{\mathcal{L}+\mathcal{G}_{\mathcal{M}}+\mathcal{Z}} \tag{38}$$

$$\hat{\epsilon}(s,\, i) \;=\; \tilde{\nu} \circ \mathbf{e}(s,\, i) \tag{39}$$

Hence, for a given state and input, $\hat{\epsilon}$ maps a location to a set of successor locations potentially including terminal locations in $\mathcal{G}_{\mathcal{M}}$ or $\mathcal{Z}$. A control tree is thereby generated. If iterated over, $\hat{\epsilon}$ generates a tree of entry locations with terminations or suspensions as leaves. It is worth noting that the codomain of $\hat{\epsilon}$ can also be expressed as $2^{\mathcal{L}} \times 2^{\mathcal{G}_{\mathcal{M}}} \times 2^{\mathcal{Z}}$, and thus the image of $\hat{\epsilon}$ can always be decomposed into these three sets denoting the possible control choices within each of their respective categories.

Non-strict versions of the **enter** and **fire** functions for the coroutine model can then be defined in terms of their kernels **e** and **f** as in figure 2. The form

$$\mathbf{e}(s,\,i,\,l) \;=\; \begin{cases} \{z\}, \text{ where } (q,\,z) = l & l \in \mathcal{Z} \\ \{l\} & l \in \mathcal{G}_{\mathcal{M}} \\ \displaystyle\bigcup_{l' \in \hat{e}(s,\,i,\,l)} \mathbf{e}(s,\,i,\,l') & l \in \mathcal{L} \end{cases} \tag{40}$$

$$\mathbf{f}(s,\,i,\,l,\,o) \;=\; \begin{cases} o & l \in \mathcal{Z} + \mathcal{G}_{\mathcal{M}} \\ \displaystyle\prod_{l' \in \hat{e}(s,\,i,\,l)} \mathbf{f}(s,\,i,\,l',\,o \oplus \mathbf{fire}_U(s,\,i,\,l)) & l \in \mathcal{L} \end{cases} \tag{41}$$

Fig. 2. Nonstrict dynamic semantics for Coroutine Models

of both definitions is similar, in that the control tree in each is traversed recursively building a collection of results for all control paths. These results are then combined by an operation to form the greatest consistent conclusion that can drawn across all of them. In the non-strict **enter**, the sets of final control decisions for each path are combined with a union to conservatively give a set of all reachable control decisions for the model. In the **fire** function, an output is computed along each path, and the outputs for all paths are combined with a greatest lower bound. The resulting partial output consists of only the consistent information across all possible paths.

Given this non-strict characterization of Coroutine Model semantics, Coroutine Models can be given a clear denotational meaning in the context of fixed-point semantics. Synchronous compositions of Coroutine Models can therefore be constructed within semantics such as those of SR [8]. Important in such synchronous models is the property of monotonicity, which can be reasoned about in a clear way with the above denotational semantics. In order to perform this kind of domain-theoretic reasoning about the above semantic equations it must be determined that these equations have clear domain-theoretic solutions. In fact, this is the case, and the following can be proven:

Theorem 1. *Given a non-strict Coroutine Model $\mathcal{M}$, if the input $\mathbb{I}_{\mathcal{M}}$ and output $\mathbb{O}_{\mathcal{M}}$ types of the model are finite-height pCPOs and operator $\oplus$ is monotonic, then the above recursive equations characterizing the kernel functions* **e** *and* **f** *have unique least fixed-point solutions in the partial order of functions with codomains $2^{\mathbb{G}}$ and $\mathbb{O}_M$, respectively.*

Since solutions to the equations for the recursive kernels exist, then it can further be asked if under the right conditions **enter** and **fire** are monotonic functions from $\mathbb{I}_{\mathcal{M}}$ to $2^{\mathbb{G}}$ and $\mathbb{O}_M$, respectively. In fact these functions are monotonic, and more specifically continuous, under the conditions described in the following theorem:

Theorem 2. *Given a non-strict Coroutine Model $\mathcal{M}$, if the input $\mathbb{I}_{\mathcal{M}}$ and output $\mathbb{O}_{\mathcal{M}}$ types of the model are finite-height pCPOs and operator $\oplus$ is monotonic,* *and if for each $q \in Q$ the functions* $\mathbf{enter}_q$ *and* $\mathbf{fire}_q$ *are monotonic in terms of* $\mathbb{I}_q$*, and the mapping functions* $m_{\mathbb{I}}$ *and* $m_{\mathbb{O}}$ *are monotonic, then the non-strict kernels* $\mathbf{e}$ *and* $\mathbf{f}$ *are continuous in terms of* $\mathbb{I}_{\mathcal{M}}$*.*

It follows that **enter** and **fire** defined in terms of these non-strict kernels are both continuous, and thus monotonic as well. The proof of this fact involves noting that the union operator is the greatest lower bound under the order of reverse inclusion. Both definitions then are formally similar and can be altered in simple ways to get the same general formula for both. This general formula, taking the greatest lower bound of every branch formed by the traversal, can be shown to be monotonic because an increase in the codomain of the traversal function, ordered by reverse inclusion, corresponds to there being fewer branches, and thus fewer possible control paths. Furthermore, the greatest lower bound of a set of decreasing size always corresponds to the value of this bound remaining equal or increasing. That is

$$A \sqsubseteq B \;\Leftrightarrow\; A \supseteq B \;\Rightarrow\; \bigsqcap A \sqsubseteq \bigsqcap B \tag{42}$$

The proof follows from working out the details of this general relationship. The most important consequence of this theorem is that, under the above conditions, monotonicity is compositional for Coroutine Models.[3]

4 Related Work

The semantics of the component-based model *42* defined by Maraninchi [15] also gives an atomic interface description for *components* that includes control along with data, but the aims of the control dimension are very different. The control ports of a component in *42* receive tokens from a model *controller*, whereas the entry locations and exit labels of Continuation Actors are meant to form a network of control relationships. Since the control behavior of a *42* model is specified by its *controller*, which can be any imperative program, *42* by itself does not constitute any particular dynamic semantics. Moreover, there is no particular way in the interface semantics of *42* to handle non-strict control decisions.

A denotational semantics for Stateflow is given by Hamon [9] in which states are represented as continuations. However, the denotations given are functional programs relevant particularly in the context of understanding compilation. Since these functional programs act on both data and continuation environments, there is no clear way give this formalism a non-strict interpretation or compose it with other models that do not involve its environments. Although Hamon's semantics provide a backtracking mechanism, partial information cannot be combined from several potential paths as it can in the non-strict semantics given here. Finally, Hamon's model treats transition guards and actions as a part of the semantics of the execution. Here, in contrast, the role of guards and

[3] The full proof of the above two theorems is available in the appendix of: `http://chess.eecs.berkeley.edu/pubs/902.html`.

transition actions are considered to be part of the Continuation Actors. The semantics of Coroutine Models is thereby considerably simpler and applicable to a wider set of cases where different guard languages or other mechanisms are used to determine control transitions.

Another denotational formalism for state machines is given by Broy, et al. [7] as part of a general denotational semantics for UML. Although the denotational nature of this formalism lends itself to reasoning about composition, the formalism requires a complex state mechanism involving a frame stack and an event-driven model of behavior. The state machine semantics given does not provide a mean to articulate sequences of immediate transitions within a computational step, as opposed to transitions that happen in separate steps. Moreover, no investigation is made in this work of how to deal with non-strict state machines. One should note that the monotonicity properties discussed by Broy, et al., are not with respect to the state machines themselves being non-strict, in the manner this paper discusses, but instead regarding the causal properties of the event passing system cast into the formulation of streams.

5 Conclusion

The *Coroutine Model of Computation* defined here provides a general denotational model for representing control-oriented behavior, capable of use in hierarchical and heterogeneous systems. Both a strict and a non-strict denotational semantics have been given for Coroutine models allowing the compositional analysis between these models, and models with other semantics, to be expressed in functional terms. In particular, the non-strict semantics enable such models to be used in synchronous compositions with clear conditions for monotonicity. Coroutine Models also fit the definition of a *Director* given by Tripakis et al.[16] as a function from the interfaces of the constituting actors and structure of the model to an actor representation of the composite model. Given this language, many control-oriented models can be expressed in its terms by defining a set of constituting Continuation Actors and potentially making small modifications to the model semantics. Some preliminary work has thus far been done for fully modeling the semantics of SyncCharts [1] in terms of Coroutine Models. Work has also been done to implement the Coroutine Model of Computation in the **Ptolemy II** environment, where it can be used to develop and test executable heterogeneous models.

References

1. André, C.: SyncCharts: A visual representation of reactive behaviors. Rapport de recherche tr95-52, Université de Nice-Sophia Antipolis (1995)
2. André, C.: Computing syncharts reactions. Electronic Notes in Theoretical Computer Science 88(33), 3–19 (2004)
3. Berry, G.: The Constructive Semantics of Pure Esterel. In: Program (1999)
4. Boussinot, F., de Simone, R.: The ESTEREL language. Proceedings of the IEEE 79(9), 1293–1304 (1991)

5. Boussinot, F.: Reactive C: An extension of C to program reactive systems. Software: Practice and Experience 21(4), 401–428 (1991)
6. Broy, M.: Modelling Operating System Structures by Timed Stream Processing Functions. Journal of Functional Programming, 1–26 (1992)
7. Broy, M., Cengarle, M.V., Rumpe, B.: Semantics of UML, Towards a System Model for UML, Part 3: The State Machine Model. Technical report, Technische Universität München (2007)
8. Edwards, S.A.: The Specification and Execution of Heterogeneous Synchronous Reactive Systems. Technical report, University of California Berkeley (1997)
9. Hamon, G.: A denotational semantics for Stateflow. In: Proceedings of the 5th ACM International Conference, pp. 164–172 (2005)
10. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8(3), 231–274 (1987)
11. Kahn, G., MacQueen, D., et al.: Coroutines and networks of parallel processes (1976)
12. Lee, E.A.: The semantics of dataflow with firing. Semantics to Computer Science 0720882(c), 1–20 (2008)
13. Lee, E.A., Neuendorffer, S.: Actor-oriented design of embedded hardware and software systems. Journal of Circuits Systems 12(3), 231–260 (2003)
14. Lee, E.A., Tripakis, S.: Modal models in Ptolemy. In: 3rd International Workshop on Equation-Based Object-Oriented Modeling Languages and Tools (EOOLT), vol. 47, pp. 11–21 (2010)
15. Maraninchi, F., Bouhadiba, T.: 42: Programmable models of computation for a component-based approach to heterogeneous embedded systems. In: Proceedings of the 6th International Conference on Generative Programming and Component Engineering, pp. 53–62. ACM (2007)
16. Tripakis, S., Stergiou, C., Shaver, C., Lee, E.A.: A Modular Formal Semantics for Ptolemy. Mathematical Structures in Computer Science (to appear, 2012)

Assume-Guarantee Scenarios: Semantics and Synthesis*

Shahar Maoz[1] and Yaniv Sa'ar[2]

[1] School of Computer Science, Tel Aviv University, Israel
[2] Dept. of Computer Science, The Weizmann Institute of Science, Israel

Abstract. The behavior of open reactive systems is best described in an assume-guarantee style specification: a system guarantees certain prescribed behavior provided that its environment follows certain given assumptions. Scenario-based modeling languages, such as variants of message sequence charts, have been used to specify reactive systems behavior in a visual, modular, intuitive way. However, none have yet provided full support for assume-guarantee style specifications.

In this paper we present *assume-guarantee scenarios*, which extend live sequence charts (LSC) — a visual, expressive, scenario-based language — syntax and semantics, with an explicit distinction between system and environment entities and with support not only for safety and liveness system guarantees but also for safety and liveness environment assumptions. Moreover, the semantics is defined using a reduction to GR(1), a fragment of LTL that enables game-based, symbolic, efficient synthesis of a correct-by-construction controller.

1 Introduction

It has long been recognized that the behavior of open reactive systems [11], discrete event systems that interact with their environment over time, and of other systems, is best specified using an assume-guarantee style specification: a system guarantees certain prescribed behavior provided that its environment follows certain given assumptions (see, e.g., [14,20]). Environment assumptions may be related to the laws of physics (when interacting with the physical world) or to knowledge about the behavior of external systems (when interacting with other systems). They are crucial in many application domains, because some system requirements may only be realizable under assumptions about behaviors the environment will never or will always eventually exhibit. Scenario-based languages, however, which have been used to specify reactive systems behavior in a visual, modular, intuitive way, have not yet provided full support for the assume-guarantee paradigm.

* This research was supported by The John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science. In addition, part of this research was funded by an Advanced Research Grant awarded to David Harel of the Weizmann Institute from the European Research Council (ERC) under the European Community's 7th Framework Programme (FP7/2007-2013).

One such language is live sequence charts (LSC) [5,8], a visual language for scenario-based modeling, which extends classical sequence diagrams with a distinction between mandatory-universal behavior (hot elements) and provisional-existential behavior (cold elements). While LSC allows one to specify possible and mandatory scenarios that a system should follow, and negative scenarios that a system should never allow, its current syntax and semantics do not allow one to condition the realization of these system guarantees on the fulfilment of certain behaviors of the environment. In other words, it does not support environment assumptions. This limits the expressive power of the language and its applicability to specifying real-world systems.

In this paper we present *assume-guarantee scenarios*, an extension of the LSC language with support for environment assumptions. We define the syntax and semantics of the extended language, allowing one to express safety assumptions, that is, what the environment is assumed never to do, and liveness assumptions, what the environment is assumed to always eventually do. The extension does not add external constructs to the language. Rather, it is defined by embedding assumptions implicitly in the LSCs, in keeping with the scenario-based nature of the language, just like safety and liveness system guarantees are specified in LSCs implicitly in the scenarios, using the distinction between hot and cold elements.

Moreover, we formulate the semantics of the extended language in GR(1), a fragment of linear temporal logic (LTL). The GR(1) formulation allows us to build on the game-based, symbolic, efficient synthesis algorithm of [26] and generate a correct-by-construction, executable controller. Assuming the environment adheres to the assumptions, the generated controller behavior meets the guarantees.

We have implemented our ideas using JTLV APIs [28] and integrated them into PlayGo [9]. We extended PlayGo's visual editor to support the extended language syntax, and implemented both the reduction to the GR(1) fragment and the solution of the GR(1) synthesis. The resulting controller is realized in a standalone, generated executable Java application.

We discuss related work below. Sect. 2 recalls the LSC language and presents a semi-formal overview of the assume-guarantee extension using examples. Sect. 3 recalls the GR(1) fragment of LTL, which we use as the target for the definition of the semantics of the extended language. Sect. 4 presents our main contribution: the semantics of assume-guarantee scenarios, formulated in GR(1) form. Sect. 5 presents a running example, and the second contribution of our work: synthesis of assume-guarantee scenarios. Sect. 6 describes our implementation and Sect. 7 concludes with a discussion and future work directions.

1.1 Related Work

Several scenario-based specification languages have been suggested in the literature, each with a different semantics. We discuss some of these here, focusing on the distinction between system and environment and on the ability to specify assumptions and guarantees.

Haugen et al. [13] present STAIRS, a requirements specification methodology based on UML2.0, where the semantics of interactions is given using interaction obligations. STAIRS does not distinguish between system and environment controlled objects. Thus, one may interpret its semantics to include only system guarantees and no environment assumptions.

Knapp and Wuttke [15] use UML2.0 interactions as a specification in a model-checking setup. They interpret a sequence diagram as an observer of the message exchanges and state changes in a system. Again, no distinction is made between system and environment entities / controlled messages and thus the work can be viewed as checking system guarantees, with no environment assumptions.

Whittle and Schumann [30] generate a statechart from a set of scenarios annotated with OCL constraints. The construction distinguishes messages sent by the user from messages sent by the system, and thus may be viewed as relying on implicit assumptions. However, these are only safety assumptions.

Krueger et al. [17] consider a translation of an MSC specification into a statechart. In the process, scenarios are projected onto each of the components participating in them. This may be viewed as considering each component alone to be a system and the other components as its environment. However, an explicit distinction between assumptions and guarantees is not discussed.

Additional scenario-based specification languages, such as VTS [1] and PST [2] do not explicitly distinguish between system controlled and environment controlled events, and thus do not support assumptions.

Greenyer [6] presents a translation of timed and untimed modal sequence diagrams [8] specifications into UPPAAL-TIGA [3], for the purpose of synthesis. Environment assumptions are supported through the use of *assumption MSDs*, scenarios explicitly tagged as specifying assumptions. In contrast, we chose to integrate assumptions into the same scenarios, so that a single scenario can specify a combination of system guarantees and environment assumptions. We believe this provides more flexibility. To the best of our knowledge, [6] is the only previous work that supports liveness and safety assumptions in the context of scenario-based specifications and synthesis.

Finally, Kugler et al. and Harel and Segall [12,18] present controller synthesis from LSC. These works, however, consider 'classic' LSCs, where the semantics ignores the temperature of environment controlled events, and thus do not support environment assumptions.

2 An Overview of Assume-Guarantee Scenarios

We start off with background about classic LSC and then demonstrate the contribution of the assume-guarantee extension through a presentation of a small example, a scenario-based specification of a vending machine. Part of the specification is presented here. Additional LSCs are presented in Sect. 5. The overview is semi-formal. Required formal definitions are given in the following sections.

2.1 Background on LSC

Live sequence charts (LSC) [5,8] is a scenario-based specification language, which extends classical message sequence charts (MSC) mainly with a universal interpretation and a distinction between mandatory and possible behavior. We give

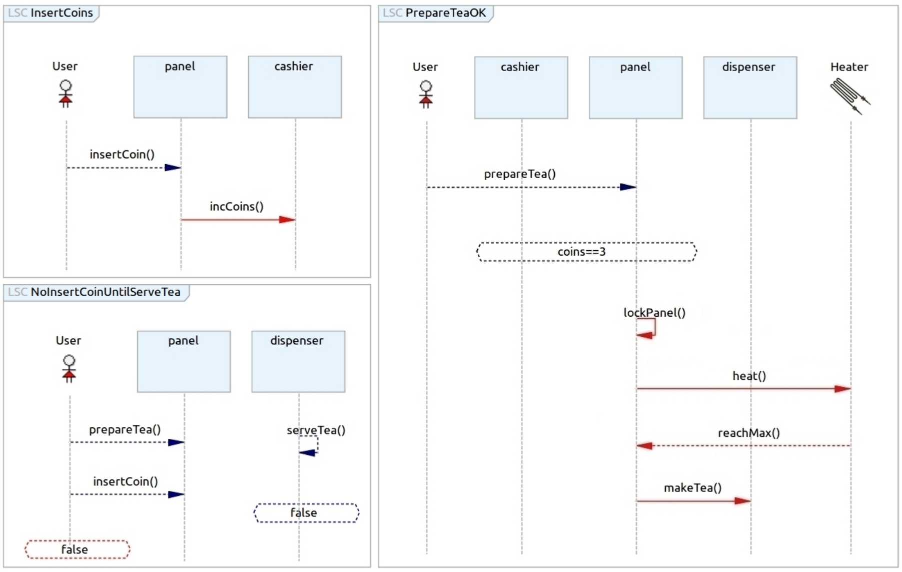


Fig. 1. Three scenarios from the vending machine specification. Note the distinction between system entities (panel, cashier, dispenser) and environment entities (user, heater). Also note the distinction between hot and cold elements, the hot environment controlled message `reachMax` in LSC `PrepareTeaOK` and the hot `false` condition on the user lifeline in LSC `NoInsertCoinUntilServeTea` (see Sect. 2).

a short, simplified overview of the language, with an emphasis on the parts most relevant to our present work. Detailed descriptions are available in [5,8].

An LSC consists of lifelines, messages, and conditions. A *lifeline* represents an interacting entity, controlled either by the system under development or by its environment (other systems, users etc.). A *message* represents a call between one entity and another. A message is a *system message* if it is sent from a lifeline controlled by the system, and is an *environment message* otherwise (if it is sent from a lifeline controlled by the environment). The LSC defines a partial order on its messages, induced by the vertical ordering of messages sent and received along the lifelines.

As an example, Fig. 1 (top left) shows the LSC `InsertCoins`. This LSC has one environment lifeline (controlled by the user) and two system lifelines, representing the system's `panel` and `cashier`. The first message `insertCoin` is an environment message and the second message `incCoins` is a system message.

The current state of an LSC is represented by a *system cut*, marking the progress of events along the LSC's lifelines. The *minimal cut* represents the state where the chart is closed. A cut induces a set of enabled and violating messages and conditions: a message is *enabled* in a cut of a chart if it appears immediately after the cut in the partial order defined by the chart; a message is *violating* in a cut of a chart if it appears in the chart, but is not enabled in the cut.

Messages have a hot or a cold temperature (red line or blue line syntax): a hot enabled message must eventually occur, while a cold enabled message may or may not eventually occur. A cut is hot if at least one of its enabled system messages is hot, and is cold otherwise. When an enabled message occurs, the chart progresses to the next cut. When a violating message occurs, progress depends on the temperature of the cut: if the cut was cold, the chart closes gracefully (the cut is set to be the minimal cut); if the cut was hot, this is a violation of the requirements and thus should have never occurred. In the LSC `InsertCoins` the first message is cold and the second message is hot.

Conditions have a hot or a cold temperature too and they are evaluated as soon as they are enabled. A hot enabled condition must be evaluated to true, while a cold enabled condition may or may not be evaluated to true. When a condition (hot or cold) is evaluated to true, the chart progresses to the next cut. When a condition is evaluated to false, progress depends on its temperature: if the condition was cold, the chart closes gracefully (the cut is set to be the minimal cut); if the condition was hot, this is a violation of the requirements and thus should have never occurred.

System messages can be marked as either *execution* (solid line) or *monitoring* (dashed line). All environment messages are marked as monitoring. A chart is *active* if its current cut has an enabled (system) message for execution. In the LSC `InsertCoins` the first message is marked for monitoring while the second is marked for execution. The cut after the first message is sent is active.

The semantics of a single LSC uses the partial order on messages and conditions defined by the chart, adds a universal interpretation, and relates to the hot (mandatory) and cold (optional) elements in it. Messages that do not appear in a chart are not constrained by the chart to occur or not to occur at any time, including in between the occurrence of messages that do appear in it.

For example, the semantics of the chart `InsertCoins` specifies the basic scenario of coin insertion: whenever the user inserts a coin (the user sends an `insertCoin` message) to the panel, the panel should eventually send `incCoins` message to the cashier (this increases the cashier's coins property). Implicitly, this also means that after `insertCoin` is sent, the system message `incCoins` must come before another `insertCoin` message is sent by the environment.

2.2 LSCs with Environment Assumptions

`InsertCoin` is a classic LSC: it specifies a system guarantee. What is impossible to specify in classic LSC are assumptions on the behavior of the environment. This is possible in the extended language. We give two examples below.

LSC `PrepareTeaOK` (Fig. 1 (top right)) describes the use case where the user asks the system to prepare tea and the number of coins inserted is exactly 3. Whenever the user sends a `prepareTea` message, the cold condition `coins==3` is evaluated. If it is false, the scenario exits gracefully. If it is true, the chart continues: the system's panel must eventually send its own `lockPanel` message and then ask the heater (controlled by the environment) to heat the water. This is followed by an assumption that the heater will eventually send a `reachMax` message back to the system's panel (note that `reachMax` is a hot message controlled by the environment). When a `reachMax` message is eventually received, the panel should eventually send a `makeTea` message to the dispenser.

LSC `NoInsertCoinUntilServeTea` (Fig. 1 (bottom left)) involves the user, the panel, and the dispenser. It specifies that whenever the user sends a `prepareTea` message to the panel, the user must not send an `insertCoin` message unless the dispenser has sent its own `serveTea` message. Note that if the user sends `insertCoin` after she sends `prepareTea` and before the dispenser has sent the `serveTea` message, then the LSC would reach a hot `false` condition on the user's lifeline, that is, this would constitute a violation of the assumption (when the `serveTea` message is sent, the chart closes gracefully because it reaches a cold `false` condition).

These two LSCs demonstrate the power of assume-guarantee scenarios in combining system guarantees and environment assumptions within a scenario-based specification setup.

3 Generalized Reactive Specification

We recall the definition of the class of generalized reactive of rank 1 specifications (GR(1)) [4,26], a fragment of LTL, which we use as the target for the definition of the semantics of assume-guarantee scenarios.

Linear temporal logic (LTL) [21,27] extends propositional logic with operators that describe variables valuations along infinite computation paths. Given a finite set of atomic propositions P, LTL formulae are constructed as $\varphi ::= p \mid \neg\varphi \mid \varphi \vee \varphi \mid \bigcirc\varphi \mid \varphi\mathcal{U}\varphi$, where $\bigcirc\varphi$ is the *next* temporal operator, roughly meaning that φ is true in the next step in the computation, $\varphi\mathcal{U}\psi$ is the *until* operator, roughly meaning that in any sequence of future steps φ is true *until* ψ is true. We use the usual abbreviations of the Boolean connectives $\wedge$, $\rightarrow$ and $\leftrightarrow$ and the usual definitions for true and false. Additional future temporal operators, $\Diamond$ (*eventually*) and $\Box$ (*globally*), are defined as abbreviations to $\mathsf{true}\mathcal{U}\varphi$ and $\neg\Diamond\neg\varphi$, respectively.

Definition 1 (The Class of Generalized Reactivity of Rank 1). *Let $\mathcal{V} = \{v_1, \ldots, v_n\}$ be a finite set of Boolean variables, $\mathcal{X} \subseteq \mathcal{V}$ be a set of input variables, and $\mathcal{Y} = \mathcal{V} \setminus \mathcal{X}$ be a set of output variables. The class of generalized reactive of rank 1 specifications (*GR(1)*) is defined to be* LTL *formulae of the form*

$$\psi : \left(\varphi_a^e \ \wedge \ \varphi_t^e \ \wedge \ \varphi_g^e\right) \longrightarrow \left(\varphi_a^s \ \wedge \ \varphi_t^s \ \wedge \ \varphi_g^s\right) \tag{1}$$

where:

(i) φ_a^e and φ_a^s are Boolean formulae which characterize the initial values that are assumed by the environment, and guaranteed by the system, respectively.

(ii) φ_t^e and φ_t^s are formulae of the form $\bigwedge_{i\in I}\Box B_i$ where each B_i is a Boolean formula that is a combination of variables from $\mathcal{X} \cup \mathcal{Y}$ and expressions of the form $\bigcirc v$ where $v \in \mathcal{X}$ and $v \in \mathcal{X} \cup \mathcal{Y}$, respectively. Intuitively, φ_t^e characterizes possible input to the controller, and φ_t^s characterizes possible transition of the controller.

(iii) φ_g^e *and* φ_g^s *are formulae of the form* $\bigwedge_{i \in I} \Box\Diamond B_i$ *where each* B_i *is a Boolean formula. The formula* φ_g^e *characterizes liveness assumptions on the environment input, and the formula* φ_g^s *characterizes liveness guarantees on the controller.*

Open systems are systems that interact with their environment, that is, receive some inputs and react to them. For such systems specifications are usually partitioned into assumptions and guarantees. The intended meaning is that if all assumptions hold then all guarantees should hold as well. That is, if the environment behaves as expected then the system will behave as expected as well. In the next section we present the semantics for assume-guarantee scenarios.

4 Assume-Guarantee Scenarios: Semantics

We are now ready to present our main contribution, i.e., an LTL-based semantics for a specification consisting of a set of assume-guarantee scenarios. We form the semantics within the GR(1) fragment.

4.1 Formal Settings

To model LSC behavior, we present formal settings similar to those presented in [7, 12, 19]. Given a set of LSCs, $\mathcal{L} = \{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$, we define $\mathcal{M}^s(\mathcal{L})$ (resp. $\mathcal{M}^e(\mathcal{L})$) to be the set of messages that the system (resp. environment) can send in the charts. The sets of system and environment messages are disjoint, i.e., $\mathcal{M}^s(\mathcal{L}) \cap \mathcal{M}^e(\mathcal{L}) = \emptyset$. We define a formal model using the following variables:

- m_e is an input *environment message variable* over the domain of all possible messages that the environment can send in $\mathcal{L}$, and an additional no-op value for doing nothing. Intuitively, to every message $m \in \mathcal{M}^e(\mathcal{L}) \cup \{$ *"no-op"*$\}$ sent by the environment, the synthesized strategy will "know" how to react.
- m_s is an output *system message variable* over the domain of all possible messages that the system can send in $\mathcal{L}$, and an additional no-op value for doing nothing. Intuitively, in every state, the synthesized strategy entails which system message $m \in \mathcal{M}^s(\mathcal{L}) \cup \{$ *"no-op"*$\}$ should be sent.
- $l_1, \ldots, l_n$ is a set of output *cut variables*. Each l_i encodes a *cut-automaton* for $\mathcal{L}_i$. The domain of l_i, which we denote by $Dom(l_i)$, consists of all possible cuts in $\mathcal{L}_i$, including a *minimal cut* of the chart (denoted by the value MIN). We add two unique sink values VIO^s and VIO^e, to represent hot-violation of the system guarantees and environment assumptions, respectively. The variable l_i maintains where the execution is at the moment along each lifeline in $\mathcal{L}_i$.[1]

Each $\mathcal{L}_i$ semantics is captured by the transitions of the cut-automaton that update its corresponding l_i variable according to the taken steps. The minimal cut value MIN indicates that the chart is currently closed. If a message is not a part of the chart then the cut-automaton can perform an idle step. The value

[1] Note that there are other ways to encode a cut (e.g., a variable per lifeline per LSC). Our formal settings is independent of any one specific encoding.

VIO^s captures the fact that the system performed a hot violation, and the chart can no longer be satisfied. On the other hand, the value VIO^e indicates that the environment did not fulfil its assumptions, and the chart is vacantly satisfied. We denote by $\rho_{\mathcal{L}_i}$ the transition of the cut-automaton for $\mathcal{L}_i$.

4.2 Superstep Requirements

Formally, a superstep is a series of messages sent by the system, encapsulated between two messages sent by the environment. When assumptions are not included in the semantics, as in the closed LSCs synthesis handled in [12,18], an artificial technical step is needed in order to enforce the superstep semantics. This exposes the internals of GR(1) and requires to deal with the mechanics of the game structure. Thus, it ties the solution to the GR(1) synthesis algorithm.

When assumptions are allowed, as in our open scenarios, a more natural and elegant way to describe the superstep semantics is available. Rather than working with the internals of the GR(1) game structure, we define the superstep semantics using two guarantees and two assumptions: G.1, G.2, A.1, and A.2 (see below). Thus, our approach defines a standalone LTL semantics that is independent of the mechanics of the synthesis algorithm (it could be solved with any LTL synthesis solution given that it is expressive enough to cover our specification).

First, we require the system to perform only a finite number of messages and give the environment a fair chance to communicate its messages.

Guarantee 1 (superstep: system fair turn): *The system always stops sending messages eventually.*

$$\Box\Diamond\,(m_s = \textit{no-op}) \tag{G.1}$$

We also require the system to perform a message only if the environment is not sending a message.

Guarantee 2 (superstep: system safe turn): *If the environment sent a message (i.e. m_e is different from no-op) then the system cannot send a message.*

$$\Box\bigcirc\,(m_e \neq \textit{no-op} \rightarrow m_s = \textit{no-op}) \tag{G.2}$$

Next, we require the environment to send one message at a time, allowing the system a fair chance to react to each message sent by the environment.

Assumption 1 (superstep: alternating turn): *If the environment sent a message in the last step (i.e. m_e is different from no-op) then the environment cannot send a message in the next step.*

$$\Box\left(m_e \neq \textit{no-op} \rightarrow \bigcirc\,(m_e = \textit{no-op})\right) \tag{A.1}$$

Finally, we require the environment to send a message only when the system is ready to receive one, allowing the system a fair chance to finish its (guaranteed to be) finite number of steps.

Assumption 2 (superstep: environment fair turn): *If the system sent a message in the last step (i.e. m_s is different from no-op) then the environment cannot send a message in the next step.*

$$\Box\left(m_s \neq \textit{no-op} \rightarrow \bigcirc\,(m_e = \textit{no-op})\right) \tag{A.2}$$

Note that the superstep requirements are fixed; they are not part of the application-specific semantics of the LSC specification. That is, the superstep requirements model our settings, whereas the additional requirements for the system (Subsect. 4.3) and for the environment (Subsect. 4.4, described below) model the application-specific semantics of the given LSC specification.

4.3 System Requirements

Given a set of LSCs $\mathcal{L} = \{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$, the application-specific system's semantics is defined using three guarantees: G.3, G.4, and G.5 (see below). To identify stable states in $\mathcal{L}_i$, we define $Act_i \subseteq Dom(l_i)$ to be the subset of active cuts from the domain of all cuts in the cut-automaton, i.e., cuts that contain an executable message that the system should perform.

First, we require the system to guarantee that each chart starts from its minimal cut.

Guarantee 3 (system: initial state): *For every* LSC $\mathcal{L}_i \in \mathcal{L}$, *the system starts from a state in which the cut variable* l_i *is set to the minimal cut.*

$$\bigwedge_{i=1}^{n} (l_i = \textsc{min}) \tag{G.3}$$

Second, we require the system to guarantee that each chart follows its transitional semantics as discussed in Subsect. 4.1.

Guarantee 4 (system: transition): *For every* LSC $\mathcal{L}_i \in \mathcal{L}$, *the system continuously preserves the transitions of the cut-automaton of* $\mathcal{L}_i$.

$$\bigwedge_{i=1}^{n} \Box \rho_{\mathcal{L}_i} \tag{G.4}$$

Finally, we require the system to guarantee to infinitely often visit a stable state, i.e., that infinitely often all charts visit inactive cuts in which there are no executable messages to be performed by the system.

Guarantee 5 (system: stable state): *The system always eventually reaches a state where every* $\mathcal{L}_i \in \mathcal{L}$ *is not active.*

$$\Box\Diamond \bigwedge_{i=1}^{n} (l_i \notin Act_i) \tag{G.5}$$

4.4 Environment Requirements

Given a set of LSCs $\mathcal{L} = \{\mathcal{L}_1, \ldots, \mathcal{L}_n\}$, the application-specific environment's semantics is given using three assumptions. Assumptions A.3, A.4 characterize liveness requirements, and A.5 characterizes safety requirements (see below). To identify states in which the system is waiting for messages from the environment in $\mathcal{L}_i$, we define $Exp_i \subseteq Dom(l_i)$ to be the subset of expecting cuts from the domain of all cuts in the cut-automaton, i.e., cuts that contain executable message to perform by the environment.

Furthermore, given a cut $c \in Dom(l_i)$ we define $\mathcal{E}^e(c)$ to be the set of hot environment messages enabled in cut c (i.e., if $c \in Dom(l_i) \setminus Exp_i$, then $\mathcal{E}^e(c) = \emptyset$). Intuitively, in cut c the system assumes that the environment messages $\mathcal{E}^e(c)$ are bound to happen eventually. On the other hand, the semantics of cold environment messages do not require any assumption, and are treated just like cold system messages (that is, a violation of a cold environment cut closes the chart gracefully and is not considered a violation of the requirements).

First, we require the environment to comply with a restricting (safety) property stating that if the system is in an expecting cut, then the next message sent by the environment is either *no-op* or one of the messages from the set of enabled hot environment messages. That is, the environment must focus on the hot messages at hand.

Assumption 3 (environment: active environment): *For every* LSC $\mathcal{L}_i \in \mathcal{L}$ *and every expecting cut* $c \in Exp_i$*, if in the last step the system was in cut* c*, then in the next step the environment sends either no-op message, or a message from the set of hot enabled environment messages.*

$$\bigwedge_{i=1}^{n} \bigwedge_{c \in Exp_i} \Box \Big(l_i = c \rightarrow \bigcirc \left(m_e \in \{\mathcal{E}^e(c) \cup \textit{no-op}\}\right)\Big) \tag{A.3}$$

Note that the requirement needs a rather loose restriction on the next step message since there could be cases where there are more than one possible hot environment message that is enabled. In such cases we would like to consider all possible combinations in which the environment meets its assumptions.

On the other hand, we require the environment to comply with the liveness property that states that when the system is in an expecting cut, then each enabled hot environment message must eventually be sent.

Assumption 4 (environment: fair environment): *For every* LSC $\mathcal{L}_i \in \mathcal{L}$*, every expecting cut* $c \in Exp_i$*, and every hot enabled environment message* $m \in \mathcal{E}^e(c)$*, the environment always eventually either sends message* m*, or the system is not in cut* c*.*

$$\bigwedge_{i=1}^{n} \bigwedge_{c \in Exp_i} \bigwedge_{m \in \mathcal{E}^e(c)} \Box\Diamond \left(l_i = c \rightarrow (m_e = m)\right) \tag{A.4}$$

The transition system semantics makes sure that if two hot environment messages are enabled in an expecting cut, and the first is being sent, then the following cut is also an expecting cut, which still awaits for the second hot environment message to be sent. Furthermore, unless the chart is closed, the execution cannot return to the previous expecting cut, thus the second hot environment message is bound to eventually be sent.

Note that from the system's perspective, the expecting cut is cold, that is, the system is allowed to violate it. However, as long as the system does not violate the expecting cut, the left side of the implication in both assumptions A.3 and A.4 hold, and the environment must follow in a way that would satisfy the right sides of these implications.

Finally, we would like to support explicit environment safety assumptions. The transition semantics makes sure that whenever a cut reaches a hot environment violation caused by an environment condition, l_i is set to the sink value $\textsc{vio}^e$. Even though variable l_i is a system output (that the environment cannot control directly), the guarantee of the transition semantics to indicate a hot environment violation, enables the environment to reason in its strategy all possible future violations of its assumptions.

Formally, we require the environment to avoid letting the system reach (in the future) the sink value that indicates a hot environment violation.

Assumption 5 (environment: safe environment): *For every* LSC $\mathcal{L}_i \in \mathcal{L}$, *the environment is never allowed to reach a hot environment violation.*

$$\bigwedge_{i=1}^{n} \square \left(l_i \neq \text{VIO}^e\right) \tag{A.5}$$

4.5 Summary

The combination of all the above LTL assumptions and guarantees (A.1–5 and G.1–5), consists of a semantics for an LSC specification. We formalize it in a GR(1) form, as shown in Equ. (2).

$$\begin{array}{c|c|c|c}
\text{environment} & A.3 & \bigwedge_{i=1}^{n} \bigwedge_{c \in Exp_i} \square \left(l_i = c \rightarrow \bigcirc \left(m_e \in \{\mathcal{E}^e(c) \cup \textit{no-op}\}\right)\right) & \wedge \\
 & A.4 & \bigwedge_{i=1}^{n} \bigwedge_{c \in Exp_i} \bigwedge_{m \in \mathcal{E}^e(c)} \square\Diamond \left(l_i = c \rightarrow (m_e = m)\right) & \wedge \\
 & A.5 & \bigwedge_{i=1}^{n} \square \left(l_i \neq \text{VIO}^e\right) & \wedge \\
\hline
\text{superstep} & A.1 & \square \left(m_e \neq \textit{no-op} \rightarrow \bigcirc \left(m_e = \textit{no-op}\right)\right) & \wedge \\
 & A.2 & \square \left(m_s \neq \textit{no-op} \rightarrow \bigcirc \left(m_e = \textit{no-op}\right)\right) & \\
\hline
 & \multicolumn{3}{c}{\textbf{implies}} \\
\hline
 & G.1 & \square\Diamond \left(m_s = \textit{no-op}\right) & \wedge \\
 & G.2 & \square\bigcirc \left(m_e \neq \textit{no-op} \rightarrow m_s = \textit{no-op}\right) & \wedge \\
\hline
\text{system} & G.3 & \bigwedge_{i=1}^{n} \left(l_i = \text{MIN}\right) & \wedge \\
 & G.4 & \bigwedge_{i=1}^{n} \square \rho_{\mathcal{L}_i} & \wedge \\
 & G.5 & \square\Diamond \bigwedge_{i=1}^{n} \left(l_i \notin Act_i\right) &
\end{array} \tag{2}$$

5 Assume-Guarantee Scenarios: Synthesis

The formulation of the semantics in the GR(1) form allows us to take advantage the game-based, symbolic, efficient synthesis algorithm of [26] and generate a correct-by-construction, executable controller from a specification consisting of a set of assume-guarantee scenarios. Below we motivate the need for synthesis, in comparison with weaker forms of execution. We then give an overview of the synthesis algorithm.

5.1 Running Example

As a running example we use a scenario-based specification of a vending machine. The specification consists of six LSCs, the three LSCs presented earlier

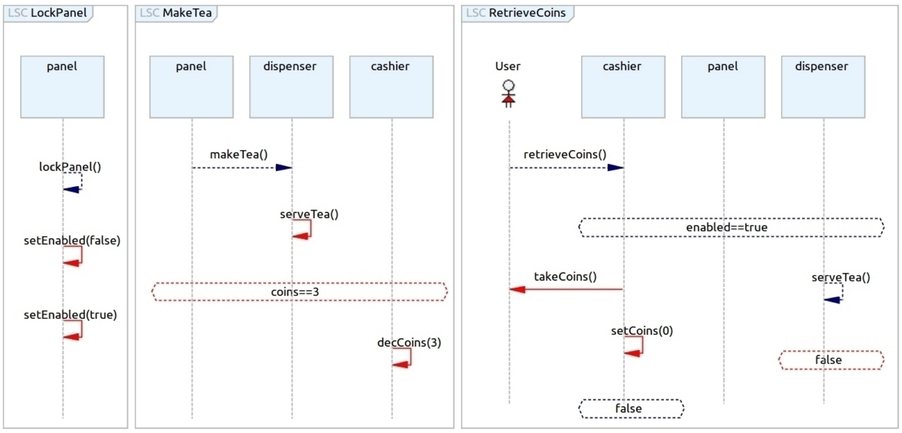


Fig. 2. Three additional LSCs from the vending machine specification (see Sect. 5)

in Fig. 1 and discussed in Sect. 2, namely `InsertCoins`, `PrepareTeaOK`, and `NoInsertCoinUntilServeTea`, and three additional LSCs, as shown in Fig. 2.

LSC `LockPanel` (Fig. 2 (left)) describes the implementation of the panel's locking mechanism. Whenever the `lockPanel` message is sent, the panel should eventually lock itself by setting its `enabled` property to false, and then eventually unlock itself by setting the `enabled` property to true.

LSC `MakeTea` (Fig. 2 (middle)) describes the behavior the system should follow whenever the panel sends the dispenser a `makeTea` message. In this case, the dispenser should send a self message to `serveTea`, an abstraction of a different scenario that entails the proper way to serve the tea. The chart continues to specify that after `serveTea`, the cashier's `coins` property must be exactly 3 (it is a hot condition), and be followed by a `decCoins(3)` message that will consume 3 coins (decrease the coins property by 3).

LSC `RetrieveCoins` (Fig. 2 (right)) enables a cancellation functionality. If the user sends a `retrieveCoins` message to the panel and the panel is enabled (note, a cold condition), then the system must send the user a `takeCoins` message (give back the coins to the user) and follow with a `setCoins(0)` message that sets the cashier's `coins` property to 0. Furthermore, the chart also specifies that during the process of cancellation, the dispenser cannot send a `serveTea` message (sending this message would make the hot `false` condition on the right hand side of the chart enabled, and thus result in a hot violation of the requirements).

Finally, the specification includes initial values for two properties: `coins` is set to 0 and `enabled` is set to `true`.

5.2 Why Do We Need Assume-Guarantee Synthesis?

LSC specifications are underspecified, since the language allows various kinds of non-determinism. Thus, a special mechanism is needed in order to execute an LSC specification. This execution mechanism is generically termed play-out [10]. The core of the play-out process is a strategy mechanism that is responsible for choosing the next method to execute. The choice is based on the specification and the current state of the system. Different kinds of play-out mechanisms may be defined. Each may be viewed as a different operational semantics for LSC. However, only synthesis can guarantee deadlock free execution (if one exists, see Subsect. 5.3), where if the environment behavior satisfies the assumptions then the system behavior would satisfy the guarantees. To motivate the need for assume-guarantee synthesis, we demonstrate the weaknesses of previously suggested play-out mechanisms below.

A naive operational semantics, termed play-out in [10], chooses a single system message that is enabled in some active LSC and that does not violate the current cut in all active LSCs, and executes it. Naive play-out does not guarantee that no violations will eventually occur (or rather that at each step there will be an enabled message that is not violating). Violations might happen since naive play-out makes its choices locally, without considering their future consequences.

For example, in the vending machine, after the user inserts a coin, the system must increase the coins property (LSC `InsertCoins`). After three coin insertions and sending `prepareTea`, the cold condition `coins==3` in `PrepareTeaOK` will hold and naive play-out would send a `lockPanel` message and a `heat` message to the `Heater`. Moreover, to follow LSC `LockPanel`, and since naive play-out does not consider future executions, it will immediately execute `setEnabled(false)` and `setEnabled(true)`. Now, if the user chooses to send `retrieveCoins`, the panel is enabled, the coins property will be set to zero, and so after `reachMax` is sent and `makeTea` is sent, a hot violation will be unavoidable in LSC `MakeTea`.

A better operational semantics for direct execution of scenario-based specifications is smart play-out (SPO) [7]. SPO can reason about possible violations within a single superstep. It guarantees to lead the system to a state where no LSC is active (a stable state), in preparation for the next environment message (if such a superstep exists).

However, looking only one superstep ahead is insufficient. For example, consider our vending machine specification, when `prepareTea` message is sent, smart play-out would fail to see the consequences of completing the superstep in `LockPanel`, since the violation is bound to occur only after two more supersteps (after the user will send `retrieveCoins` and the heater will send `reachMax`).

Both operational semantics presented above are rather weak and may in fact be viewed as unsound, as they may result in (partial) executions that cannot be extended to ones that satisfy the semantics of the LSC specification.

A stronger operational semantics for direct execution of scenario-based specifications is the synthesis presented in [12, 18], which we term closed synthesis. Closed synthesis reasons about the ongoing interaction between the environment and the system, and guarantees that in every state that the execution may reach, there exists a superstep that leads the system to a stable state. However, closed synthesis does not support environment assumptions. Thus, in our example, it will not be able to rely on the assumption induced by LSC `NoInsertCoinUntilServeTea` and thus would conclude that a controller cannot be synthesized: without this assumption a controller cannot be synthesized because the user may insert a coin while the heater heats the water, and thus force the system to serve tea when `coins > 3`, which would violate the hot condition in LSC `MakeTea`.

This discussion shows that assume-guarantee synthesis is indeed required.

5.3 Assume-Guarantee Scenarios Synthesis

The solution we use for synthesis requires a winning strategy. Given a GR(1) specification, computing a winning strategy for the system is done by solving a Streett game [29] where the system tries to either satisfy all its guarantees, or constantly falsify one of the environment's assumptions. We do this following the symbolic fixpoint algorithms described in [4,26]. Roughly, the algorithm starts from the set of all states and iterates 'backwards' by removing states from which the system is unable to force the execution to either reach all of the system's liveness guarantees, or constantly violate one of the environment's assumptions (each set of states where the assumption is constantly violated is computed using another nested fixpoint).

The fixpoint is reached when no additional states can be removed. If to every environment initial choice there exists a system initial choice in the fixpoint set, then the specification is realizable. A controller that implements the system's winning strategy can be constructed from the intermediate values of the fixpoint computation (see [4,26]). If the specification is realizable, then the construction of such a controller constitutes a solution to the synthesis problem.[2]

Going back to our example, assume-guarantee synthesis generates a controller that meets the specification. Specifically, it avoids the problems encountered by naive and smart play-out by sending the `setEnabled(false)` message but not sending the `setEnabled(true)` message until after the heater has sent `reachMax` (as it has to eventually). It also relies on the assumption that after `prepareTea` is sent, the user will not send an `insertCoin` message to the panel until `serveTea` is sent (as specified in LSC `NoInsertCoinUntilServeTea`).

6 Implementation

We have implemented our ideas using JTLV APIs [28] and integrated them into PlayGo [9]. PlayGo is an eclipse-based IDE built around the language of LSC and the play-in/play-out approach [10]. It includes a compiler that translates LSCs (given in a UML compliant form, using a profile, see [8]) into AspectJ code (based on [22,23]), and provides means for visualization, exploration, and debugging of LSC executions. JTLV is a Java-based framework for the development of formal verification algorithms, implemented as an Eclipse plug-in. The framework provides editors and developer-friendly high-level APIs.

We extended PlayGo's visual editor to support the extended language syntax, and implemented the reduction to the GR(1) game setup. The synthesis algorithm

[2] If the specification is unrealizable, then the synthesis computation fails. We have work in progress on addressing this case [24].

itself is implemented using JTLV. Finally, the resulting controller, as computed by the algorithm, is not only statically presented to the engineer. Rather, we translate it back and represent it using a play-out strategy, by generating the Java code PlayGo can use to guide the execution of the system.

7 Conclusion and Future Work

We have presented an extension of live sequence charts that supports environment assumptions. The semantics of the extended language is given in the form of a GR(1) formula, and thus enables the efficient synthesis of a correct-by-construction controller. The work is implemented and demonstrated with running examples.

In a related work in progress [24] we deal with the debugging of unrealizable scenario-based specifications (with or without assumptions). When a specification is unrealizable, we reverse the roles of the system and the environment and compute a counter strategy [16,25]. The counter strategy serves as a formal proof that shows how an adverse environment can adhere to the assumptions (if any) while forcing any system to fail in fulfilling its guarantees.

In Sect. 4 we have defined a global stability guarantee G.5. An alternative, weaker semantics, could have used a local stability guarantee: $\bigwedge_{i=1}^{n} \Box\Diamond\big(l_i \notin Act_i\big)$. Note that this semantics may induce a more complex synthesis solution, but which is still of course within the GR(1) fragment. Moreover, the global variant implies the local one. Although we have chosen to present the global stability guarantee, we believe that the local one may be useful in some contexts and may perhaps be more in line with the breakup of the specification into scenarios. We leave the choice between the two alternative semantics open for discussion.

Finally, one may consider an alternative, tighter semantics for LSC, using the stronger GR(K) form (see Chap. 4. of [21]), which handles formulae consisting of k conjunctions of GR(1) implications. GR(K) is more expressive than GR(1) (in fact GR(K) is as expressive as LTL), however solving it is computationally harder (exponential in k, [25]). GR(1) can serve as an efficient precondition to the more locally aware formulation of GR(K).

In the context of scenario-based specifications, the essence of the difference between GR(1) and GR(K) is in the question of whether all assumptions should be grouped together into a single conjunct on the left side of the GR(1) implication, or whether each scenario should induce its own local implication between assumptions and guarantees. It is not clear whether the GR(K) semantics captures the idea of scenario-based specifications better than the GR(1) semantics. We leave the formal definition of the alternative GR(K) semantics and its evaluation against the GR(1) semantics for future work.

References

1. Alfonso, A., Braberman, V.A., Kicillof, N., Olivero, A.: Visual timed event scenarios. In: Finkelstein, A., Estublier, J., Rosenblum, D.S. (eds.) ICSE, pp. 168–177. IEEE Computer Society (2004)

2. Autili, M., Inverardi, P., Pelliccione, P.: Graphical scenarios for specifying temporal properties: an automated approach. Autom. Softw. Eng. 14(3), 293–340 (2007)
3. Behrmann, G., Cougnard, A., David, A., Fleury, E., Larsen, K.G., Lime, D.: UPPAAL-Tiga: Time for Playing Games! In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 121–125. Springer, Heidelberg (2007)
4. Bloem, R., Jobstmann, B., Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. Journal of Computer and System Sciences 78(3), 911–938 (2012)
5. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. Formal Methods in System Design 19(1), 45–80 (2001)
6. Greenyer, J.: Scenario-based Design of Mechatronic Systems. PhD thesis, University of Paderborn, Department of Computer Science (2011)
7. Harel, D., Kugler, H., Marelly, R., Pnueli, A.: Smart Play-out of Behavioral Requirements. In: Aagaard, M.D., O'Leary, J.W. (eds.) FMCAD 2002. LNCS, vol. 2517, pp. 378–398. Springer, Heidelberg (2002)
8. Harel, D., Maoz, S.: Assert and negate revisited: Modal semantics for UML sequence diagrams. Software and Systems Modeling 7(2), 237–252 (2008)
9. Harel, D., Maoz, S., Szekely, S., Barkan, D.: PlayGo: towards a comprehensive tool for scenario based programming. In: ASE, pp. 359–360. ACM (2010)
10. Harel, D., Marelly, R.: Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine. Springer (2003)
11. Harel, D., Pnueli, A.: On the Development of Reactive Systems. In: Apt, K.R. (ed.) Logics and Models of Concurrent Systems. ATO ASI Series, vol. F-13, pp. 477–498. Springer (1985)
12. Harel, D., Segall, I.: Synthesis from scenario-based specifications. Journal of Computer and System Sciences 78(3), 970–980 (2012)
13. Haugen, Ø., Husa, K.E., Runde, R.K., Stølen, K.: STAIRS towards formal design with sequence diagrams. Software and Systems Modeling 4(4), 355–367 (2005)
14. Jackson, M.: The world and the machine. In: Perry, D.E., Jeffrey, R., Notkin, D. (eds.) ICSE, pp. 283–292. ACM (1995)
15. Knapp, A., Wuttke, J.: Model Checking of UML 2.0 Interactions. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 42–51. Springer, Heidelberg (2007)
16. Könighofer, R., Hofferek, G., Bloem, R.: Debugging formal specifications using simple counterstrategies. In: FMCAD, pp. 152–159. IEEE (2009)
17. Krüger, I., Grosu, R., Scholz, P., Broy, M.: From MSCs to Statecharts. In: DIPES, pp. 61–72 (1998)
18. Kugler, H., Plock, C., Pnueli, A.: Controller Synthesis from LSC Requirements. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 79–93. Springer, Heidelberg (2009)
19. Kugler, H., Segall, I.: Compositional Synthesis of Reactive Systems from Live Sequence Chart Specifications. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 77–91. Springer, Heidelberg (2009)
20. Kupferman, O., Vardi, M.Y.: Module Checking Revisited. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 36–47. Springer, Heidelberg (1997)
21. Manna, Z., Pnueli, A.: The temporal logic of concurrent and reactive systems: specification. Springer (1992)
22. Maoz, S., Harel, D.: From multi-modal scenarios to code: compiling LSCs into AspectJ. In: SIGSOFT FSE, pp. 219–230. ACM (2006)
23. Maoz, S., Harel, D., Kleinbort, A.: A compiler for multimodal scenarios: Transforming LSCs into AspectJ. ACM Trans. Softw. Eng. Methodol. 20(4), 18 (2011)
24. Maoz, S., Sa'ar, Y.: Counter play-out: Executing unrealizable scenario-based specifications (in preparation, 2012)

25. Piterman, N., Pnueli, A.: Faster solutions of Rabin and Streett games. In: LICS, pp. 275–284. IEEE Computer Society (2006)
26. Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of Reactive(1) Designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 364–380. Springer, Heidelberg (2005)
27. Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57. IEEE (1977)
28. Pnueli, A., Sa'ar, Y., Zuck, L.D.: JTLV: A Framework for Developing Verification Algorithms. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 171–174. Springer, Heidelberg (2010)
29. Streett, R.S.: Propositional dynamic logic of looping and converse is elementarily decidable. Information and Control 54(1/2), 121–141 (1982)
30. Whittle, J., Schumann, J.: Generating statechart designs from scenarios. In: ICSE, pp. 314–323. ACM (2000)

An Exploratory Study of Forces and Frictions Affecting Large-Scale Model-Driven Development

Adrian Kuhn, Gail C. Murphy, and C. Albert Thompson

Department of Computer Science
University of British Columbia, Canada

Abstract. In this paper, we investigate model-driven engineering, reporting on an exploratory case-study conducted at a large automotive company. The study consisted of interviews with 20 engineers and managers working in different roles. We found that, in the context of a large organization, contextual forces dominate the cognitive issues of using model-driven technology. The four forces we identified that are likely independent of the particular abstractions chosen as the basis of software development are the need for diffing in software product lines, the needs for problem-specific languages and types, the need for live modeling in exploratory activities, and the need for point-to-point traceability between artifacts. We also identified triggers of accidental complexity, which we refer to as points of friction introduced by languages and tools. Examples of the friction points identified are insufficient support for model diffing, point-to-point traceability, and model changes at runtime.

1 Introduction

Model-driven engineering (MDE) is the primary use of, often visual, models for software engineering. Although technical approaches of model-driven engineering are well-documented, there is a paucity of information about how humans interact with and adapt to the technology.

In this paper, we investigate the human aspects, reporting on an exploratory qualitative study conducted at General Motors, a large automotive company who makes extensive use of model-driven engineering.

Our study involved interviews with 20 engineers and managers. These interviews took an individual-out perspective, that is from the perspective of engineers to their context, focusing on how an individual is applying and grappling with model-driven technology to complete assigned goals. We analyzed the interviews to identify triggers of complexity that may arise when working with software models and how those triggers compare to those found in more traditional forms of source-based development.

We look at triggers of complexity in terms of forces and points of friction. The forces are likely independent of the particular abstractions chosen as the basis of software development and thus should be considered in the design of any new abstractions. Our notion of forces is similar to Brooks's notion of essential complexity from his "No Silver Bullet" essay [1]; they transcend the modeling technologies used. Related to each force we also identified points of friction, which are akin to Brooks's notion of accidental complexity; namely complexity introduced by languages and tools.

Through our study, we identified four forces and five points of friction that affect the use of model-driven engineering at the industrial site we studied, which may provide insight into model-driven engineering in general. They are as follows:

– Teams are typically working on multiple versions of the same software model (force), yet engineers lack proper tooling to identify and share diffs (friction).
– Domain experts use a rich set of visual and formal languages to invent novel algorithms (force), yet they lack tool support to define their own little visual languages (friction) or pluggable ad-hoc type systems (friction).
– Inventing novel algorithms for vehicle control is an exploratory activity (force), while the needs of early prototyping are well addressed by in-silico simulations, testing on actual vehicles, which occurs later in the process, suffers from lacking tool support for model changes at runtime (friction).
– Requirements documents and software models need be kept consistent across development iterations (force), yet engineers lack proper tooling to track point-to-point correspondences between corresponding artifacts (friction).

We believe that the forces and frictions we have identified through this empirical study can help software engineering researchers understand the context in which model-driven software engineering occurs in practice and that the friction points we identified can influence new modeling languages and tools. The specific results of this study can also help those adopting model-driven engineering to understand cognitive issues that may impact the use of MDE.

This paper makes four contributions:

– it introduces the notion of forces and points of friction in tooling to describe the impact of technical issues in the use of model-driven engineering,
– it identifies and presents four forces that may significantly impact the use of model-driven engineering,
– it identifies and presents five points of friction in existing language and tool support for model-driven engineering,
– it provides points of comparison with source code development to help tease apart essential and accidental complexity.

The remainder of this paper is structured as follows. Section 2 discusses methodology of our field study, Section 3 presents the software development process at the organization we studied, Section 4 presents the findings of our study, enumerated as contextual forces and points of frictions, Section 5 discusses our findings in the general context of model-based design, Section 6 presents related work, and Section 7 concludes with concluding remarks.

2 Methodology

To enable the gathering of detailed, rich and contextual information about model-driven engineering, we chose a qualitative study approach. We visited the industry of interest (General Motors) on two separate occasions, collecting data constructed through semi-structured in-depth interviewing. We interviewed 12 engineers and 8 managers. Overall, the engineers we interviewed came from four different teams from different company departments. All teams were global, that is spread across sites in India and America, however we interviewed people from the American sites only. The 12 engineers selected for interviews were sampled from several roles however their profiles are similar, that is they all work with the same process and use the same modeling technology. Each interview was 90–120 minutes long, recorded on tape and transcribed for encoding by one of the authors of this paper.

In a first visit, we interviewed 10 participants from both management and technical roles to familiarize ourselves with the software process used in the automotive industry. Based on what we learned from the first interviews, in our second visit, we interviewed an additional 10 participants, all of them working with software models but in different roles. The interviews were semi-structured, following an exploratory case-study approach where open ended questions are asked in order to identify research hypothesis for future studies [2]. We asked participants to describe their work, how their work fits into the process of the organization, with whom they interact on a weekly basis, and which artifacts are the input and which are the output of their work. We also asked to see current or recent examples of artifacts on which they were working.

We transcribed the 12 interviews with engineers (4 from the first visit and 8 from the second visit). We encoded the transcripts and from this encoding, we distilled the contextual forces and points of friction presented in this paper. We encoded the interviews by tagging sentences with hashtags as if they were tweets. We then used a series of tag clouds to identify patterns in the data, merging and splitting tags as we saw need. We did two passes over the tags, a first one to identify all forces and frictions that shape the work of the participants, and a second pass to identify forces and frictions that might provide the basis for general hypotheses on model-driven engineering, ruling out those that are specific to the organization under study.

The data presented in this paper is largely from the in-depth interviews with 12 engineers. These engineers worked in the following roles: 2 domain experts, 7 software modelers, and 3 testing engineers. The participants had an average of 12.5 ± 8 years of professional experience with software engineering and an average of 4.5 ± 4 years of professional experience with modeling; their backgrounds were electronic engineering (9 mentions), mechanical engineering (4 mentions), computer engineering (3 mentions), and software engineering (1 mention). There are more than 12 mentions as some engineers had two degrees.

Threats to Validity: We selected all participants from the same organization, whose common context and corporate culture may bias the results. We were fortunate however to interview participants from four global teams and a wide variety of roles, providing us with multiple views on the cognitive issues of working

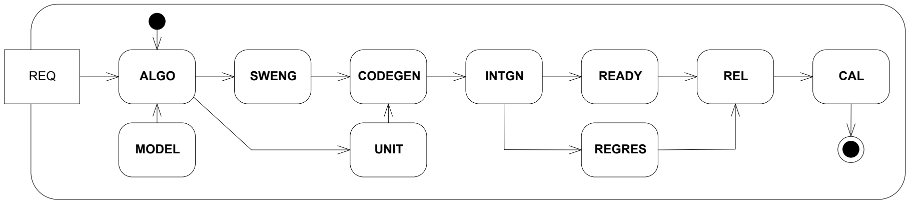


Fig. 1. Software development process: the doodle shows all stages of an iterative six week release. From left to right, the stages of the process are: REQ) requirements collection, ALGO) algorithm design, MODEL) in-silico simulations; SWENG) software model development; CODEGEN) code-generation; UNIT) unit testing; INTGN) integration on embedded chips; READY) readiness testing; REGRES) regression testing; REL) internal release; CAL) calibration on actual vehicles.

with modeling technology. Although a study at one organization is not sufficient to make broad generalizations, this initial data can provide at least one practical reference point of context that is otherwise often absent in language and tool design. This practical reference point can provide a basis for more specific hypotheses to study in future empirical work in this area.

3 Modeling at General Motors

To enable interpretation of our qualitative study results, we provide an overview of the software development process at General Motors. We begin by describing the overall software development process used, followed by a more in-depth description of the various roles involved with software development and the artifacts produced and consumed during the process.

3.1 Process Overview

Figure 1 shows the software development process commonly used in the automotive industry. While the figure depicts a sequential flow from requirements to deployed software on the vehicle, the actual process happens in iterative releases of six weeks with different stages of the process running in parallel on subsequent releases. Development begins with requirements collection, which typically happens outside the software development team (REQ in Figure 1). The requirements are consumed by domain experts of the team who perform algorithm design (ALGO), which includes running tests of developed models on in-silico simulated vehicles. Software modelers consume developed algorithms (either requirements or model patch) to produce software models from which code can be generated in an automated step (CODEGEN); code generation is 100% automated, a special team of "language designers" maintains the rules used for code generation. Test engineers use the results of algorithm design and generated code to perform unit tests; these engineers work primarily with source code. Integration engineers take care of integrating produced software to embedded chips. Test engineers take the results and perform readiness (READY) and regression tests (REGRESS). Every six weeks, teams downstream in the process receive new software that is calibrated on the car (CAL). This step involves calibrating the parameters of the typically generic features developed in the software to a specific car model.

Each team following this process typically owns a single feature and the models that describe that feature. The models for a single feature are reused for different versions (world region, national legislation, car model and year) of a particular car. As described above, special teams do exist that provide the other teams with infrastructure and code-generation rules.

3.2 Roles

A software development team responsible for a feature consists of about a dozen people working in different roles and possibly different countries. Through our interviews we learned about four different roles.

Domain Experts are responsible for maintaining requirements documentation and inventing novel algorithms. In the former responsibility, domain experts work more distinctly from software modelers. In the latter responsibility, domain experts work closely with software modelers, including drafting changes to models on which the software modelers work. The algorithms that the domain experts are designing are not so much computational algorithms but rather involve the physics of a vehicle. Most domain experts thus have a strong background in electrical or mechanical engineering, but typically no formal education in software engineering.

Software Modelers implement and maintain models as specified by domain experts. Software modelers are responsible for three to four models and are in close collaboration with the domain experts who own the corresponding requirements documentation. Software modelers use the MatLab Simulink[1] or IBM Rhapsody[2] tools; we describe more about these tools in the next section. When the models compile, they are passed on to integration engineers for integration into a release. Most software modelers have a background in mechanical or electrical engineering, some have a minor in computer or software engineering, but this is the exception rather than a rule.

Test Engineers perform delta and regression testing of releases and are responsible for root cause analysis of an incoming anomaly report (i.e., bug reports). Test engineers typically work with generated sources rather than models. Test engineers are exposed to all artifacts in the process and tend to have the broadest knowledge of a team's feature. New hires are often first assigned a test engineering role before moving on to a software engineering role. The professional background of test engineers is the same as for software modelers.

Code-generation engineers belong to a special team that owns and maintains the rules used to automatically generate source code from the software models. These experts also publish modeling guidelines and naming conventions. Even though not formally established by the process, software modelers are often in close contact with code generation engineers, providing them with feedback and getting help when they struggle with code-generation issues.

[1] http://www.mathworks.com/products/simulink

[2] http://www.ibm.com/software/awdtools/rhapsody

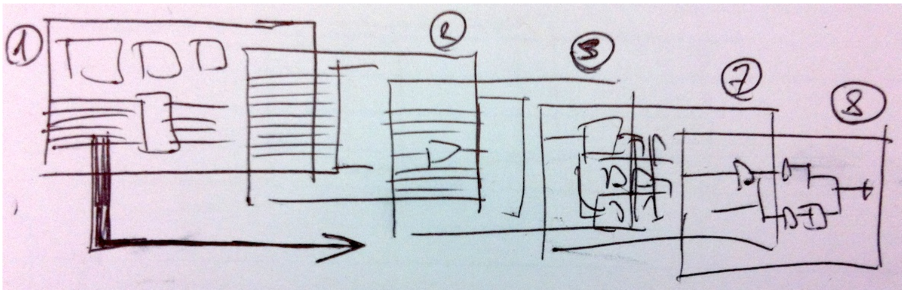


Fig. 2. Sketch of a Simulink model: from left to right we see model layers of increasing nesting level, starting with the entry function down to implementation logic

3.3 Artifacts

Requirements and software models are the main representations used in the software development process. We describe these two artifact types and highlight four kind of secondary artifact types that are relevant to our results.

Requirements Documents are specifying features and owned by a team. These documents are maintained by the domain experts. The requirements documents that we saw are loosely structured MS Word documents, typically containing a mixture of natural language text, pseudo-code and figures. Figures within these documents often use problem-specific visual languages and are created manually. In maintenance teams, requirements documents are changed first and drive subsequent changes to software models. In innovative teams, domain experts explore the solution space by drafting changes to the software models themselves and requirements documents are updated once the algorithms stabilize.

Software Models are created by software modelers with either Matlab Simulink or IBM Rational Rhapsody. While the two are interchangeable in the process and used by the same roles, they are technically quite different:

– Matlab Simulink is a model-based design tool focused on the design of control flows. Simulink models are written in a low-level visual language which resembles the visuals of circuit diagrams. Code generation with Simulink is automated but cannot be customized.
– IBM Rhapsody is a model-driven engineering tool. The structure of Rhapsody models is specified using UML class diagrams, where engineers can choose between visual and non-visual representations, and behavior is specified using either blocks of C-code or state machine diagrams. Code generation with Rhapsody is automated and highly customizable.

Figure 2 shows a sketch of a typical Simulink model that implements part of a feature. A typical model consists of about 100,000 blocks and a dozen nested layers. From left to right, we see model layers of increasing nesting level: 1) the top most layer of the model, which is structured according to the modeling guidelines with "the function" on top and other diagnostics function on the bottom; 2) the second layer, zooming into the function, showing 96 input signals and 45 output

Table 1. List of observed frictions by participant numbers. Roles are X) domain expert, S) software modeler, and T) test engineer; an asterisk indicates that their team focused on inventing novel algorithms rather than on maintenance of stable technology.

Sec	**Friction**	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8	P_9	P_{10}	P_{11}	P_{12}
4.1	Insufficient model diffing	●	●	●	●	●	●	●	●	●	●	●	●
4.2	Need for Visual DSLs									●	●	●	
4.2	Hungarian notation as types	●	●			●	●						
4.3	Need for exploratory programming				●					●	●	●	
4.4	Lack of P2P traceability	●	●	●	●	●	●	●	●	●	●	●	●
	Role	S	S	S	T*	S	S	T	T	S*	X*	X*	S*

signals; 3) One of many layers that serves to convert the unit and magnitude of input signals, typically each of those layers corresponds to a paragraph in the requirements document; 4) about a dozen layers deeper, program logic such as conditionals and loops are laid-out as a graphical circuit with each major block corresponding to a paragraph in the requirements document; 5) further down inside one of the blocks with program logic, basic arithmetic operations, such as addition and division of numbers, are modeled using the visual language of circuits rather than using mathematical notation.

Auto-Generated Source Code is often defined as a secondary artifact type; however, this code serves as the primary artifacts used by test engineers and sometimes, for diffing and change tracking, by software modelers.

Code-Generation Rules are used for automated code generation. These rules are maintained by a special team of code-generation engineers.

Model Patches are used by domain experts when they invent novel algorithms to exchange their model prototypes with the software modelers. These patches are not a formally defined part of the software development process and thus are *ad-hoc* artifacts. Model patches take many forms, such as Excel spreadsheets with annotated screenshots of a model.

Tests ensure that software models are implementing a feature as specified in the requirements documents. Within a team all tests are owned by the test engineers. Many tests require manual manipulation on a workbench, that is a partial vehicle in the testing lab, while other tests are fully automated and run in an in-silico simulation of the vehicle and its environment.

4 Results — Forces and Frictions

From our analysis of the interview data, we identified four forces and five points of friction. Using Brooks's terminology, forces are indicators of innate complexity while frictions introduce accidental complexity [1].

Table 1 documents the observed frictions by participant number. As we distilled the forces and frictions from the encoded interview data, we tried to identify those triggers of complexity that are not specific to the organization that we studied. We focused on those that are more general and thus might form the basis for more specific hypotheses in future research work.

4.1 Force: Need for Diffing in Software Product Lines

Engineers in a team are often working on different versions of the same artifact. Thus, engineers need to identify the changes in a model and, possibly, share those changes with engineers in other roles. Working with multiple versions is a result of business needs and thus a contextual force, independent of the primary abstraction used for software representation, i.e., models or code.

Internal releases happen every six weeks, however, the length of a full iteration might be longer. It is common for multiple engineers, in the same team, to work on different releases of the same model. In particular, we learned that domain experts typically use previous releases to prototype the changes that are supposed to drive future releases. Thus, domain experts need to exchange those prototypes as model patches with the software modelers. Also, test engineers reported that they need to learn about the most recent changes to a model under test.

Friction: Insufficient Support for Model Diffing (12/12 interviews). Engineers use version control to keep track of different versions and revisions of the same model. However, they experience friction when merging and handling comparison of these versions.

Although there are commercially available third-party tools that offer diffing capabilities for modeling, the engineers we interviewed described that they are limited in their scalability and in their usability. Engineers described the experience of using these tools as *"going blind"* (P_{10}) and leading them to make mistakes. Engineers seem to prefer a linear reading path of textual diffing in order to make it easier for them to *"not miss a change"* (P_9). Model-based approaches which highlight the changes in the spatial and possibly nested visual representation do not provide that kind of linear reading path.

We learned about several different strategies that engineers use to work around the lack of model diffing:

– When coming back to their own work, or keeping track of changes for code reviews, software modelers adopted a habit of documenting all model changes with comments. One example is the unique identifier of the current work ticket used as a marker, such that a search for this marker returns all model changes. This approach is the same as the approach adopted to handle missing point-to-point traceability, which is discussed in Subsection 4.4.
– When comparing versions for regression testing and root cause analysis, test engineers use textual diffing tools on the auto-generated code, which puts them at risk to misinterpret the modeler's intention of a feature.
– When inventing algorithms and prototyping on their own branch of a model, domain experts often use screenshots to communicate their changes back to the software modeler who owns the model. They take screenshots where the changes before and after, marked them in red, and share them in a PowerPoint slide deck as an *ad-hoc* model patch which is either emailed or attached to a change ticket.

The engineering needs for model diffing are similar to those found in traditional source code development. We did not hear in the interviews that an increased level of abstraction in representation (that is modeling rather than code) leads to an increased need for semantic diffing. Our observations suggest that end users would like to have more scalable and usable syntactic diffing rather than semantic diffing. As reported by the engineers who are falling back to textual diffing of auto-generated source, syntactic diffing is in their words *"more than good enough"* (P_7) for most use cases; in particular when diffing is needed to track the changes from one version to another.

4.2 Force: Need for Problem-Specific Expressibility

Domain experts use a rich set of visual and formal languages to invent and design their algorithms. The requirements documents that we encountered in our study made use of a rich and diverse visual language to describe the desired behavior of algorithms. Some of these languages are by virtue of the domain expert's training as mechanical or electrical engineers, whereas others of those languages are a result of the domain expert's struggle to find the best way to explore the problem and solution spaces of their inventions.

We found that the modeling tools in our study, while providing the specialized team of code-generation engineers with powerful abstractions, do not empower end-users, that is domain experts and software modelers, to define their own problem-specific "little languages" [3]. We highlight two major points of friction related to insufficient expressibility that we identified: visual languages and *ad-hoc* types.

Friction: Lack of Problem-Specific Visual "Little Languages" (3/12 interviews). Domain experts often need to prototype their innovations in a software model, yet the visual language of modeling tools limits their ability to express themselves. The notations that domain expert use to talk and think about their algorithms are those found in mechanical and electrical engineering.

For example, a domain expert might be prototyping a novel clutch control. In the requirements documents, the domain experts might describe the behavior of two dependent variables as a graph with two signals in time, quote: *"there are pictures in here of how I want the data to behave, and when I am done I want to see this [on the oscilloscope] on a car."* (P_{10}) Yet, when the domain expert uses software models to explore the solution space of the novel algorithm, he has to constantly translate back and forth between his mental model and the programing constructs. The domain expert cannot just draw a graph of the expected behavior and have appropriate code generated.

Support for domain-specific modeling might alleviate this point of friction, please refer to the discussion in Subsection 5.3 for more information.

Friction: Hungarian Notation Used as Ad-hoc Types (4/12 interviews). We also found that some teams used Hungarian notation to denote physical unit and magnitude of signal names in the Simulink models. Hungarian notation was popular in software engineering before the introduction of type systems. It is a naming convention where variable names were prefixed with abbreviations indicating the type of a variable, e.g., `szName` for a variable storing a username as zero-terminated string.

The software engineers described to us how they use Hungarian notation to denote the physical type and magnitude of signals in their models. For example, they use a prefix to indicate that a signal is temperature in degree Celsius and that it is a fixed-point number with base 10 and radix 2. The printed list of all prefixes used in the system fills four pages and they keep them close to the keyboard, to have them always ready when working with the models. Engineers use these prefixes to make sure that values are properly converted and normalized before use. However, these coding conventions are only manually, not automatically, verified.

Support for problem-specific type systems in modeling technology, as for example pluggable types [4], might alleviate this point of friction.

4.3 Force: Inventing Novel Algorithms as an Exploratory Activity

Developing algorithms for vehicle control is an exploratory activity. While the needs of early prototyping are well addressed by in-silico simulations, this is not the case for later stages where novel algorithms are tested on actual vehicles. As engineers are testing software "on the car," during a test drive on the proving grounds, they often encounter the need for updates to the software system. This need has been reported by domain experts and software engineers in those teams who work on inventing novel algorithms.

Friction: Long Build-Cycles Prevent Live Modeling (4/12 interviews). The build process of the model-driven tool chain may take up to several hours. As a result, when working "on the car," as soon as the need for a software change arises the test drive has to be interrupted and rescheduled for another day. Engineers reported that build times with an older C-based tool chain had been in the half-hour range and thus within a tolerance interval where it had been possible to continue the test drive on the same day. Ideally though, when working "on the car" engineers should be able to apply model changes at runtime and continue their test drive instantly.

This point of friction might be alleviated by an abstraction which does away with compile-build-deploy cycles, such that changes to the software can be applied at runtime. Technologies that allow a form of hot-swapping of quickly-generated code from models might be a means to address exploratory adaptation of software at runtime. Such technologies would not be limited by the processing power of target hardware (embedded control units) since while working "on the car" those chips are stubbed by more powerful hardware anyway.

4.4 Force: Need for Traceability in Incremental Release Cycles

A major theme that appeared throughout the interviews was the need for traceability between specification documents and software artifacts. Requirements documents, software models and tests are all essentially different representations of the same information, which need be kept consistent as those artifacts are independently updated with each release cycle.

While the content management system used in the organization provides engineers with document-to-document traceability, for many tasks point-to-point traceability is required. Engineers need to be able to quickly navigate from a visual block in the software model to the corresponding paragraph in the requirements document, or the corresponding test, or even the auto-generated sources, and vice versa. While this need is essentially representation independent, the introduction of software models as an additional layer of abstraction exponentially increases the traceability needs of engineers.

Friction: Lack of Point-to-Point Traceability (12/12 interviews). Currently, engineers establish traceability by relying on naming conventions. All interviewed engineers mentioned the use of markers as a work-around for missing point-to-point traceability. We found that engineers have adopted a habit of using change ticket identifiers as markers to establish point-to-point traceability through manual search. This is similar to one of the habits adopted to tracking changes between model versions as discussed in Subsection 4.1.

While this workaround establishes limited point-to-point traceability, the approach is inefficient and fragile. If engineers forget to mark one of the documents with the unique identifier, traceability is broken. In addition, while names contained in software artifacts are verified by code generation or compilation, names contained in specification documents often contain spelling errors or use old names that predate a renaming refactoring. Spelling differences make it hard, if not impossible, to navigate these traceability links using keyword search.

5 Discussion

In this section, we discuss our observations in the context of model-driven engineering and provide points of comparison with source code development to help tear apart essential and accidental complexity.

5.1 On the Terminology of "Model"

In our interviews, we found that the terms "model" and "modeling" were used ambiguously. Engineers generally did not refer to their work as "modeling" but used the terms "auto-coding" and "hand-coding." These terms were used to differentiate between working with tools which include a step of code generation versus writing C-level code manually. Engineers used the term "model" ambiguously to refer to software models, as well as the *plant models* used for the in-silico simulation of vehicles. Engineers also used the term "simulation" ambiguously to refer to running the in-silico simulation of the plant models, as well as to running software models from within the modeling tools as opposed to running the auto-generated sources.

We believe the terminology we observed is mixing model-based design (MBD, an approach in system engineering for disentangling the development of control software and corresponding vehicles, using in-silico modeling while vehicles are not yet available) and model-driven engineering (MDE). The ambiguous use of terminology can be explained if we look at model-driven engineering as a division of labour between a few specialized language designers and many modelers. After all, the software engineers do not have to understand the full complexity of modeling, this is up to the specialized code-generation engineers. However, we found that points of friction in modeling tools, in particular the insufficient support of model diffing, may break the abstraction and nevertheless expose engineers to these complexities.

5.2 On Visual Models and Linear Reading Paths

During our interviews we learned about heated controversy around modeling among engineers, and whether hand-coding is superior to code generation. While some of the critique was targeted at the long build cycles of the modeling toolchain (see Subsection 4.3), much of it was concerned with the visual representation of models and its lack of abstraction such as scopes and subroutines.

Without the linear order of text lines, which is superimposed upon source code, visual programming as found in models has no linear reading path and can possibly stretch in all directions, left, right, top, bottom, and even down to the next nesting level. While modeling guidelines try to alleviate this by imposing a flow from top-left to bottom-right, engineers struggle with reading visual models as to make sure they are not missing a part of their work to-be-done. Engineers expressed difficulties with reading order both when navigating (see Subsection 4.4) and changing (see Subsection 4.1) models.

For example, when doing readiness testing, all changes in the current release need be covered with tests and no single change must be missed. One participant gave an account of a case where they printed a whole model, put all layers up on a huge wall and worked together on the wall-sized printout to make sure they *"can walk through the complete model and don't miss a block"* (P_7).

Another engineer showed us how she uses a numbering scheme to reduce the spatial complexity of her visual models down to linear reading path, quote: *"this is just [a] little help for myself, we don't have to do this, I add numbers to each blocks, like 7 and 8 and 9, and then 8.1 and 8.2 and deeper down 8.3.1.6, so I can read the model from top to bottom."* (P_{12}) The same motivation, that is introducing linear reading paths, was brought forward by engineers when they described their practices of sharing model patches as PowerPoint decks and or when motivating their preference of textual diffing tools.

Related to this point, when offered an alternative to visual programming engineers seem to prefer non-visual representations. The Rhapsody tool offers engineers an alternative to visual programming which is editing the class diagrams through a tree view and property dialogs. Engineers seemed to prefer this option over visual modeling of UML class diagrams and they even reported that, to their best knowledge, the visual representation of class diagrams is not used by other engineers either.

5.3 On Problem-Specific Needs of Modelers

While model-driven engineering at GM provides the specialized team of code-generation engineers with powerful abstraction to capture domain-specific architectures, it does not empower its end-users, i.e., domain experts and software engineers, to express their own problem-specific languages and type systems.

In general, the visual language of domain experts seems to be much richer and broader than the languages provided by modeling tools. In particular, there seems to be a need for problem-specific "little languages" that can be defined on the fly. Currently, domain experts are unable to create new abstractions that would allow them to achieve productivity gains in algorithm innovation. Neither Simulink, which is largely a visual representation of common coding patterns, nor Rhapsody, by virtual of its limitation to the UML standard's visual languages, offer the ability to define the kind of rich visual languages that we learned about from the domain expert's requirements documents.

Visual programming in Simulink traces its ancestry to circuit diagrams and aims at expressing low-level programming constructs such as conditionals and mathematical operators with the visual language of circuit diagrams. Mathematical operations and conditionals are each represented as single blocks. While this language is visual, it does not seem be an actual abstraction from source code. Even worse, as we learned through our interviews, the level of abstraction seems to be lower than high-level source code. For example, engineers reported that they struggle to introduce abstraction such as nested scopes of variable visibility, enumerators, or refactoring duplicated code into a new method.

Compared to source-based high-level languages, we found that, while model-driven engineering increases the abstraction level of program compilation, it does achieve the same increase in abstraction for program representation. Model-driven engineering provides specialists with the power to build a domain-specific global architecture by customizing the program compilation through code-generation rules. Yet, the "end-users" of model-driven engineering, that is domain experts and software engineers, are left without the power to create their own APIs to address local problem-specific needs in a formal manner.

We are aware that our observations with regard to visual languages and the lack of domain-specific modeling are tied to the technology used in the setting under study, in particular Simulink's visual language and Rhapsody's use of UML. Our findings reflect the state of practice in one organization and are not necessarily representative of the latest state-of-the art in research or even other industries. In particular, domain-specific modeling (DSM) might alleviate the frictions discussed in this subsection [5]. In domain specific modeling the domain experts are empowered to specify the code generation such that modeling concepts map directly to domain concepts rather than computer technology concepts, thus overcoming the limitations of Simulink and UML.

6 Related Work

In this section we discuss related work, namely empirical studies of MDE. For a discussion of the state-of-the-art in, e.g., model diffing or other technologies related to the frictions presented in this paper, readers are advised to refer to recent proceedings of the MODELS conference and its workshops.

Although model-driven engineering claims many potential benefits, it has largely developed without the support of empirical data. There are few reports of empirical evaluations of modeling in the literature. Even fewer discuss human factors and cognitive issues of model-driven engineering, since most empirical studies has been focused technological aspect of MDE or UML in particular.

In parallel to our study, Aranda et.al. investigated the organizational consequences of adopting MDE at the same organization [6]. They interviewed the same participants as our second visit, but while we investigated cognitive issues of technology, driven largely from an individual's perspective, they looked into organizational forms, patterns, and processes of MDE adoption. They found that switching to MDE may disrupt organizational structure, creating morale and power problems. They conclude that the cultural and institutional infrastructure of MDE is underdeveloped and until MDE becomes better established, transitioning organizations need to exert additional adoption efforts.

Most recently Hutchinson et.al. presented their results of a qualitative user study, consisting of semi-structured interviews with 20 engineers in 20 different organizations [7,8]. They identified lessons learned, in particular the importance of complex organizational, managerial and social factors, as opposed to simple technical factors, in the relative success, or failure, of MDE. As an example of organizational change management, the successful deployment of model driven engineering appears to require: a progressive and iterative approach; transparent organizational commitment and motivation; integration with existing organizational processes and a clear business focus.

Mohagheghi and Dehlen presented a study on the impact of MDE on productivity and software quality [9]. Their methodology was a meta-analysis of the literature, selecting 25 papers published in quality conferences and venues between 2000 and 2007. Almost all these papers were experience reports from single projects and most of the papers present results anecdotally. Software processes were reported as being of integral importance in successfully applying MDE, and the importance of suitable tools was reported as of crucial importance. The meta-study also looked for evidence that MDE improves software quality, but the evidence was anecdotal. In conclusion, they suggested that there is a need for more empirical studies evaluating MDE before sufficient data will be available to prove the benefits of its use.

Forward and Lethbridge conducted a survey of 117 engineers to find practitioners' opinions and attitudes towards MDE [10]. In accordance with our findings, the study concludes that model-driven techniques may benefit from features that, synchronize code and models, better traceability between models and code, better modeling capabilities and expressibility the reduce the need for external artifacts. Alas, the survey provides little data on the participant's context, size of their organizations and their adoption-level of MDE. As it seems, only 32% of the participants reported that they generate all or some code from the models. Dobing and Parson discussed the survey to discover commonly-held perceptions which may not hold true in practice [11]. Anda et.al. reported on disadvantages of adopting modeling practices, such as the difficultly of integrating legacy code and models, but found anecdotal advantages of improved traceability [12]. Afonso et.al. wrote about a case study where developers migrate from code-centric to model-centric practices [13].

7 Conclusion

When technologies are introduced, it is often hard to separate myth from reality. To investigate the benefits and challenges of model-driven engineering we performed a field study about the use of model-based design in a large automotive company. We showed how, for one large organization, model-driven engineering is shaped by contextual forces, which seem to be independent of the abstraction chosen to help develop the system. Through this study, which involved interviews with 20 engineers and managers, we identified four forces and five points of friction (as itemized in the introduction). We differentiate between forces that are contextual and external to software modeling technologies and as frictions, which are accidental issues caused by current tooling on software modeling.

As we worked with the data, the contextual forces affecting individuals using modeling became clear. While architectural complexity is well hidden from software engineers, they are still exposed to substantial innate complexity (contextual forces) and often even new accidental complexity (points of frictions in modeling tools). In particular, the representational abstraction of visual modeling languages does not seem to be as broad and rich as the problem-specific visual and formal languages of the domain experts.

We identified three concluding themes that span across many of the identified forces and points of friction, which might be of interest for tool builder and language designers in their future work. They are as follows:

– Engineers seem to prefer the linear reading paths of textual representations over the spatial representation of nested visual models. Both when navigating and changing models as well as when using model diffing. They describe their experience as *"going blind"* and struggling *"to not miss anything."*
– While the MDE tools under study provide specialized code-generation engineers with powerful abstraction, they do not similarly empower its end-users, i.e., domain experts and software engineers, to express their own problem-specific languages and type systems.
– The needs of engineers who are inventing novel algorithms differ from those of engineers who are working on more mature features. Invention is an exploratory activity and its needs, such as instant model changes as runtime, seem not to be well addressed by current modeling tool-chains.

Acknowledgments. This research has been funded by the Canadian Network on Engineering Complex Software Intensive Systems for Automotive Applications (NECSIS). Special thanks to Joe D'Ambrosia and Shige Wang from GM Research, as well as Indira Nurdiani and Jorge Aranda, for their help conducting the interviews. We are grateful to all our study participants for their time and collaboration.

References

1. Brooks, F.P.: No silver bullet: Essence and accidents of software engineering. Computer 20(4), 10–19 (1987)
2. Yin, R.K.: Case study research: design and methods, 3rd edn. Publications. Sage Publications (December 2003)
3. Bentley, J.L.: Programming pearls: Little languages. Communications of the ACM 29(8), 711–721 (1986)
4. Bracha, G.: Pluggable type systems. In: OOPSLA Workshop on Revival of Dynamic Languages (October 2004)
5. Bracha, G.: DSM case studies and examples, `http://www.dsmforum.org/cases.html`
6. Aranda, J., Borici, A., Damian, D.: Transitioning to model-driven development: What is revolutionary, what remains the same? In: MODELS 2012 (2012)
7. Hutchinson, J., Rouncefield, M., Whittle, J.: Model-driven engineering practices in industry. In: Proceeding of the 33rd International Conference on Software Engineering, ICSE 2011, pp. 633–642. ACM, New York (2011)
8. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of MDE in industry. In: Proceeding of the 33rd International Conference on Software Engineering, ICSE 2011, pp. 471–480. ACM, New York (2011)
9. Mohagheghi, P., Dehlen, V.: Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 432–443. Springer, Heidelberg (2008)
10. Forward, A., Lethbridge, T.C.: Problems and opportunities for model-centric versus code-centric software development: a survey of software professionals. In: Proceedings of the 2008 International Workshop on Models in Software Engineering, MiSE 2008, pp. 27–32. ACM, New York (2008)
11. Dobing, B., Parsons, J.: How UML is used. Commun. ACM 49(5), 109–113 (2006)
12. Anda, B., Hansen, K., Gullesen, I., Thorsen, H.: Experiences from introducing UML-based development in a large safety-critical project. Empirical Software Engineering 11, 555–581 (2006)
13. Afonso, M., Vogel, R., Teixeira, J.: From code centric to model centric software engineering: practical case study of mdd infusion in a systems integration company. In: Fourth and Third International Workshop on Model-Based Development of Computer-Based Systems and Model-Based Methodologies for Pervasive and Embedded Software, MBD/MOMPES 2006, 2006, vol. 10, p. 134 (March 2006)
14. Corbin, J.M., Strauss, A.: Grounded theory research: Procedures, canons, and evaluative criteria. Qualitative Sociology 13(1), 3–21 (1990)
15. Adolph, S., Hall, W., Kruchten, P.: Using grounded theory to study the experience of software development. Empirical Software Engineering 16(4), 487–513 (2011)
16. Adolph, S., Hall, W., Kruchten, P.: Wide gap amongst developers' perception of the importance of UML tools, developereye study reveals. Technical report, Developer Eye.com (April 2005)

A Model-Driven Approach to Support Engineering Changes in Industrial Robotics Software

Yu Sun[1], Jeff Gray[2], Karlheinz Bulheller[3], and Nicolaus von Baillou[3]

[1] University of Alabama at Birmingham, Birmingham AL 35294
yusun@cis.uab.edu
[2] University of Alabama, Tuscaloosa, AL 35401
gray@cs.ua.edu
[3] Bulheller Consulting, Inc., Tuscaloosa, AL 35406
bulheller+partner ingenieure, Boeblingen, Germany
{k.bulheller,n.vonbaillou}@bulheller-consulting.com

Abstract. Software development has improved greatly over the past decades with the introduction of new programming languages and tools. However, software development in the context of industrial robotics is dominated by practices that require attention to low-level accidental complexities related to the solution space of a particular domain. Most vendor-specific robotics platforms force the developer to be concerned with many low-level implementation details, which presents a maintenance challenge in the context of making engineering changes to the robotics solution. Additionally, satisfying the timing requirements across the platforms of multiple robot vendors represents an additional challenge. We introduce our work using Domain-Specific Modeling to support the control of industrial robots using models that are at a higher level of abstraction than traditional robot programming languages. Our modeling approach assists robotics developers to plan the schedule, validate timing requirements, optimize robot control, handle engineering changes, and support multiple platforms.

Keywords: Domain-Specific Modeling, Robotics, Software Maintenance, Digital Factory, Digital Master.

1 Challenges in Industrial Robotics

Industrial robots have been applied widely in various domains to perform different tasks [4][9], such as welding robots used in the automobile industry, or assembly robots used in manufacturing factories. Robots are often controlled and programmed using textual and imperative robot programming languages that are customized environments from each robot vendor. Even with the implementation of a digital factory (i.e., a virtual representation of the manufacturing process and facility), most robotics languages are domain-specific languages designed by specific robot vendors (e.g., the KUKA [14] robot programming language, and RAPID [16] from ABB [15]), they are still at a low-level of abstraction. This requires a great deal of knowledge about implementation and configuration details (e.g., the coordinates of the movement destination, the speed and acceleration of the movement, the ports to write and read data), which presents a host of challenges in robotics software development and maintenance. Based on our 20+ years of automotive industry experience, the following paragraphs describe what we have observed as the key challenges in supporting engineering changes in industrial robotics software (in particular, in an automotive factory context using digital factory methodologies).

Challenge 1 – The Complexity of Adapting Engineering Changes Across Robotics Software Solutions. Similar to other types of software development, software evolution is also inevitable in robotics development. For instance, working on different types of products and work tasks, robots need to be modified frequently with different hardware parameters, environment configurations, and more importantly a different sequence of actions needed to address a new requirement on the assembly line. Changing and evolving the robot programs to adapt to new requirements is challenging, particularly when performing the changes on large-scale heterogeneous robotics systems. To make an engineering change across a specific cell of an assembly line, robotics programmers need to search through a collection of robot programs manually, locate the correct location of low-level configuration information, and make the correct modification corresponding to the new requirement.

Challenge 2 – The Difficulty of Satisfying Timing Requirements and Optimizing Action Schedules for Cycle Time Optimization. The correct timing and scheduling configuration on a robotics system plays an essential role in multi-robot coordination. Consider the category of welding robots as an example, where each robot must finish its own task on time and ensure the robot next to it has the needed parts within a certain time target. Failure to meet a task deadline in the prescribed time will either cause unnecessary delays or trigger collision conflicts among different robots. In order to minimize the duration of completing a certain task, an optimized schedule for each robot is required to avoid unnecessary delays. However, due to a lack of native support for time in most robot programming languages, satisfying the correct timing requirements in robotics development has become a tedious, time-consuming and error-prone task that requires much manual tweaking and refined intuition in order to elaborate a successful implementation. The most commonly used approach in practice is to plan the schedule manually, and then hand it to robot developers who then implement the schedule plan manually. There are several current well-known automotive factories that still use standard spreadsheets for determining such timing considerations. Developers must write robot programs based on the timing requirements, and test the program in an ad hoc manner to obtain various timing measurements during commissioning. If the measurement indicates the violation of specific timing requirements, changes must be made either to the schedule or the robot programs. This process iterates until all the timing requirements are satisfied. This type of scenario, based on manual and iterative refinements, is ripe for application of model-driven techniques.

Challenge 3 – The Challenge of Supporting Multiple Platforms. With multiple robot manufactures throughout a manufacturing facility, it is often necessary to swap out robots from different vendors at different stations in a manufacturing cell. However, if each vendor uses a different robot programming language, the same task will have to be programmed multiple times in different languages. This requires much redundancy and maintenance of multiple programs for the same task – a situation that is fertile for creating software failures. A desired capability is to be able to describe the intellectual property associated with a robot task at a level that can be maintained and preserved across current vendors. Such a capability would also protect against obsolescence and allow integration of future robot platforms that may later emerge.

We have designed a modeling, planning, and code generation tool suite that addresses the needs of these three challenges. This tool, called Automax, serves as the future input to our existing robotics optimization solution (called Robmax) that is currently deployed on over 3,000 industrial robots in "Body in White"[1] shops at manufacturing facilities in the USA and Europe. The main objectives of our work described in this paper are: 1) to raise the level of abstraction in robotics programming and hide the low-level implementation details. This is done by capturing key domain concepts and constructing code frameworks and libraries, in order to facilitate multiple types of engineering changes on the manufacturing line; 2) to combine timing and scheduling information for robot programs, and provide timing analysis to ease the process of satisfying timing requirements; 3) to build a common representation for expressing robot control, from which automatic generation to specific vendor platforms is possible.

An overview of the proposed solution will be given in Section 2, followed by the illustration of each key component in the solution from Section 3 to Section 5. Section 6 summarizes the related work and Section 7 offers concluding remarks.

2 Automax Overview

Our solution to address the key challenges presented in Section 1 is to use Domain-Specific Modeling (DSM) [3] to support robotics development. Raising the level of abstraction from programming languages to modeling languages has been shown to be an effective approach to attack the increasing complexity of software systems [1]. Domain-Specific Modeling Languages (DSMLs) [2] assist domain experts in focusing on the level of abstraction relevant to their problem space by providing notations and constructs tailored specifically to that domain, while removing the accidental concerns of a specific solution space. DSMLs help to represent the solution of the problem domain and reduce miscommunication between stakeholders by providing common abstractions and notations.

Figure 1 is an overview of our solution. The core part of the solution is a graphical DSML defined by a metamodel specifically for the industrial robotics domain. Our modeling language captures the key configurations for robots, all types of actions an industrial automotive manufacturing robot can perform, as well as the scheduling and timing information. Compared with traditional robot programming languages (e.g., KUKA), this DSML is at a higher level of abstraction by hiding many low-level

[1] Body In White refers to a phase of automotive manufacturing when the metal body of the car has been welded together, just before the addition of attached structures (e.g., doors) and prior to painting.

implementation details and extracting patterned program code fragments as abstract model concepts, so that users can specify the robot models using direct domain concepts.

Instead of creating models using the DSML manually from scratch, users can start with the planning of the robot system timing requirements in the planner, followed by generating the base robot model automatically. On the other hand, with the existing robot control code, Automax's future vision supports reverse engineering of the source code to generate the models as well as the timing information in the planner. Users can then operate on the models directly and make necessary changes. The actual implementation code can be generated automatically from the models for different platforms.

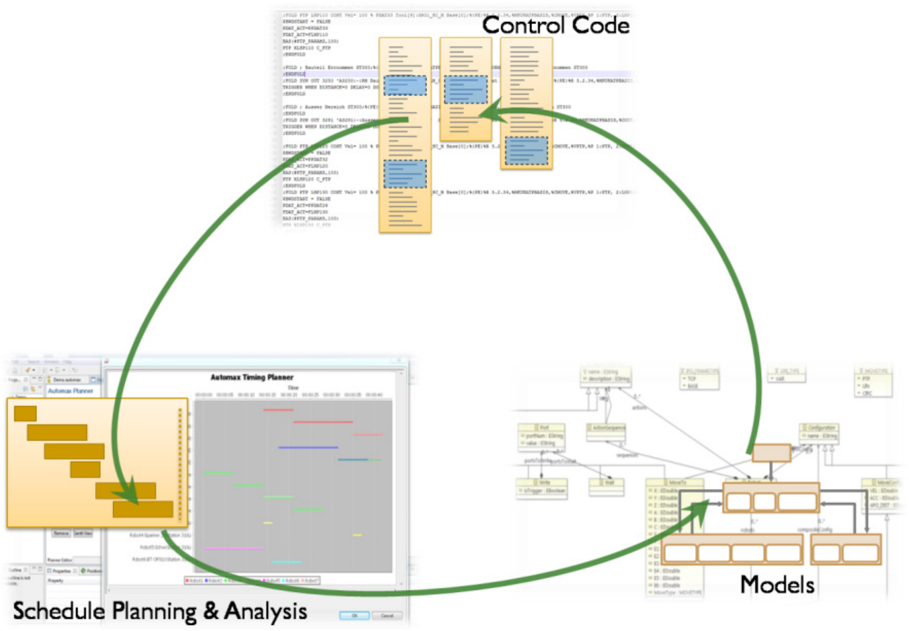


Fig. 1. Overview of the Automax solution

With models as the first-class entities to program robots, any engineering changes can be realized by modifying the robot models and re-generating the code, which is an alternative to changing the code manually across multiple robot programs at a lower level of abstraction. Additionally, because the timing and scheduling information has been incorporated into the robot models, the model provides a direct input to the scheduler, so that the timing can be estimated and validated. Moreover, the robot models are platform-independent, which enables multiple code generators for different robotics vendor platforms.

Being different from the traditional top-down model-driven code generation framework from models to code, Automax connects system planning and analysis, system models, and implementation code together, and supports an iterative development process from planning to models, models to code, and code back to analysis and planning. The goal is to enable users to create models rapidly with integrated timing requirements, directly generate implementation code and measure the performance, and more seamlessly make engineering changes on models and re-generate code.

Our solution with Automax has been implemented as a modeling tool in Eclipse using multiple Eclipse Modeling Projects [17], which provides a unified set of modeling frameworks, tooling, and standards implementations on the evolution and promotion of model-driven development technologies within the Eclipse community. Automax provides a schedule planner, robot modeling editor, code generator, and a number of tools to facilitate the timing and scheduling design, validation, and optimization. The next sections will present the main components of our Automax solution.

3 Using Models to Facilitate Engineering Changes

The engineering changes in robotics may emerge from the need for a new group of robots to collaborate with each other, a new sequence of actions to perform, or a new set of configuration parameters for each robot. The main challenge of handling these changes comes from locating the correct parts of the source code and making the needed changes. To raise the level of abstraction, we analyzed the source code of existing robot programs currently in use within an automotive assembly line and identified the key concepts and relationships in the robotics domain through a manual reverse-engineering process (i.e., identify the functions or statement blocks in the source code and extract them as unique and reusable modeling concepts). A DSML was defined using these concepts. Figure 2 shows the core part of the metamodel used to define the Automax DSML, with the model attributes and some extra data types elided. A robotics configuration can include multiple robots. Each robot can perform various types of sequential actions, such as moving, welding, opening/closing grippers, checking pivot equipment and halting. Corresponding attributes are available for direct configuration for each action. The Composite command pattern can be used to include a group of actions. Special configurations (e.g., movement configuration, tooling configuration) are available for separate definition and shared by the action commands. Advanced flow control mechanisms such as repetition and decision-making are not defined in the metamodel for the purpose of hiding the low-level programming details.

Figure 3 shows an excerpt of a robot model instance. Users can construct a group of robots, configure the actions and parameters in the editor, and specify the sequence of actions using arrows. Based on this DSML, any engineering changes defined in the Digital Master of the product can be implemented by modifying the model instances to adopt the changes in the manufacturing process. For example, a group of robots can be changed by directly adding or removing robots; the actions for each robot can be updated by editing the action command model elements; the attribute editor allows the changes on parameters; and the sequence of actions can be changed by redirecting the arrows. Code generators have been built to generate the actual implementation code for several popular robot vendors. Therefore, the changes on models can be realized immediately by re-generating the code from models, as shown in Figure 4.

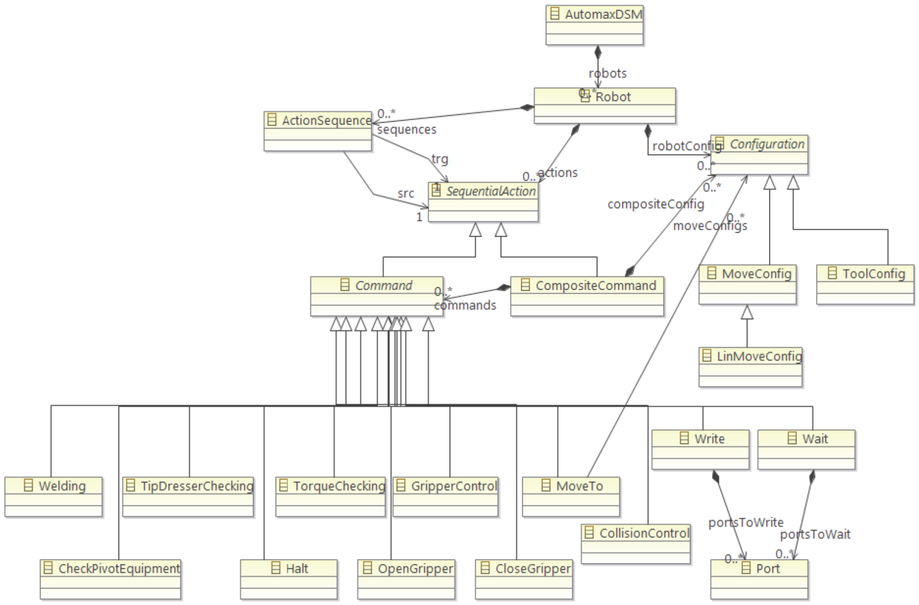


Fig. 2. The core Automax metamodel

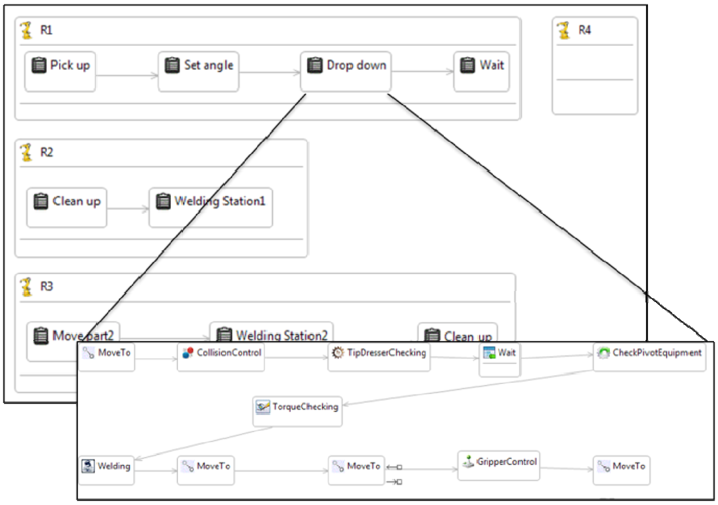


Fig. 3. An excerpt of a robot model

Engineering changes in the robotics domain often involve a large number of small changes on the parameters and action configurations across all robots. In order to assist users in tracking and checking all the changes, a special change view has been embedded in the robot modeling editor to display all types of changes occurring in the editor (i.e., new elements, removed elements, or updated attributes). As the example shown in Figure 5, the “Position Changes” view illustrates the changes to the destination location for each action. There are new locations being added, and old locations being updated or removed. Each change record in the view is associated with the editor, so that clicking on any change highlights the corresponding location or model element in the editor.

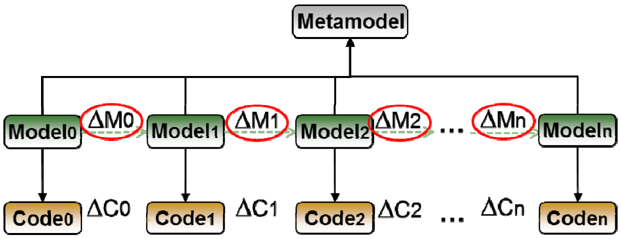
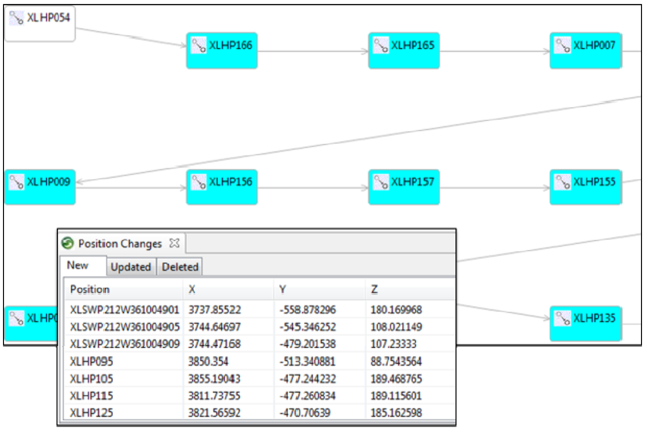


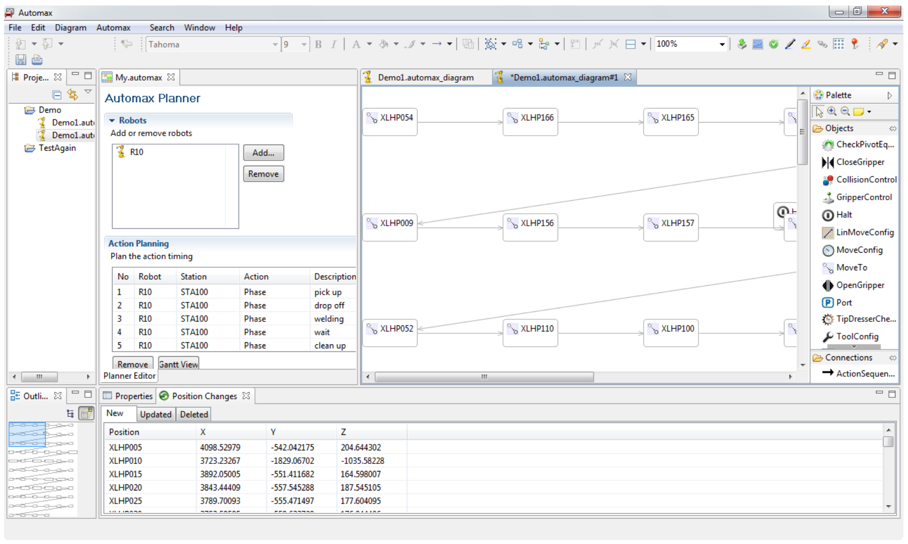
Fig. 4. Changing models triggers code changes



Position	X	Y	Z
XLSWP212W361004901	3737.85522	-558.878296	180.169968
XLSWP212W361004905	3744.64697	-545.346252	108.021149
XLSWP212W361004909	3744.47168	-479.201538	107.23333
XLHP095	3850.354	-513.340881	88.7543564
XLHP105	3855.19043	-477.244232	189.468765
XLHP115	3811.73755	-477.260834	189.115601
XLHP125	3821.56592	-470.70639	185.162598

Fig. 5. The change view and change highlight

From a real digital master of a robot cell in a manufacturing center, we can use Automax to trace the lifecycle of an engineering change from the process planning phase to robot code generation. As shown in Figure 6, a process planner (who is not a computer scientist and does not know how to develop robotics software) can provide an action planning strategy (Automax planner section) that generates high-level models representing key concepts of a robot cell at a manufacturing plant (right-side of Figure 6, where the Palette represents the visualized domain concepts available from the Automax metamodel definition). From this model, robot code can be generated within seconds, from what previously would take several weeks. From our experience, this new capability allows the exploration of the design alternatives in a way that is very productive (within seconds) and accurate (the maturity of the code generators produces code that is always more reliable than human-generated code), allowing engineers the flexibility to understand design tradeoffs in a manner that is currently not possible due to the time needed for manual adaptation.



Automax Planner

Robots

Add or remove robots

R10

Add... Remove

Action Planning

Plan the action timing

No	Robot	Station	Action	Description
1	R10	STA100	Phase	pick up
2	R10	STA100	Phase	drop off
3	R10	STA100	Phase	welding
4	R10	STA100	Phase	wait
5	R10	STA100	Phase	clean up

Remove Gantt View

Planner Editor

New Updated Deleted

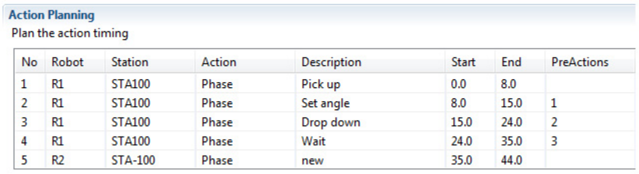
Position	X	Y	Z
XLHP005	4098.52979	-542.042175	204.644302
XLHP010	3723.23267	-1829.06702	-1035.58228
XLHP015	3892.05005	-551.411682	164.598007
XLHP020	3843.44409	-557.545288	187.545105
XLHP025	3789.70093	-555.471497	177.604095

Fig. 6. The integration of planning and high-level robot control in Automax

The basic implementation of this tool is based on the Eclipse Modeling environment. The metamodel is defined using EMF [13][18] and the editing environment is generated using GMF [19]. Building and maintaining a modeling tool with GMF is not an easy task, which requires six individual models that are highly dependent on each other and all need to be in sync with each other. Instead of creating these models manually one by one, we use the Eugenia tool [20], which considerably sped up the creation of the graphical DSML editors. The Eugenia tool essentially reduces development and maintenance effort down to one model plus some optional, separate customization information.

4 Incorporating Timing Requirements to Optimize Schedules

The separation of timing and scheduling information from the traditional robot control programs in process planning makes it difficult to satisfy and validate timing requirements. The traditional robotics development requires a timing plan on paper (or in Excel) and then a program that meets the timing requirements. Traditional robotics development may involve many iterations to test the performance of the completed programs, validate the timing requirements and make necessary changes. Using DSMLs, it is possible to specify multiple views or multiple aspects for a certain domain, which enables us to define the timing and scheduling information together with the robot configurations in the model to better analyze the timing status. In Automax, users begin configuring robots with a schedule plan in a customized editor, as shown in Figure 7. In the planner, a sequence of tasks is defined with the information about the involved robots, start time, end time, and prerequisite tasks. This planner serves as the high-level description about the tasks to be accomplished and the desired timing requirements. The scheduling information is part of the DSML (i.e., the timing attributes in some of the action command) and thus saved as part of the robot model.


Action Planning
Plan the action timing

No	Robot	Station	Action	Description	Start	End	PreActions
1	R1	STA100	Phase	Pick up	0.0	8.0	
2	R1	STA100	Phase	Set angle	8.0	15.0	1
3	R1	STA100	Phase	Drop down	15.0	24.0	2
4	R1	STA100	Phase	Wait	24.0	35.0	3
5	R2	STA-100	Phase	new	35.0	44.0	

Fig. 7. The schedule plan editor

From a complete plan, users can generate an initial model automatically that contains all the needed robots and the high-level actions. The actions have been ordered correctly based on the prerequisite actions defined in the planner. From this initial model, users can fill each high-level action (i.e., the composite action which includes a set of atomic actions) with the specific action commands needed. The transformation from the schedule model to robot model is an endogenous model transformation, which means we start with creating the schedule model elements, and then the tool will enrich the model with the robot information. The transformed model can always be edited in the schedule planner directly, and the information will always be synchronized. The model excerpt shown in Figure 3 has the fully configured action commands based on the generated model from the planner. With a complete robot model, the total duration of each high-level action can be estimated based on the included atomic actions through a computation engine. By comparing the estimated time and the planned schedule, users can determine directly if the current action configuration can satisfy the timing requirements, as shown in Figure 8. Each blue bar in the figure represents a task to finish with its start and end time. The inner green bar reflects the estimated duration based on the current actions included. The chart can be shown during editing time, so that users can modify the plan or change the actions during a schedule violation or optimization.

Besides the timing analysis, special features have been implemented to optimize the schedule in regards to cycle time. For instance, in manufacturing process development, movement is the most typical and frequent action that a robot performs. A sequence of movement steps is always needed to reach a desired location. Without an optimized sequence, it may cause unnecessary delays. Thus, a feature has been implemented in Automax to identify all the movement steps in a robot automatically, and re-order the sequence of these steps using a shortest path algorithm. Because the location configuration is captured in the model elements, this type of optimization can be done on models directly, which we have observed to be easier than realizing the same optimization on robot code through parsing and program transformation processes. The timing planner editor, analysis viewer, and optimizer are realized as Eclipse plug-ins to the Automax environment. Models are the direct artifacts to be operated by the plug-ins, and provide a convenient programming and exchange interface.

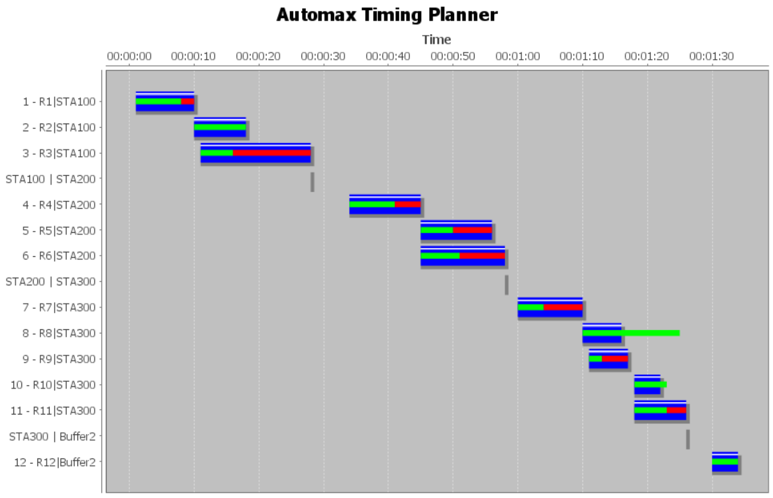
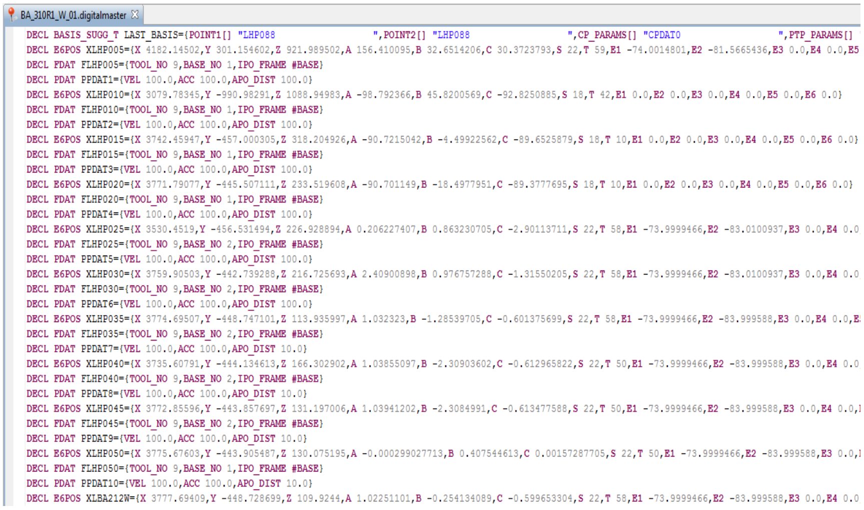


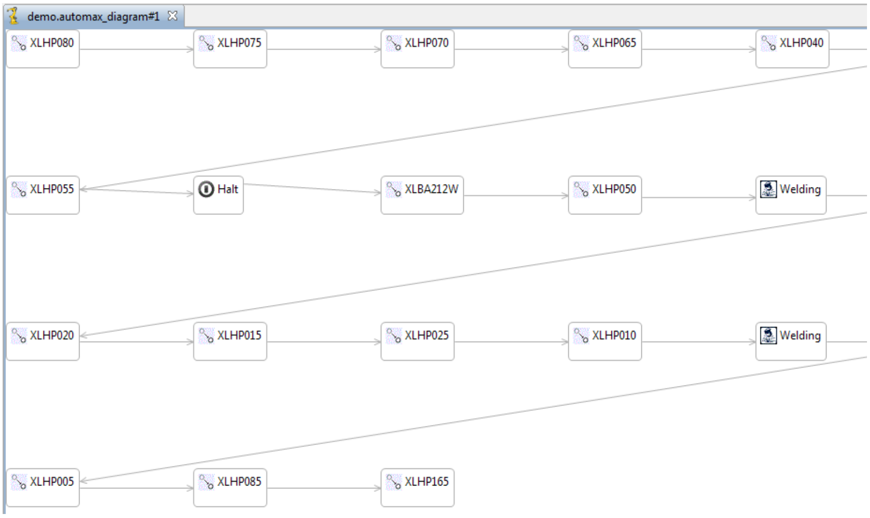
Fig. 8. The chart showing the current timing status

With the implementation of the timing planner and optimization features, benefits can be seen from using models to do robotics development. However, the large amount of legacy code has already been used in production, which cannot be directly applied in the Automax modeling environment. In order to support the legacy robot code, we are further enhancing the tool to enable reverse engineering of the existing KUKA robot code. As the first version, a subset of the KUKA grammar has been specified in Xtext [21], which considers KUKA as a textual DSL. The grammar is also mapped to the Automax metamodel, so that with the generated text editor using Xtext, the legacy robot code can be parsed and converted into Automax models directly, as shown in Figure 9. From the models, users can perform the typical model editing operations, analyze the timing, apply optimizations, and then re-generate a new version of the code. Changes can also be made directly to code, which can be reverse engineered again and injected back into models.


BA_310R1_W_01.digitalmaster

```
DECL BASIS_SUGG_T LAST_BASIS={POINT1[] "LHP088                    ",POINT2[] "LHP088                    ",CP_PARAMS[] "CPDAT0                    ",PTP_PARAMS[]
DECL E6POS XLHP005={X 4182.14502,Y 301.154602,Z 921.989502,A 156.410095,B 32.6514206,C 30.3723793,S 22,T 59,E1 -74.0014801,E2 -81.5665436,E3 0.0,E4 0.0,E5
DECL FDAT FLHP005={TOOL_NO 9,BASE_NO 1,IPO_FRAME #BASE}
DECL PDAT PPDAT1={VEL 100.0,ACC 100.0,APO_DIST 100.0}
DECL E6POS XLHP010={X 3079.78345,Y -990.98291,Z 1088.94983,A -98.792366,B 45.8200569,C -92.8250885,S 18,T 42,E1 0.0,E2 0.0,E3 0.0,E4 0.0,E5 0.0,E6 0.0}
DECL FDAT FLHP010={TOOL_NO 9,BASE_NO 1,IPO_FRAME #BASE}
DECL PDAT PPDAT2={VEL 100.0,ACC 100.0,APO_DIST 100.0}
DECL E6POS XLHP015={X 3742.45947,Y -457.000305,Z 318.204926,A -90.7215042,B -4.49922562,C -89.6525879,S 18,T 10,E1 0.0,E2 0.0,E3 0.0,E4 0.0,E5 0.0,E6 0.0}
DECL FDAT FLHP015={TOOL_NO 9,BASE_NO 1,IPO_FRAME #BASE}
DECL PDAT PPDAT3={VEL 100.0,ACC 100.0,APO_DIST 100.0}
DECL E6POS XLHP020={X 3771.79077,Y -445.507111,Z 233.519608,A -90.701149,B -18.4977951,C -89.3777695,S 18,T 10,E1 0.0,E2 0.0,E3 0.0,E4 0.0,E5 0.0,E6 0.0}
DECL FDAT FLHP020={TOOL_NO 9,BASE_NO 1,IPO_FRAME #BASE}
DECL PDAT PPDAT4={VEL 100.0,ACC 100.0,APO_DIST 100.0}
DECL E6POS XLHP025={X 3530.4519,Y -456.531494,Z 226.928894,A 0.206227407,B 0.863230705,C -2.90113711,S 22,T 58,E1 -73.9999466,E2 -83.0100937,E3 0.0,E4 0.0
DECL FDAT FLHP025={TOOL_NO 9,BASE_NO 2,IPO_FRAME #BASE}
DECL PDAT PPDAT5={VEL 100.0,ACC 100.0,APO_DIST 100.0}
DECL E6POS XLHP030={X 3759.90503,Y -442.739288,Z 216.725693,A 2.40900898,B 0.976757288,C -1.31550205,S 22,T 58,E1 -73.9999466,E2 -83.0100937,E3 0.0,E4 0.0
DECL FDAT FLHP030={TOOL_NO 9,BASE_NO 2,IPO_FRAME #BASE}
DECL PDAT PPDAT6={VEL 100.0,ACC 100.0,APO_DIST 100.0}
DECL E6POS XLHP035={X 3774.69507,Y -448.747101,Z 113.935997,A 1.032323,B -1.28539705,C -0.601375699,S 22,T 58,E1 -73.9999466,E2 -83.999588,E3 0.0,E4 0.0,E
DECL FDAT FLHP035={TOOL_NO 9,BASE_NO 2,IPO_FRAME #BASE}
DECL PDAT PPDAT7={VEL 100.0,ACC 100.0,APO_DIST 10.0}
DECL E6POS XLHP040={X 3735.60791,Y -444.134613,Z 166.302902,A 1.03855097,B -2.30903602,C -0.612965822,S 22,T 50,E1 -73.9999466,E2 -83.999588,E3 0.0,E4 0.0
DECL FDAT FLHP040={TOOL_NO 9,BASE_NO 2,IPO_FRAME #BASE}
DECL PDAT PPDAT8={VEL 100.0,ACC 100.0,APO_DIST 10.0}
DECL E6POS XLHP045={X 3772.85596,Y -443.857697,Z 131.197006,A 1.03941202,B -2.3084991,C -0.613477588,S 22,T 50,E1 -73.9999466,E2 -83.999588,E3 0.0,E4 0.0,
DECL FDAT FLHP045={TOOL_NO 9,BASE_NO 2,IPO_FRAME #BASE}
DECL PDAT PPDAT9={VEL 100.0,ACC 100.0,APO_DIST 10.0}
DECL E6POS XLHP050={X 3775.67603,Y -443.905487,Z 130.075195,A -0.000299027713,B 0.407544613,C 0.00157287705,S 22,T 50,E1 -73.9999466,E2 -83.999588,E3 0.0,
DECL FDAT FLHP050={TOOL_NO 9,BASE_NO 1,IPO_FRAME #BASE}
DECL PDAT PPDAT10={VEL 100.0,ACC 100.0,APO_DIST 10.0}
DECL E6POS XLBA212W={X 3777.69409,Y -448.728699,Z 109.9244,A 1.02251101,B -0.254134089,C -0.599653304,S 22,T 58,E1 -73.9999466,E2 -83.999588,E3 0.0,E4 0.0
```

a) Original manually created robot code for a particular welding task



b) Corresponding Automax model representation

Fig. 9. Converting robot code to an Automax model

5 Applying Diverse Code Generators to Support Multiple Platforms

The robot model only contains platform-independent information, so it can be used to generate code for different implementation platforms (i.e., different robot vendors). When designing a code generator, a preferred practice is to design a domain framework, which contains common functionality so that a minimum amount of code needs to be generated from the models [8]. Thus, we identified the code framework used in several robotics languages (e.g., environment initialization, PLC and tool communication working environment clean up), which are fixed and used in many robotics tasks. The generated code realizes the specific sequence of actions and the configuration of each action. A separation of the generated code from the code framework reduces the complexity of code generation specification, which we believe leads to a more maintainable architecture.

6 Related Works

Angerer et al. introduced an Object-Oriented (OO) framework for modeling industrial robotics applications to improve robotics development and maintenance [5]. By analyzing the existing low-level and imperative robot programming languages, a set of robotics APIs were designed across 70 classes, which covers the concepts to model geometric relations (e.g., Frame, SpatialObject, PhysicalObject), device and control (e.g., Device, Joint, Manipulator), and commands (e.g., Action, Trigger). These APIs can be integrated with the traditional OO programming languages and executed through a special library to map the APIs to the original low-level code [6]. The main benefit of having an OO robotics framework is that developers can utilize OO design and use OO programming languages to improve the robotics development and maintenance process. However, when moving the traditional robotics language to an OO language, it becomes a general-purpose language (with domain concepts summarized as APIs); thus, this solution is not at the same level of abstraction as our Automax modeling solution. Furthermore, the timing requirements and multiple platform support have not been considered in their approach.

Robmann et al. presented another robotics development approach from a different direction [7]. The context of their approach is the existence of an online (real execution) and an offline (simulation) robot system. They designed a new system called “ProDemo” to improve the setup (i.e., configuration and programming) of both systems. There are two main components in ProDemo: 1) Modeling by demonstration provides a new approach to build 3D models for the simulation. Users teach the robots about certain behaviors by directly demonstrating the process. 2) A visual programming robot control language can be used to program the control flow of robots, which enables users to program the robot in a graphical and more intuitive way. However, this system only focuses on the control specification of robots, without considering the timing and scheduling requirements. Additionally, the visual programming language is in fact at the same level of abstraction as the traditional textual robot language. It only changes the concrete syntax, without raising the level of abstraction by hiding the low-level implementation details.

There are many usage examples of DSMLs in different domains to improve software development. For instance, a similar modeling approach has been applied to create a time-triggered system for electrical cars that support different communication protocols (e.g., Flexray, CAN bus) [10]. In the area of high-performance computing, Jacob et al. designed and implemented a modeling framework called PPmodel to assist programmers in separating the core computation from the details of a specific parallel architecture, identifying and retargeting the parallel section of a program to execute in a different platform [11]. Another example is the application of model-driven engineering and a supporting tool infrastructure for the industrial process control domain, done by Lukman et al. [12]. The work described in this paper distinguishes itself from the following aspects: 1) it focuses on the robotics domain; 2) non-functional requirements (i.e., timing and scheduling requirements) have been integrated with domain concepts and reflected in the generated code; 3) performance analysis and optimization can be made to models during editing time; 4) the same metamodel is mapped to both the textual and the graphical DSL so that the two formats can be interchanged with each other; 5) an iterative development approach and reverse engineering are both supported in our framework.

7 Conclusions and Future Work

In this paper, we presented the concept of applying DSM to the robotics domain to handle the challenges of industrial robotics development. Our solution is based on a high-level DSML designed specifically for configuring robots so that users can model the robot control using direct domain concepts, and generate code for different platforms automatically. The code generation enables users to only change models to adapt engineering changes, without manually evolving the implementation. We also integrated the timing and scheduling requirements into the modeling language, which eases the schedule planning, validates the timing requirements, and optimizes the schedule. Our Robmax framework for process optimization system is used in automobile factories in Europe and North America. The new Automax modeling suite serves as the input to Robmax and has demonstrated improved advantages over traditional robot programming in terms of the its ability to facilitate engineering changes that crosscut much of the boundaries of the lower level robotics code. Figure 10 shows the integration point for Automax and Robmax.

One of the main research directions in the future is to extend the grammar used in the reverse engineering so that it can support the complete integration of legacy code from past robot programs. Currently, our solution supports a subset of the KUKA robot programming grammar that is related with the data-centric configurations. The challenge of supporting the full grammar is how to map every detail of the language to the metamodel. Although it is possible to extend the metamodel to fit the complete language, it will inevitably lower the level of abstraction and undermine the benefits of using DSMLs. Thus, the ideal situation would be to have the capability of parsing all the legacy code, but generating a model that conforms to a metamodel that is still at a higher-level of abstraction without covering each language detail used in the legacy code. On the other hand, the optimization and analysis are currently dependent on the metamodel definition and implemented as separate plug-ins. This dependency brings about problems with metamodel changes. Therefore, it would be very useful to investigate how to integrate the semantics of optimization and analysis into the metamodel and then automatically generate these functions based on the metamodel.

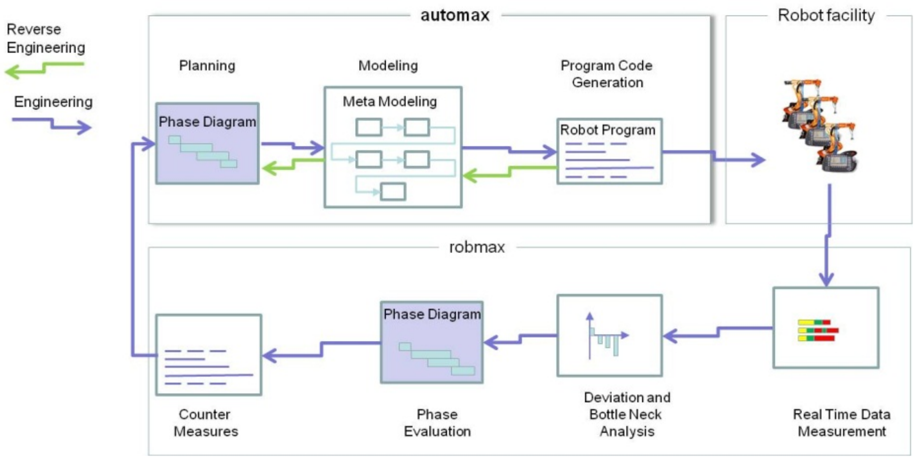


Fig. 10. Automax interaction with Robmax (a highly successful automation efficiency solution already deployed across 3,000 robots in Europe and North America)

Acknowledgement. This work is supported by NSF CAREER award CCF-1052616.

References

1. Schmidt, D.: Model-Driven Engineering. IEEE Computer 39(2), 25–32 (2006)
2. Lédeczi, Á., Bakay, Á., Maróti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing Domain-Specific Design Environments. IEEE Computer 34(11), 44–51 (2001)
3. Gray, J., Tolvanen, J., Kelly, S., Gokhale, A., Neema, S., Sprinkle, J.: Domain-Specific Modeling. In: Handbook of Dynamic System Modeling, ch. 7, pp. 7.1–7.20. CRC Press (2007)
4. Brogardh, T.: Present and Future Robot Control Development – An Industrial Perspective. Annual Reviews in Control 31(1), 69–79 (2007)
5. Angerer, A., Hoffmann, A., Schierl, A., Vistein, M., Reif, W.: The Robotics API: An Object-Oriented Framework for Modeling Industrial Robotics Applications. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, Taipei, pp. 4036–4041 (September 2010)
6. Muhe, H., Angerer, A., Hoffmann, A., Reif, W.: On Reverse-engineering the KUKA Robot Language. In: International Workshop on Domain-Specific Languages and Models for Robotic Systems, Taipei, pp. 11–17 (September 2010)
7. Robmann, J., Ruf, H., Schlette, C.: Model-Based Programming "by Demonstration" – Fast Setup of Robot Systems (ProDemo). Advances in Robotics Research 5, 159–168 (2010)
8. Kelly, S., Tolvanen, J.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley (2008)
9. Freund, E., Rossmann, H., Schluse, M., Schlette, C.: Using Supervisory Control Methods for Model Based Control of Multi-agent Systems. In: IEEE Conference on Robotics, Automation and Mechatronics, Singapore, pp. 649–654 (December 2004)
10. Sun, Y., Wienands, C., Felser, M.: Apply Model-Driven Design and Development to Distributed Time-Triggered Systems. In: International Conference on Engineering and Meta-Engineering, Orlando, FL, pp. 557–563 (March 2011)

11. Jacob, F., Gray, J., Bangalore, P., Sun, Y.: A Platform-Independent Tool for Modeling Parallel Programs. In: 49th Annual ACM Southeast Conference, Kennesaw, GA, pp. 138–143 (March 2011)
12. Lukman, T., Godena, G., Gray, J., Strmcnik, S.: Model-Driven Engineering of Industrial Control Process Applications. In: IEEE International Conference on Emerging Technologies and Factory Automation, Bilbao, Spain (September 2010)
13. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.: Eclipse Modeling Framework. Addison-Wesley (2004)
14. KUKA Robots (2012), `http://www.kukarobotics.com/`
15. ABB Group (2012), `http://www.abb.com/`
16. RAPID Reference Manual (2012), `http://rab.ict.pwr.wroc.pl/irb1400/overviewrev1.pdf`
17. Eclipse Modeling Project, EMP (2012), `http://www.eclipse.org/modeling/`
18. Eclipse Modeling Framework, EMF (2012), `http://www.eclipse.org/modeling/emf/`
19. Graphical Modeling Framework, GMF (2012), `http://www.eclipse.org/gmf/`
20. Eugenia (2012), `http://www.eclipse.org/gmt/epsilon/doc/eugenia/`
21. Xtext (2012), `http://www.eclipse.org/Xtext/`

Managing Related Models in Vehicle Control Software Development

Rick Salay[1], Shige Wang[2], and Vivien Suen[1]

[1] Department of Computer Science, University of Toronto, Toronto, Canada
{rsalay,vsuen}@cs.toronto.edu
[2] General Motors Global Research and Development, Warren, Michigan, USA
shige.wang@gm.com

Abstract. Model management is critical for large software-intensive system development as it ensures the consistency and correctness of the models that are separately developed but interrelated. It is especially crucial when the models are acquired from different sources and evolve frequently. Traditional approaches to model management in vehicle control software development rely on information examination guarded by a rigorous development process, which requires a high-level of knowledge and may be less effective than is desirable. To address this issue, we investigate the applicability of the macromodel concept – a formal method for the specification of model relationships – to model management of vehicle control system development. Through studying some representative relationships, we build a macromodel based management method and demonstrate its effectiveness using the flow diagrams in a functional architecture model from industry.

1 Introduction

Model-based technologies have been adopted increasingly in the development of vehicle control systems to facilitate the activities of design, analysis, implementation, integration, and validation. The key concept that enables model-based technologies is the adoption of models for expressing all artifacts in the development process. In development of complex vehicle control systems, models represent the design of different system elements and properties, and are created, manipulated, and maintained by multiple designers, design groups, and organizations, each of which may work on and is responsible for activities in a specific engineering discipline. To ensure the system functional correctness of final products, these resultant models must be consistent. Such consistency means that the functions represented by these models integrally yield the correct system-level behaviours, which in turn requires that the involved teams in the development, including customers, designers, developers, project managers, etc., communicate unambiguously and collaborate closely. Given that people working in vehicle control system development typically have different technical backgrounds and expertise, sometimes working in different geographical locations, unambiguous communication and close collaboration through traditional approaches based on natural language (e.g. project documentation and face-to-face meetings) is extremely challenging, if not impossible, in practice.

The use of models often introduces accidental complexities because models in a large collection are usually related and need to be collectively managed. To this end, the discipline of *model management* provides a set of techniques to address the management need. In current industry practices, model management is done in an informal manner, as a process activity with assistance of development tools. The model relations are typically captured informally as the relations among process steps and communications among development teams. As an example, some control development team may define control algorithms as control models in Simulink/Stateflow while its software counterpart defines the software design as a UML model using IBM Rhapsody. Engineers are assigned ownership of these models and are responsible for the consistency of related models following a change management process with assistance of a change management tool. Although effective, this practice has been experiencing issues with tracing the root cause of an error in final product because the responsible engineer may not have sufficient information to reason about the model relationships. This is especially true for "refinement" or "derivation" type relationships that must preserve some modeling properties but allow significant representation and semantic alterations to the original model.

To address such model management challenges, we have explored the use of the *macromodel* concept to build the model management methodology for specifying and managing related models [9]. A macromodel models a collection of related models, with their relationships formally captured as mappings and constraints. With a macromodel defined, any change made to a model in the collection can be checked formally using techniques such as logic inference rules and constraint satisfaction to determine the existence of inconsistency. Therefore, model management using macromodels can detect, and repair automatically if allowed, inconsistencies between models through formal expressions of model relationships.

In this paper, we present the results of an initial case study in the application of macromodels for model management in an industrial model-based vehicle control system development process. The focus of the project is on the identification, formalization, and use of model relationships to help detect model inconsistencies and facilitate model evolution. As the initial step of this exercise, we limit our scope to a subset of models used in software development.

The rest of the report is organized as follows. Section 2 presents background on the use of macromodels for model management, including approaches, formalization methods, and relations among different types of models. Here we set out the methodology that we apply to the case study that structures the remainder of the paper. Section 3 discusses the identification of the focus area for the case study. Section 4 describes the formalization of the model relationships used. Section 5 reports on the results of creating macromodels for the relationships of flow diagrams defined in functional architecture models. Section 6 describes the results of checking for relationship inconsistencies. Section 7 discusses related work and Section 8 reports on conclusions and next steps.

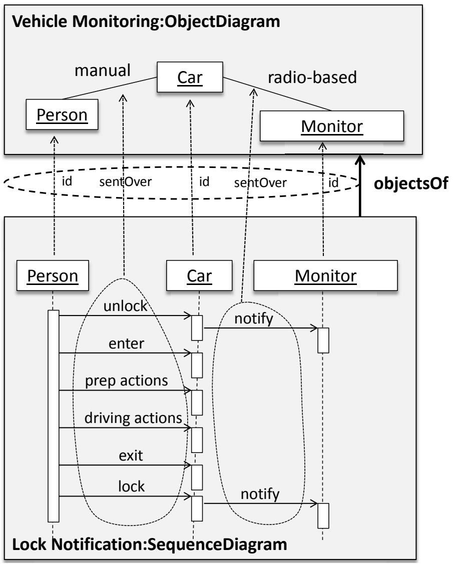


Fig. 1. A relationship between a sequence diagram and an object diagram

2 Background

A significant portion of model management activity deals with the management (creating, maintaining, organizing, repairing, evolving, etc.) of model relationships. To this end, the model relationship management is a centerpiece of many model management approaches.

Following a general definition that models say things about their "subject of study" [10], two models are said to be "related" (semantically) when what one expresses imposes some constraints on what the other one is able to express. For example, a design model for a system should be related to its requirements model by a "satisfies" relationship (i.e. the design should satisfy the requirements). When these constraints are violated, the models are considered to be inconsistent. Conceptually, the relationship between two (or more) models can be expressed as a special kind of model that defines the mappings relating the model elements. These relationships can be classified into types and can be formally defined using metamodels. Each type of relationship captures certain constraints that need to hold between models. As an example, Figure 1 shows an *objectsOf* relationship used in a generic vehicle control system.

The relationship of type *objectOf* is defined between a UML sequence diagram and an object diagram to express how the modeling elements in the sequence diagram containing types `Lifelines` and `Messages` are mapped to the modeling elements in the object diagram containing types `Objects` and `Links`. As defined in the relationship, each object/lifeline instance in the sequence diagram is mapped to an object instance in the object diagram that represents the same object (via the identity relation `id`). Similarly, each message instance in the sequence diagram is mapped to the association link instance in the object diagram over which the message is sent (via the relation `sentOver`).

In addition, the mapping is constrained in a way that both `id` and `sentOver` are functions, and must be consistent in terms of the endpoint objects of a message being the same as the endpoint objects of the link that it is mapped to. The mapping shown in Figure 1 satisfies these constraints, so it is a well-formed *objectsOf* instance. As development proceeds, if either diagram `Lock Notification` or `Vehicle Monitoring` is changed, the constraints of the *objectOf* relationship must continue to hold – failing to do so yields an inconsistency that must be corrected by modifying the diagrams. Defining various relationship types such as *objectsOf* allows the instances of these types to formally model the relationships between models at different abstraction levels: at the detail level, a relationship type defines how the elements of the models are related and the constraints that must hold between them; at the aggregate level, the modeled relationship can be used to express how the models are related as a whole and to convey information about how a collection of models is structured.

The aggregate-level model expressed using model types and their relationships is called a *macromodel* [9]. With all relationships of interest being formally and explicitly modeled, we can thus use them to specify meaningful macromodels. Figure 2 shows an example of the relationships between some UML models and diagrams of the hypothetical vehicle control system. Here, the model `Vehicle Control Design` captures the vehicle control design, and a collection of diagrams called `Vehicle Monitoring Diagrams` models the design details of the monitoring functionality.

The diagrams in this collection belong to the base model `Vehicle Control Design`. The relationship is indicated by the link `theModel`. Within this collection are the object diagram `Vehicle Monitoring` from Figure 1 along with several sequence diagrams that hold the *objectsOf* relationship to this object diagram. Specifically, these sequence diagrams include `Lock Notification` from Figure 1, `Emergency Notification`, and `Engine Malfunction Notification`. The last one is further decomposed into the two sequence diagrams `Sensor Misread` and `Power Drop`.

Two other relationships are defined in this macromodel fragment: `ODRelatedTo` and `deployedOn`. `ODRelatedTo` is a function that takes a model as its input, extracts the set of objects and links related to this argument from the base model, and constructs an object diagram. In this case, the macromodel specifies that the object diagram `Vehicle Monitoring` is a result of applying `ODRelatedTo(Monitor)` to `Vehicle Control Design`. The relationship `deployedOn` defines how a UML design is deployed on an architecture (also in UML) and is used here to relate `Vehicle Control Design` and `Vehicle Architecture`.

The macromodel represents the intended interrelations among the models and diagrams. As the development of the vehicle control system proceeds and as the design artifacts evolve, we expect the relationships expressed in the macromodel to be maintained. Thus, the macromodel provides a new type of specification in which the intentions of the modelers are expressed as constraints at the

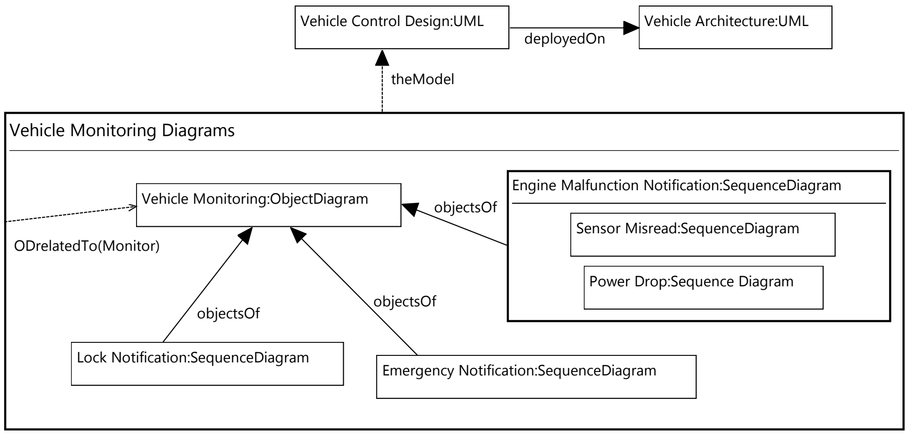

Vehicle Control Design:UML
deployedOn
Vehicle Architecture:UML
theModel
Vehicle Monitoring Diagrams
Vehicle Monitoring:ObjectDiagram
ODrelatedTo(Monitor)
objectsOf
Engine Malfunction Notification:SequenceDiagram
Sensor Misread:SequenceDiagram
Power Drop:Sequence Diagram
objectsOf
objectsOf
Lock Notification:SequenceDiagram
Emergency Notification:SequenceDiagram

Fig. 2. A partial macromodel of a vehicle control system specification

macroscopic level regarding what models must exist and what relationships must hold between these models [8]. Since these constraints are formalized using the metamodels of the relationship types and can be encoded to support machine analysis and processing, automation of model management activities, such as consistency checking and change propagation, becomes possible.

2.1 Macromodeling Methodology

A typical application of the macromodel based model management technique described above in general software development involves steps of the following methodology:

1. Identify and define the relationship types that are required for relating model types using metamodels.
2. Create an initial macromodel to specify the required models and their intended relationships.
3. Apply the macromodel to support the comprehension of the models and model management activities such as checking conformance of models with the intended relationships.
4. Evolve both the macromodel and the constituent models and relationships as the development process advances.

In remainder of this paper, we report on a case study applying steps (1)-(3) of this methodology to the vehicle control software development process. Step (4) is left as future work.

3 Case Study Preparation

In this section we describe the process used to identify the focus area for the case study.

The software development process is a part of the integral vehicle control system development process. In this process, the overall high-level control system requirements are specified according to the business needs, and are used for both downstream design and later system verification. Following the development process, the main dimension of decomposition is along the system functional architecture, which contains the Domains and their constituent Subsytems. For functional development, the output of each phase is specified as the requirements models, design models, software implementation, and test for the function. Design models are further divided into algorithm design and software design models.

The set of different kinds of models yield more than 70 relationship types. In this case study, concentrating on an early phase of a project for long-term adoption of model management, we selected a subset of these relationships to formalize them. The following criteria were used in the selection:

– Feasibility of formalization: The relationship must be sufficiently precise and the knowledge about the relationship must be available.
– Impact: Different relationships deliver different value to the development, and thus require us to consider the trade-offs when making a selection.
– Relevance to consistency checking: The focus of the model management in this research is consistency checking and repair. As such, the selected relationships should be relevant to this objective.
– Theoretical significance: As the first application of the macromodel concept to model management in vehicle control software development, theoretical exploration and study is critical to ensure the macromodel provides a proper foundation to address the model management needs in the automotive domain.
– Representativeness: The selected relationships should represent a broad set of model management scenarios where the relationships are maintained.

Based on these considerations, it was concluded that relationships between *flow diagrams* are a good candidate for this investigation with sufficient data (models and metamodels) to apply model management with macromodels. Flow diagrams (*FD*) are used to represent high-level system design using a set of system elements, the communication flows between these elements, and the flow items passed along the communication flows. Figure 3 shows an example flow diagram. The input to the case study was a fragment of an industrial vehicle product line UML model covering 3 domains and 7 subsystems. Two types of flow diagrams are used in this fragment:

– Functional Architecture Diagram (*FAD*). An *FAD* is a context diagram used at the architecture stage to identify interfaces and show the communications between domains and subsystems. The elements are `Domain` and `Subsystem`, both specialize `Package`. Only one-level containment exists: `Domain`*s* contain `Subsystem`*s*. The flow diagram in Figure 3 is a FAD.
– Component Diagram (*CpD*). A *CpD* is used at the design stage to show the flow of signals between components and interfaces. The elements are

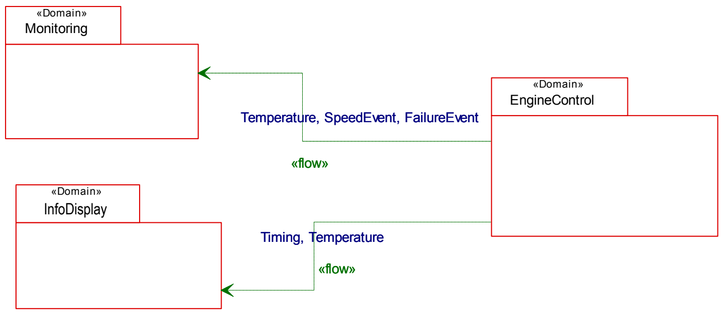


Fig. 3. An example flow diagram

`Component`*s* and `Interface`*s*. The classifiers conveyed across a flow are `Signal`*s*. An information flow can only occur between two components or between a component and an interface. No element containment exists in a *CpD*.

Multiple types of relationships are defined among the flow diagrams and the specific relationships of interest are defined in the next section.

4 Formalization of Flow Diagram Relationships

In this section we describe the application of step (1) of the methodology in Section 2.1 and describe the relationship types used with flow diagrams and their formalization.

The top of Figure 4 shows the flow diagram metamodel (FD), which is a segment of a domain-specific profile derived from UML 2 specification [7]. In this metamodel, modeling element `NamedElement` is an abstract class representing any identifiable UML elements. A `NamedElement` may contain other modeling elements via the `ownedElement` relationship. The `InformationFlow` type is defined to connect a source and a target `NamedElement` and can convey concrete elements, such as classes, objects, signals, etc., as well as abstract elements of type `InformationItem` representing other elements via the represented relationship.

Our observations of the flow diagrams used in this case study led us to identify several flow diagram relationship types. We first describe these as generic flow diagram relationships and then specialize them to versions that apply to FAD and CpD flow diagrams.

FD Submodel Relationship. $SubFD(D'$:FD, D:FD). Submodel relationships exist among flow diagrams. One flow diagram D' can be a submodel of another flow diagram D when all modeling elements in D' are also in D.

FD Refinement Relationship. $RefineFD(Con$:FD, Abs:FD). Flow diagram refinement deals with relative level of detail in different flow diagrams: a flow diagram with more modeling details, called the concrete one and denoted by Con,

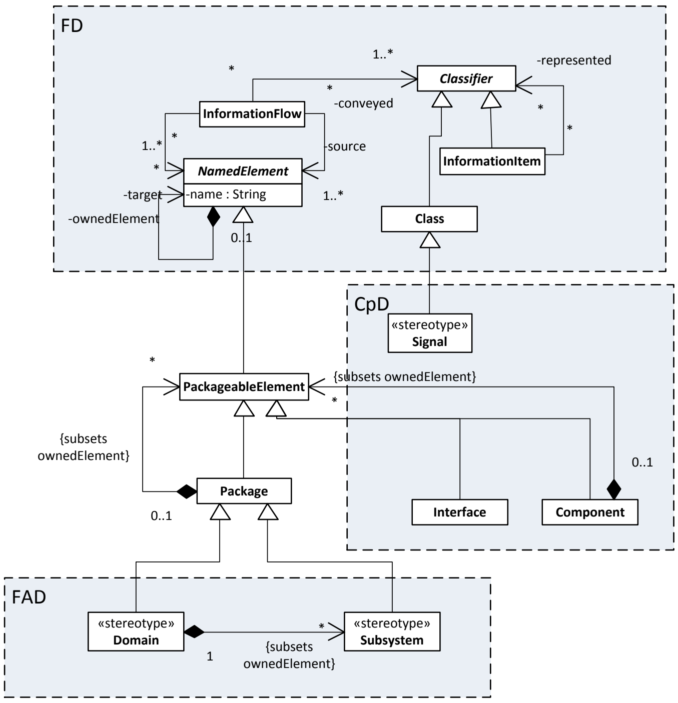


Fig. 4. Metamodel for flow diagram: FD and specializations FAD and CpD that extend it.

refines a flow diagram with fewer details, called the abstract one and denoted by *Abs*. To formally define the refinement relationship, we first define the concept of the relative level of detail between elements, between information flows, and between the conveyed classifiers. Specifically, the level of detail is defined as ordering relations:

– Element detail: $e_1 \preceq e_2 \iff TC(ownedElement(e_1, e_2))$. Element e_1 is more detailed than e_2 if they are connected by a sequence of zero or more `ownedElement` relationships. Here, $TC(ownedElement(.,.))$ is the "transitive closure" of *ownedElement*.
– Flow detail: $f_1 \preceq f_2 \iff source(f_1) \preceq source(f_2) \wedge target(f_1) \preceq target(f_2)$. Information flow f_1 is more detailed than flow f_2 if its endpoints are more detailed.
– Classifier detail: $c_1 \preceq c_2 \iff (c_1 = c_2) \vee (InformationItem(c_2) \wedge TC(represented(c_2, c_1)))$. For classifiers conveyed on a flow, only information items can be used to increase the level of detail via the `represented` relationship.

The detail ordering definition is used to determine when information is at the same level of detail, or are *siblings*:

$$sibling(x, y) \iff \exists a \cdot x \preceq a \wedge y \preceq a \wedge (\neg\exists a' \cdot x \preceq a' \wedge y \preceq a' \wedge a' \prec a)$$

According to this definition, x and y are siblings iff they have a common ancestor and there is no other common ancestor in between — i.e., they must have a common parent. We now state the three conditions for *Con* to be a flow diagram refinement of *Abs*:

1. *Con* should only contain information that refines some information in *Abs*:
$$\forall x \in Con \exists y \in Abs \cdot x \preceq y$$
2. *Con* refines *Abs* completely at a given level of detail:
$$\forall x \in Con, x' \in Base \cdot sibling(x, x') \Rightarrow x' \in Con$$
3. *Con* refines all of *Abs*:
$$\forall y \in Abs \exists x \in Con \cdot x \preceq y$$

Note that these conditions are defined generically so they can be applied to elements, flows or conveyed classifiers. In condition (2), *Base* refers to the UML model that *Con* and *Abs* are diagrams of.

FD Submodel Refinement Relationship. $SubRefineFD(Con{:}\mathrm{FD}, Abs{:}\mathrm{FD})$. We will say that *Con* is a submodel refinement of *Abs* when *Con* is a submodel of a refinement of *Abs*. Thus, *SubRefineFD* is a composition of *SubFD* and *RefineRD*. Formally,

$$SubRefineFD(Con, Abs) \iff \exists m : FD \cdot SubFD(Con, m) \wedge RefineFD(m, Abs)$$

FD Extractor Transformations. Extractor transformations generate diagrams from the base model. Two extractor transformations related directly to FDs are:

– $FDfor(E : \texttt{NamedElement}) : FD$ which generates the FD that shows the element E and all its neighbouring elements connected to E by information flows at the same level of detail.
– $FDin(E : \texttt{NamedElement}) : FD$ which generates the FD that shows the elements within E (i.e. related to E by the `ownedElement` relationship) and the information flows between these elements.

Specialization of Flow Diagram Relationships. The lower part of Figure 4 shows the metamodels of the specialized flow diagrams used in the vehicle control software development process. For *FAD*'s, the elements are `Domain` and `Subsystem`, and both specialize `Package`. Only one-level containment exists: `Domain`*s* contain `Subsystem`*s*. For *CpD*'s the elements are `Component`*s* and `Inter-face`*s*. The conveyed classifier is `Signal`*s*. An information flow can only occur between two components or between a component and an interface. No element containment exists in a *CpD*.

FD relationships are specialized correspondingly as follows. The submodel relationship *SubFD* is specialized to *SubFAD* and *SubCpD* by restricting the

Table 1. Summary of flow diagrams in the ACCFCA subsystem

Name	Type	Description
SITM	FAD	Driving Management domain context
ACCFCA	FAD	Adaptive Control subsystem context
ACCB	CpD	Adaptive Control Algorithm component
ACCC	CpD	Adaptive Control Communication component
ACCP	CpD	Adaptive Control Interface component
ACCMV	CpD	Adaptive Control component for motion
GS	CpD	Hardware Interface component

models to *FAD* and *CpD*, respectively. The *RefineFD* is specialized to *RefineFAD* for *FAD* and *RefineCpD2FAD* to express the refinement from an *FAD* to a *CpD*. Since *SubRefineFD* is a composition of *SubFD* and *RefineFD*, its specializations are derived from the specializations of its components. For example, *SubRefineFAD* is the composition of *SubFAD* and *RefineFAD*. The extractors *FDfor* and *FDin* are also specialized to *FADfor*, *FADin*, *CpDfor* and *CpDin* in the natural way.

5 Macromodel Definition

In this section we describe the application of step (2) of the methodology in Section 2.1 and describe the development of the macromodels used for the case study model fragment.

In all, we defined 7 macromodels – one per subsystem in the case study model fragment. This level of decomposition seemed appropriate since the model content is decomposed at the subsystem level and subsystems are taken to represent independent units of functionality that interact with other subsystems through well defined interfaces. In total, the 7 macromodels referenced 14 FAD's and 12 CpD's. Of these 3 FAD's were referenced from multiple macromodels since they represented domain level information that was common to several subsystems.

In order to more clearly describe the result of producing a macromodel for a subsystem we illustrate with an example advanced adaptive control function, the ACCFCA subsystem, in a driving management domain $SITM$. Table 1 summarizes the diagrams used in ACCFCA subsystem.

Figure 5 shows the macromodel for the relationships of these diagrams. The dashed directed link labeled "theModel" specifies that FlowDiagramforACCFCA represents a set of submodels of base model ProductLine : UML. The other dashed directed links show extractor transformations, while the solid arrows show other relationships. The extractor transformations start at the boundary of the box(es), indicating the extraction of a submodel from the base model. The *CpD* diagrams ∗ACCFCA_Components and ∗M2 are not *realized* (indicated by the "*" prefix), so they are not actually created but are implicitly present because they are necessary to understand the relationships between the ACCFCA diagrams. Specifically, ∗ACCFCA_Components represents the complete component diagram for the subsystem ACCFCA (i.e., *CpDin*(ACCFCA)) and this is decomposed into three *CpD*

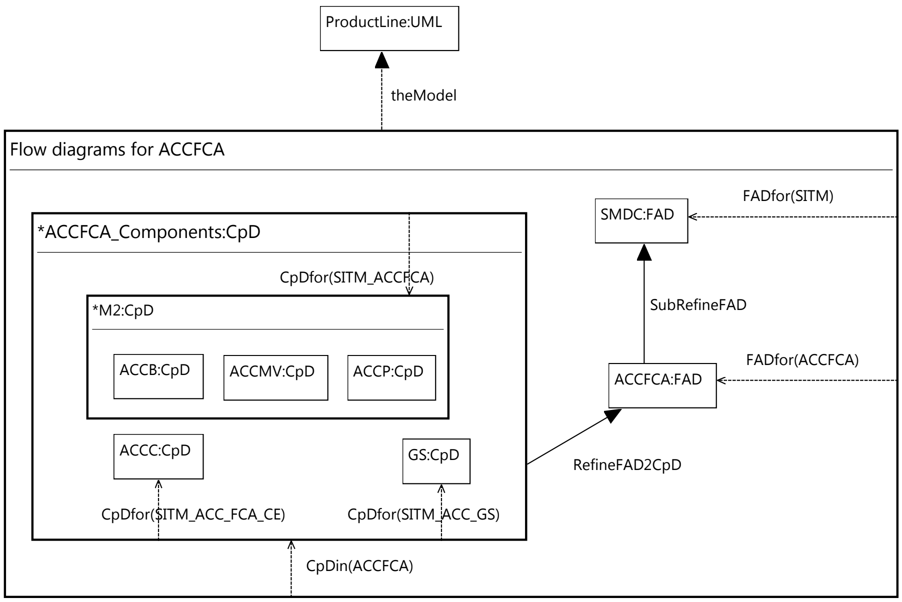


Fig. 5. The macromodel for `ACCFCA` flow diagram relationships

Table 2. The identified constraints among the flow diagrams of `ACCFCA`

Id	**Constraint**
1	$*$ACCFCA_Components $= \cup\{*$M2 , ACCC, GS$\}$
2	$*$ACCFCA_Components $= CpDin($ProductLine, ACCFCA$)$
3	$*$M2 $= \cup\{$ACCB, ACCMV, ACCP$\}$
4	$*$M2 $= CpDfor($ProductLine, SITM_ACCFCA$)$
5	ACCC $= CpDfor($ProductLine, SITM_ACCFCA_CE$)$
6	GS $= CpDfor($ProductLine, SITM_ACCFCA_GS$)$
7	$RefineFAD2CpD($*ACCFCA_Components, ACCFCA : FAD$)$
8	ACCFCA : FAD $= FADfor($ProductLine, ACCFCA$)$
9	$SubRefineFAD($ACCFCA : FAD, SITM : FAD$)$
10	SITM : FAD $= FADfor($ProductLine, SITM$)$

diagrams – one for each component in `ACCFCA`. However, the *CpD* for component `SITM_ACCFCA` is not realized (represented here as $*$`M2`) and is decomposed into three *CpD*s. The macromodel in Figure 5 graphically asserts ten constraints that must hold between the flow diagrams in `ACCFCA`. These are summarized in textual form in Table 2.

Generalized Macromodel. Macromodels are not only for visualizing the relationships between specific models but also for expressing general patterns of relationships that could be used for methodological constraints. This property enables the macromodel being used to define modeling standards and a way to enforce that the development teams follow them when creating models.

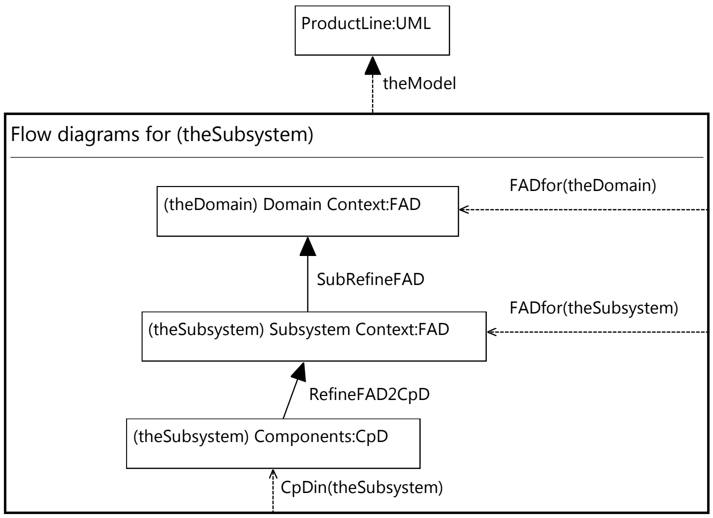

ProductLine:UML
theModel
Flow diagrams for (theSubsystem)
FADfor(theDomain)
(theDomain) Domain Context:FAD
SubRefineFAD
FADfor(theSubsystem)
(theSubsystem) Subsystem Context:FAD
RefineFAD2CpD
(theSubsystem) Components:CpD
CpDin(theSubsystem)

Fig. 6. A generalized macromodel for subsystem diagrams

Furthermore, a tool may be implemented to automatically check the conformance to these standards.

Figure 6 shows a hypothetical macromodel defining the standard way to express the collection of diagrams associated with any subsystem. Here, the diagrams have parameters (i.e., `theSubsytem` and `theDomain`) and can be instantiated to produce a macromodel for a particular subsystem by assigning values to these parameters. Since the macromodel technique is not currently being used in development, we inferred this generalized macromodel by comparing the macromodels for the different subsystems in the case study model fragment. An interesting observation is that the macromodel in Figure 6 could be considered as the undocumented standard regarding subsystem diagrams, and the macromodel technique can be used to make such standards explicit.

6 Conformance Checking

In this section we describe the application of step (3) of the methodology in Section 2.1 and apply the relationship definitions and macromodels from Sections 4 and 5, respectively, to do consistency checking. Specifically, we describe the results of checking conformance of the actual relationships in the case study model fragment to the intended relationships and generalized macromodel.

The conformance checking was done by implementing the relationship types in Section 4 using a Visual Basic (VB) script. VB was chosen as the implementation platform because it natively integrated with Rhapsody and provides programmatic access to the model contents.

Table 3 shows the results of checking conformance of the `ACCFCA` example with the constraints listed in Table 2. Constraint 2 is violated because a component exists in the model that does not appear in any diagram. That is, constraints 1 and 2 together imply that $CpDin$(ProductLine, ACCFCA) = $\cup\{$*M2 , ACCC, GS$\}$ and since by its definition, $CpDin$(ProductLine, ACCFCA) contains all components in subsystem ACCFCA and $\cup\{$*M2 , ACCC, GS$\}$ contains all the components in the component diagrams for ACCFCA, these two sets of components should be the same but they are not. A similar violation occurs for constraint 4 (in conjunction with constraint 3) but here an additional flow is found that does not appear in any component diagram. On closer examination, there is also a similarly named flow SITM_ACCFCA_rsp_0 that differs only with an additional suffix "_0" in the diagram. This may suggest that there are cases where an element is duplicated during modeling and then forgotten. The conformance checks help identify the discrepancies that when repaired, will improve the quality of the model for these cases.

Finally, we notice that constraint 7 cannot be checked because of insufficient information in the model. In particular, in the diagram ACCFCA, the "represented" relationship was not used to link the information items conveyed by the flows of the *FAD* to the signals conveyed by the flows of the *CpD*s. One possible reason why this problem has not been previously identified may be that while *CpD*s are used in code generation, the content of *FAD*s are not - thus, there is less incentive to maintain the links between these diagrams. Nevertheless, since *FAD*s are used as part of design reviews, the lack of linkage between these levels may result in incorrect conclusions in these reviews. Thus, this gap may indicate a more serious systemic problem in the modeling process.

Table 4 summarizes the results of conformance checking over all 7 subsystems in the case study fragment. The relationships considered are taken from the generalized macromodel in Figure 6 since these are found in the macromodels of all subsystems. Of the 35 instances of intended relationships, 13 (37%) were found to be conformant, 14 (40%) were found to be non-conformant and 8 (23%) could not be determined for various reasons. These results suggest that the use of macromodels can provide value in uncovering model defects and improving model quality.

Conformance to Generalized Macromodel. In addition to conformance to the relationships within a macromodel, we can check the conformance of a particular macromodel to the general macromodel. As an example, if the macromodel for ACCFCA in Figure 5 is assumed to be an instance of the generalized macromodel in Figure 6, we can check its degree of conformance to this standard. In this case, the ACCFCA flow diagram model is conformant except the context instances of *FAD* type have different names. Note that the decomposition of *ACCFCA_Components is not a case of non-conformance because the generalized macromodel does not prohibit the further decomposition of a model.

Assuming that a generalized macromodel is used as a standard in a development process, the above naming bug may be caught by using an automated conformance check. The generalized macromodel could alternatively be used as a template to guide modeling in a top-down fashion so that the the collection of diagrams is guaranteed to be "correct-by-construction".

Table 3. The results of `ACCFCA` diagrams conformance checks

Id	Conformance to Constraint
1	Conformant (by definition, since *`ACCFCA_Components` is a derived diagram)
2	!Non-Conformant - *CpDin*(`ProductLine`, `ACCFCA`) contains the component "`SITM_READ_GS`" but this is not found in *`ACCFCA_Components` (i.e., it is not in any component diagram)
3	Conformant (by definition, since *`M2` is a derived diagram)
4	!Non-Conformant - *CpDfor*(`ProductLine`, `SITM_ACCFCA`) contains the flow named "`SITM_ACCFCA_rsp_0`" with source `SITM_ACCFCA` that does not appear in *`M2` (i.e., it is not in a component a diagram for `SITM_ACCFCA`)
5	Conformant
6	Conformant
7	Unable to verify conformance to insufficient information
8	Conformant
9	Conformant
10	Conformant

Table 4. Aggregate results of conformance checking for each of the 7 subsystem in the case study model fragment for the 5 relationships from the generalized macromodel. Columns are: number conformant (# Conf), number non-conformant (# NConf) and number that could not be determined (# Unk). Here s = `theSubsystem` and d = `theDomain`.

Constraint	# Conf	# NConf	# Unk
RefineFAD2CpD((*s*)`Components`, (*s*)`SubsystemContext`)	1	4	2
(*s*)`SubsystemContext` = *FADfor*(`ProductLine`, *s*)	4	3	0
SubRefineFAD((*s*)`SubsystemContext`, (*d*)`DomainContext`)	7	0	0
(*d*)`DomainContext` = *FADfor*(`ProductLine`, *d*)	1	6	0
(*s*)`Components` = *CpDin*(`ProductLine`, *s*)	0	1	6

In the 7 macromodels, 3 cases of non-conformance were detected and in each case these were naming inconsistencies.

7 Related Work

Existing work on dealing with multiple interrelated models has been done in a number of different areas. The ViewPoints framework [6] was an influential early approach to multiview modeling. The macromodeling approach differs from this work in being more formal and declarative rather than procedural. Furthermore it treat relationships as first class entities and provide support for typing of relationships.

More recently, configurable modeling environments have emerged such as the Generic Modeling Environment (GME) [5]. None of these approaches provide general support for expressing model relationships or their types; hence, they have limited support for defining and expressing interrelated collections of models. Furthermore, the focus of these approaches is on the detail level (i.e. the content of particular models) rather than at the aggregate level as with macromodels.

Process modeling approaches like the Software Process Engineering Metamodel (SPEM) [4] are complementary to the notion of a macromodel. With process modeling, the main focus is to define how activities produce or consume models and which process actors perform these activities. A macromodel adds to this the ability to define how the content of models must be related and thus adds correctness conditions to the process definition.

The term "megamodel" as representing models and their relationships at the aggregate level emerged first in the work of Favre [3] and also later as part of the Atlas Model Management Architecture (AMMA) [2]. Macromodels bear similarity to these two kinds of megamodels, but the intent and use is quite different - to express the modeler's intentions in a development process.

Finally, the work on model traceability also deals with defining relationships between models and their elements (e.g., [1]); however, this work does not have a clear approach to defining the semantics of these relationships. Thus, the macromodeling framework can provide a way to advance the work in this area.

8 Conclusion and Future Work

In this report, we present a model management approach using formal relationships and its application to vehicle control software development. With this technique, the types of relationships used to relate models are first identified, then formalized using metamodels, and finally used in macromodels to specify the interrelationships between the models of interest in development. The model relationships specified in such a way provides a basis for automatic consistency checking when the models are altered and evolving, which consequently improves the model and product quality.

As the first phase, this research has analyzed different relationship types used in the vehicle control software development in order to determine representative relationship types. Based on the analysis results, the refinement relationships between the flow diagrams of the functional architecture context diagrams and the component diagrams have been selected for the exercise. The refinement and submodel relationship types, along with their constraints, have been modeled using macromodels and applied to 7 subsystems across 3 vehicle domains.

The results of applying our model management technique to the vehicle software development process has shown the value of the technique in several ways. First, some inconsistencies within the example models have been detected. Addressing them will improve quality of the models. Second, the relationship formalization process has uncovered some gaps in the information within the UML model that could indicate a systemic problem in the modeling process. Finally, we have revealed an undocumented convention that has been (mostly) followed in multiple subsystems. This convention has then been formalized as a generalized macromodel and has been presented as a standard for modelers to follow when expressing the *FAD*s and *CpD*s.

The future work of this research will build on the currently-obtained results in two ways. First, we will extend testing of the formalization in order to assess the robustness. Second, we will develop and test automatic repairs of identified inconsistencies. We hope that this research will ultimately lead to new tools that support formal and automatic model management.

References

1. Aizenbud-Reshef, N., Nolan, B.T., Rubin, J., Shaham-Gafni, Y.: Model traceability. IBM Systems Journal 45(3), 515–526 (2006)
2. Barbero, M., Jouault, F., Bézivin, J.: Model driven management of complex systems: Implementing the macroscope's vision. In: Proceedings of 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, pp. 277–286. IEEE (2008)
3. Favre, J.M.: Megamodelling and etymology. Transformation Techniques in Software Engineering 5161 (2005)
4. Object Management Group. Software & Systems Process Engineering Meta-Model Specification Version 2.0 (2008), `http://www.omg.org/spec/SPEM/2.0/`
5. Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., Volgyesi, P.: The generic modeling environment. In: Workshop on Intelligent Signal Processing, Budapest, Hungary, vol. 17 (2001)
6. Nuseibeh, B., Kramer, J., Finkelstein, A.: A framework for expressing the relationships between multiple views in requirements specification. IEEE Transactions on Software Engineering 20(10), 760–773 (1994)
7. Object Management Group. Unified modeling language superstructure, version 2.4.1 (2011), `http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/`
8. Salay, R., Mylopoulos, J.: The Model Role Level – A Vision. In: Parsons, J., Saeki, M., Shoval, P., Woo, C., Wand, Y. (eds.) ER 2010. LNCS, vol. 6412, pp. 76–89. Springer, Heidelberg (2010)
9. Salay, R., Mylopoulos, J., Easterbrook, S.: Using Macromodels to Manage Collections of Related Models. In: van Eck, P., Gordijn, J., Wieringa, R. (eds.) CAiSE 2009. LNCS, vol. 5565, pp. 141–155. Springer, Heidelberg (2009)
10. Seidewitz, E.: What models mean. IEEE Software 20(5), 26–32 (2003)

Detecting Specification Errors in Declarative Languages with Constraints

Ethan K. Jackson, Wolfram Schulte, and Nikolaj Bjørner

Microsoft Research, Redmond, WA
{ejackson,schulte,nbjorner}@microsoft.com

Abstract. Declarative specification languages with constraints are used in model-driven engineering to specify formal semantics, define model transformations, and describe domain constraints. While these languages support concise specifications, they are nevertheless prone to difficult semantic errors. In this paper we present a type-theoretic approach to the static detection of specification errors. Our approach infers approximations of satisfying assignments and represents them via a *canonical regular type system*. Type inference is experimentally efficient and type judgments are comprehensible by the user.

1 Introduction

Declarative specification languages with constraints are used in model-driven engineering to specify formal semantics [1,2,3], define model transformations [4,5,6], and describe domain constraints. While these languages support concise specifications, they are nevertheless prone to difficult semantic errors: A constraint may be unsatisfiable; it may be satisfiable for unintended values; or it may improperly trigger a rewrite rule, transformation step, or constraint violation. Standard type-theoretic approaches to early error-detection do not detect these kinds of errors, because they do not infer which values satisfy constraints.

In this paper we present a type system and type inference algorithm for assigning *semantic types* to variables occurring within constraints. Inferred types denote sets of values over-approximating the satisfying assignments. For instance, if the inferred type of a variable is the empty set, then the constraint is certainly unsatisfiable. However, if the inferred type is non-empty, then our approach produces a canonical description of the possible satisfying assignments. These canonical types are in practice small. Therefore, inferred types can be examined by the user and carry important information even in the absence of errors. To our knowledge, this is the first semantic type system to take this approach. It has been implemented in the FORMULA language and we present experimental results.

The paper is structured as follows: Section 2 motivates the use of semantic types by a small example. We summarize basic notation in Section 3. Section 4 introduces canonical regular gypes, which form the foundations of our approach. Section 5 generalizes regular types to incorporate constraints over interpreted

```
Predicate       Pred   ::= true | false | Rexp | ( Pred ) | B1 Pred | Pred B2 Pred
Rel. Expr.      Rexp   ::= ( Rexp ) | Aexp R2 Aexp
Arith. Expr.    Aexp   ::= Aatom | ( Aexp ) | A1 Aexp | Aexp A2 Aexp
Arith. Atom     Aatom  ::= var | int
Bin. Rel. Op.   R2     ::= == | != | < | > | <= | >=
Un. Arith. Op.  A1     ::= + | -
Bin. Arith. Op. A2     ::= + | - | * | / | %
Un. Bool. Op.   B1     ::= !
Bin. Bool. Op.  B2     ::= && | ||
```

Fig. 1. A BNF grammar for several kinds of expressions

functions. Section 6 presents type inference via a saturation algorithm, and we provide experimental results in Section 7. We end with related work and conclusions in sections 8 and 9.

2 A Motivating Example

We begin with a motivating example showing how errors can arise in model transformations, formal specifications, and domain constraints. They are generic, so we attempt to describe them in a language neutral manner. Figure 1 shows a BNF grammar for expressions found in C-like languages: Integer arithmetic expressions (Aexp) can combined with relational operators (R1, R2) into relational expressions (Rexp); these can be further combined with Boolean operators (B1, B2) to form predicates (Pred). Suppose a modeling language for state machines allows transitions to be guarded by constraints written in the *Object Constraint Language* (OCL). OCL has slightly different expressions permitting the special constant null to appear as arithmetic and Boolean atoms:

$$\mathsf{Aatom}_{\mathsf{OCL}} ::= \textbf{null} \mid \textbf{int} \qquad \mathsf{Pred}_{\mathsf{OCL}} ::= \textbf{null} \mid \textbf{true} \mid \ldots$$

One typical task of a *model transformer* or *code generator* is to translate from the abstract syntax of one *domain specific language* (DSL) to the abstract syntax of another DSL. For instance, a model transformation can transform a state machine into a C program. In doing so, it will not operate on the string representations of transition guards, but instead on abstract syntax trees (ASTs) or abstract syntax graphs (ASGs) produced after parsing the input model. Practically all model transformation languages operate on structured data, and our type system shall take advantage of this fact. Here is an example of a model transformation rule (in pseudocode) where patterns in the rule range over ASTs / ASGs:

```
FIND:    A transition $t$ s.t. $src(t) = s_a$, $dst(t) = s_b$, and $guard(t) = g_{OCL}$.
CREATE:  If(current == $id(s_a)$ && $g_{OCL}$) { current = $id(s_b)$; }.
```

This rule transforms guarded transitions into snippets of C ASTs. It also contains an error, because not every OCL predicate is a well-formed C predicate. For example, the malformed C AST $\mathsf{if}(1 + \mathsf{null} == \mathsf{null})\{\ \ldots\ \}$ can be obtained from a legal input model. This error can be statically detected by inferring types for the *find* and *create* portions of the rule. In the *find* portion a type τ is inferred for g_{OCL} denoting all ASTs that might trigger the rule if substituted for g_{OCL}. In the *create* portion a type τ' is inferred for g_{OCL} denoting all well-formed C predicate ASTs. If τ is not a *subtype* of τ', then there is some value that could trigger the rule and create a malformed AST. Here, subtype means subset inclusion of denoted values. In our experience, it is very easy to create these types of errors when transforming between complex abstract syntaxes.

The previous example can be phrased using constraints over *algebraic data types* (ADTs). The problem of inferring types is equivalent to approximating the satisfying assignments of these constraints. However, specifications often contain other kinds of constraints than just constraints over ADTs. Consider the problem of specifying the semantics of OCL expressions. Then such a formal specification might contain (in pseudocode) an axiom:

```
FORALL:  AST $e$ in $\mathsf{Aexp}_{\mathsf{OCL}}$, IF: $e = Add(v_1, v_2)$ and $k = v_1 + v_2$,
THEN:    $eval(e) = k$.
```

Here $\mathsf{Aexp}_{\mathsf{OCL}}$ denotes the set of OCL integer arithmetic ASTs. The first constraint is an ADT constraint requiring e to be a syntax tree constructed from values v_1 and v_2 by applying the *Add* operator. This constraint uses the *data constructor Add* instead of the string '+' to indicate the structure of the expression. The second constraint involves the operator + defined by the *theory of arithmetic*. This operator takes two reals and computes their arithmetic sum. When these constraints are satisfied it may be concluded that $eval(e) = k$. A full specification of OCL would contain many such statements to inductively define the evaluation of arithmetic expressions with the help of the theory of arithmetic.

In the above axiom there is a subtle interaction between the two constraints that can be a source of error. The *Add* operator is a function $Add : \mathsf{Aexp}_{\mathsf{OCL}} \times \mathsf{Aexp}_{\mathsf{OCL}} \rightarrow \mathsf{Aexp}_{\mathsf{OCL}}$. The + operator is a function $+ : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$. The values for the variables v_i must be in the domain of both operators, which is the set of integers: $\mathsf{Aexp}_{\mathsf{OCL}} \cap \mathbb{R} = \mathbb{Z}$. Therefore, while this axiom is satisfiable for some values, it never defines the evaluation of OCL addition for null values. Though this specification would not generate a type error, type inference would assign the type Integer to variables. The user could inspect the inferred type to quickly learn that null will never be handled by this axiom. The appropriate coarse of action would be to add a new axiom defining OCL addition on null values.

```
FORALL:  AST $e$ in $\mathsf{Aexp}_{\mathsf{OCL}}$, IF: $e = Add(\mathsf{null}, v)$ or $e = Add(v, \mathsf{null})$,
THEN:    $eval(e) = \mathsf{null}$.
```

Declarative specifications are also used to write domain constraints, which are constraints on models that should never be violated. For instance, suppose state machines must be deterministic, then there must never be two transitions starting from the same state leading to different starts guarded by overlapping intervals:

NEVER: Transitions t and t' s.t. $guard(t) \neq guard(t')$, $src(t) = src(t')$, $guard(t) = LessEq(e, k)$, and $guard(t') = LessEq(e, k)$

There is a mistake in this constraint. The final constraint should have been written $guard(t') = LessEq(e, \underline{k'})$. Without this correction the constraint is unsatisfiable because it implies that $guard(t) = guard(t')$. In fact, our type system will catch this error, because during type inference new equalities can be learned. The equality $guard(t) = guard(t')$ will be learned, which is inconsistent with the constraint $guard(t) \neq guard(t')$.

In summary, declarative specifications with constraints are used for many purposes in MDE. They are prone to bugs that are both difficult to find and fix and are outside the scope of existing type inference schemes. We now proceed to formalize the type system and develop type inference.

3 Notation

Terms. A *term* t is either a constant c, a variable x, or an n-ary *function symbol* f applied to n terms $f(t_1, \ldots, t_n)$. We also write $\boldsymbol{t}$ for a vector of terms $(t_1, \ldots, t_n)$ and $f(\boldsymbol{t})$ for an f-application. The i^{th} component of $\boldsymbol{t}$ is $\boldsymbol{t}(i)$. We write $t' \sqsubseteq t$ if t' is a subterm of t, and $t[\boldsymbol{x}]$ for a term with variables $\boldsymbol{x}$ as subterms. The term $t[t' \backslash t'']$ is formed by simultaneous replacement of every subterm t' with t''.

Signatures. A *signature* Σ is a set of constants, variables, and function symbols. A Σ-term is a term built from the symbols of Σ.

Models. A *model* $\mathcal{M}$ of Σ-terms over universe U is a family of functions, one for each symbol of Σ. The interpretation of a variable x is a constant $x^{\mathcal{M}} : U$. The interpretation of an n-ary symbol f is a function $f^{\mathcal{M}} : U^n \rightarrow U$. An interpretation $\mathcal{M}$ *models* a first-order formula φ, written $\mathcal{M} \models \varphi$, if $\varphi^{\mathcal{M}}$ is a tautology. We write $\varphi \models \psi$ if every model of φ is also a model of ψ.

4 Canonical Regular Types

All the examples in Section 2 relied on tree-structured data that we loosely called ASTs. In this section we review a formalization of tree-structured data using algebraic data types (ADTs). We employ a unique variant of ADTs, called *canonical regular types*, were type expressions have canonical forms. Later, canonicity shall be used to present the results of type inference in a concise and comprehensible form. ADTs begin with data constructors.

Definition 1 (Data Constructors)**.** Let U be a universe, then a set of data constructors is described by a signature:

$$\Sigma_{con} \overset{def}{=} \langle C, f_1, \ldots, f_k \rangle,$$

where $C \subseteq U$ is a (possibly infinite) set of constants, and each $f_i : U^n \to U$ is an n-ary data constructor. The only equalities satisfied by data constructors are the trivial equalities: $\forall \boldsymbol{x}.\ f_i(\boldsymbol{x}) = f_i(\boldsymbol{x})$.

Applying data constructors to elements of U returns new tree-structured data. U provides a labeling for these trees, but its details can be ignored because every Σ_{con}-term identifies a unique element of the universe. For example:

$$Add(1, Add(2,3)) \neq Add(Add(1,2),3)),$$

for $\Sigma_{con} \overset{def}{=} \langle U, \mathbb{Z}, Add \rangle$. In summary, data constructors exploit the inherent tree-structure of terms to formalize ASTs.

ADTs extend data constructors with types denoting subsets of trees. The expressive power of types depends on the formalism. *Regular types* are highly expressive; they can precisely denote recursively enumerable sets of trees. In order to define types, users provide a set of type names and a set of type equations. The denotation of user types it the smallest solution to the system of equations.

Definition 2 (Regular Type System)**.** Given a set of data constructors, then a *regular type system* is a structure:

$$\Sigma_{reg} \overset{def}{=} \langle \Sigma_{con}, B, T, \bot, \cap, \cup, E, [\![ ]\!] \rangle,$$

where B is a set of *base types*, T is a set of *user types*, $\bot \in B$ is the *empty type*, and $\cap$ / $\cup$ are type intersection / union. E is a set of *type equations* and $[\![ ]\!]$ is the *type denotation function*; it is the least function satisfying Figure 2. In the text to follow let α range over user types and β range over base types.

Sets of trees are denoted by *type terms*. A type term τ is a term built from data constants / constructors, base types / user types, or the operators $\cap$ / $\cup$. Let $\tau(\Sigma_{reg})$ be the set of all type terms; an example is:

$$f(f(\mathsf{Natural})) \cup g(1, \mathsf{Integer})$$

where $\mathsf{Natural}$ and $\mathsf{Integer}$ are base types denoting the sets of natural numbers and integers. For example, $[\![\mathsf{Natural}]\!] \overset{def}{=} \{0, 1, 2, \ldots\}$. Applying a constructor f to type terms $\boldsymbol{\tau}$ yields a type term $f(\boldsymbol{\tau})$ denoting all the trees with root f and children conforming to $\boldsymbol{\tau}$. For example, $[\![f(\mathsf{Natural})]\!] \overset{def}{=} \{f(0), f(1), \ldots\}$. Then the operators $\cap$ / $\cup$ denote the mathematical intersections / unions of types. The subtyping relation is a semantic one; $\tau' <: \tau$ iff $[\![\tau']\!] \subseteq [\![\tau]\!]$:

$$1 <: 1 \cup 2 <: \mathsf{Natural} <: \mathsf{Integer}.\ f(1) <: f(1) \cup f(2) <: f(1 \cup 2) <: f(\mathsf{Natural}).$$

$$\boxed{[\![\ ]\!] : \tau(\Sigma_{reg}) \rightarrow 2^U}$$

$$[\![\bot]\!] \overset{def}{=} \emptyset. \qquad [\![\tau_1 \cup \tau_2]\!] \overset{def}{=} [\![\tau_1]\!] \cup [\![\tau_2]\!].$$
$$[\![c]\!] \overset{def}{=} \{c\}, \text{ for } c \in C \qquad [\![\tau_1 \cap \tau_2]\!] \overset{def}{=} [\![\tau_1]\!] \cap [\![\tau_2]\!].$$
$$[\![\beta]\!] \overset{def}{=} \{c_1, c_2, \ldots\}, \text{ for } \beta \in B - \{\bot\}. \qquad [\![f(\boldsymbol{\tau})]\!] \overset{def}{=} \{f(\boldsymbol{t}) \,|\, \boldsymbol{t} \in [\![\boldsymbol{\tau}]\!]\}.$$
$$[\![\boldsymbol{\tau}]\!] \overset{def}{=} [\![\boldsymbol{\tau}(1)]\!] \times \ldots \times [\![\boldsymbol{\tau}(n)]\!]. \qquad [\![\alpha]\!] = [\![\tau]\!], \text{ if } \alpha \approx \tau \in E.$$

Base types are closed under $\cap$ *and* $\tau' <: \tau \overset{def}{=} [\![\tau']\!] \subseteq [\![\tau]\!]$.

Fig. 2. Semantics of regular type systems

4.1 Type Equations and Uniformity

Type equations allow users to define the shapes of ASTs and to assign names to sets of ASTs. A type equation is a pair $\alpha \approx \tau$ for $\alpha \in T$. For example:

$$\alpha_{\mathsf{Var}} \approx \mathsf{Var}(\mathsf{String}). \quad \alpha_{\mathsf{Not}} \approx \mathsf{Not}(\alpha_{\mathsf{Pred}}). \quad \alpha_{\mathsf{And}} \approx \mathsf{And}(\alpha_{\mathsf{Pred}}, \alpha_{\mathsf{Pred}}).$$
$$\alpha_{\mathsf{State}} \approx \mathsf{State}(\mathsf{Integer}). \quad \alpha_{\mathsf{Trans}} \approx \mathsf{Trans}(\alpha_{\mathsf{State}}, \alpha_{\mathsf{Pred}}, \alpha_{\mathsf{State}}).$$
$$\alpha_{\mathsf{Pred}} \approx \mathsf{true} \cup \mathsf{false} \cup \ \alpha_{\mathsf{Var}} \cup \alpha_{\mathsf{Not}} \cup \alpha_{\mathsf{And}}.$$

The solution to this system of equations assigns to the type name α_{Pred} all the well-formed Boolean predicate ASTs comprised of true, false, variables, Not, and And operations. Similary, α_{State} denotes all integer-labeled State terms and α_{Trans} denotes all transitions with well-formed guard predicates. Notice that each data constructor is paired with a type equation. The intent of these equations is to express the valid uses of the data constructors.

Definition 3 (Uniform Regular Type Systems)**.** A type system Σ_{reg} with equations E is a *uniform regular type system* if:

1. For every $f \in \Sigma_{con}$ there is exactly one type equation $\alpha_f \approx f(\boldsymbol{\tau})$ in E.
2. If $\alpha_f \approx f(\boldsymbol{\tau}) \in E$, then every $\boldsymbol{\tau}(i)$ does not contain data constructors.
3. For all other equations $\alpha \approx \tau$ then τ does not contain any data constructors.

The previous example is a uniform regular type system. Uniformity is important because it implies that all type terms have canonical forms. In other words, all semantically equivalent type terms are reducible to a unique type term.

Theorem 1 (Canonical Forms)**.** *If* Σ_{reg} *is a uniform regular type system, then there exists a function* $can : \tau(\Sigma_{reg}) \rightarrow \tau(\Sigma_{reg})$ *satisfying:*

1. $[\![can(\tau)]\!] = [\![\tau]\!]$.
2. $can(\tau) = can(\tau') \Leftrightarrow [\![\tau]\!] = [\![\tau']\!]$.

The term $can(\tau)$ *is called the canonical form of* τ*. Proved in [7].*

In summary, type terms denote precise sets of trees and subtyping is via subset inclusion. This is in contrast to other type systems where subtyping is based on the syntax of type terms. Type inference may generate large type terms, but these will be simplified by computing canonical forms. Consider this type term:

$$(\alpha_{\mathsf{Pred}} \cup \mathsf{true}) \cap \left[\begin{array}{c}\mathsf{And}(\mathsf{Not}(\mathsf{true}), \alpha_{\mathsf{Pred}}) \;\cup\; \mathsf{And}(\mathsf{Not}(\mathsf{false}), \mathsf{Boolean}) \cup \\ \mathsf{And}(\mathsf{Not}(\mathsf{false}), \alpha_{\mathsf{Var}} \cup \alpha_{\mathsf{Not}} \cup \alpha_{\mathsf{And}})\end{array}\right]$$

Its canonical form is simply $\mathsf{And}(\mathsf{Not}(\mathsf{Boolean}), \alpha_{\mathsf{Pred}})$.

4.2 A Preview of Type Inference

Consider once again the model transformation rule:

FIND: A transition t s.t. $src(t) = s_a$, $dst(t) = s_b$, and $guard(t) = g_{OCL}$.
CREATE: If(current $==$ $id(s_a)$ && g_{OCL}) { current $=$ $id(s_b)$; }.

This rule can now be formally stated using constraints over ADTs.

FIND: $t \approx \mathsf{Trans}(s_a, g_{OCL}, s_b)$, $s_a \approx \mathsf{State}(i)$, $s_b \approx \mathsf{State}(j)$, and $t : \alpha_{\mathsf{Trans}}$.
CREATE: If(And(Eq(Var("current"), i), g_{OCL}), Set(Var("current"), j)).

The constraints are satisfied for some substitutions of the variables by elements of U. The type constraint $t : \alpha_{\mathsf{Trans}}$ requires t to be assigned to an element of $[\![\alpha_{\mathsf{Trans}}]\!]$. For example:

$$\begin{array}{c} g_{OCL} \mapsto \mathsf{true}, \quad i \mapsto 1, \quad j \mapsto 2, \\ s_a \mapsto \mathsf{State}(1), \quad s_b \mapsto \mathsf{State}(2), \quad t \mapsto \mathsf{Trans}(\mathsf{State}(1), \mathsf{true}, \mathsf{State}(2)). \end{array}$$

The goal of type inference is to deduce a variable-wise over-approximation of the satisfying assignments using regular types. Our type inference would infer the following types for the *find* portion of the rule:

$$g_{OCL} : \alpha_{\mathsf{Pred\text{-}OCL}}, \quad i, j : \mathsf{Integer}, \quad s_a, s_b : \alpha_{\mathsf{State}}, \quad t : \alpha_{\mathsf{Trans}}.$$

Note that type judgments are expressed as implied type constraints. This inference can be used to check if the *create* portion of the rule is a subtype of valid C ASTs.

5 Constraints with Interpreted Functions

The previous example illustrated a special case where specifications only contain data constructors, equalities, and type constraints. Clearly, these kinds of specifications fit well with regular types. Suppose variables $\boldsymbol{x}$ are judged to have types $\boldsymbol{\tau}$ and that $y \approx f(\boldsymbol{x})$ appears as a constraint. Then type judgments can be propagated through data constructors yielding the judgment $y : f(\boldsymbol{\tau})$. However, specifications can contain other kinds of constraints and functions. For example:

$$\boxed{\begin{array}{rl} o & : U^n \nrightarrow U \\ \gamma^o_\uparrow & : \tau(\Sigma_{reg})^n \to \tau(\Sigma_{reg}) \\ \gamma^o_\downarrow & : \tau(\Sigma_{reg}) \to \tau(\Sigma_{reg})^n \end{array}}$$

(L1)	$\gamma^o_\uparrow(\bot) = \bot$ and $\gamma^o_\downarrow(\bot) = \bot$.	(G1)	$\boldsymbol{\tau'} <: \boldsymbol{\tau} \Rightarrow \gamma^o_\uparrow(\boldsymbol{\tau'}) <: \gamma^o_\uparrow(\boldsymbol{\tau})$.
(L2)	$o([\![\boldsymbol{\tau}]\!]) \subseteq [\![\gamma^o_\uparrow(\boldsymbol{\tau})]\!]$.	(G2)	$\tau' <: \tau \Rightarrow \gamma^o_\downarrow(\tau') <: \gamma^o_\downarrow(\tau)$.
(L3)	$o^{-1}([\![\tau]\!]) \subseteq [\![\gamma^o_\downarrow(\tau)]\!]$.	(G3)	$\gamma^o_\uparrow(\boldsymbol{\tau}) <: \tau' \Leftrightarrow \boldsymbol{\tau} <: \gamma^o_\downarrow(\tau')$.

where $o([\![\boldsymbol{\tau}]\!])$ and $o^{-1}([\![\tau]\!])$ are the image and inverse image of o.

Fig. 3. Galois approximations of functions

$$t \approx \mathsf{Trans}(s_a, g, s_b),\ s_a \approx \mathsf{State}(i),\ s_b \approx \mathsf{State}(j),\ \text{and } i - j \geq 0$$

The subtraction function is not a data constructor; neither can the $\geq$ relation be described in terms of equality or type constraints. Instead, the interpretations of $-$ and $\geq$ are fixed by the theory of arithmetic. For this reason, we refer to them as *interpreted functions*. In this section we develop a method to reason about the effects of interpreted functions on regular types.

Let $o : U^n \nrightarrow U$ be an n-ary interpreted partial function, i.e. it satisfies equalities other than the trivial equality $\forall \boldsymbol{x}.\ o(\boldsymbol{x}) = o(\boldsymbol{x})$. An interpreted function with image $\{$ true, false $\}$ is called an *interpreted relation*. We handle interpreted functions using two type-level approximations for propagating type information:

$$\gamma^o_\uparrow : \tau(\Sigma_{reg})^n \to \tau(\Sigma_{reg}). \quad \gamma^o_\downarrow : \tau(\Sigma_{reg}) \to \tau(\Sigma_{reg})^n.$$

The *upward approximation* $\gamma^o_\uparrow$ propagates type information from inside to outside of applications. Suppose $k \approx i - j$ and i : PosInteger, j : NegInteger. Then the upward approximation of $-$ determines that k : PosInteger by computing $\gamma^-_\uparrow(\mathsf{PosInteger}, \mathsf{NegInteger}) \overset{def}{=} \mathsf{PosInteger}$. The *downward approximation* propagates type information from outside to inside of applications. Given k : PosInteger then $\gamma^-_\downarrow(\mathsf{PosInteger}) \overset{def}{=} (\mathsf{Real}, \mathsf{Real})$. The downward approximation indicates that i and j must be reals, but it cannot constrain the argument types further. The Lifting Axioms (L1)-(L3) in Figure 3 formalize the relationship between interpreted functions and approximations.

The approximations must be chosen so they always converge on a most precise type. A well-known approach for constructing converging approximations is via *Galois Connections*. The Galois axioms (G1)-(G3) in Figure 3 state the additional requirements. Axioms (G1) and (G2) require approximations to be monotone w.r.t. subtyping. Axiom (G3) is the key: The upward and downward approximations are always compatible, but the upward approximation can be more precise. Consider these two derivations, where $[\![\top]\!] \overset{def}{=} U$ denotes all possible values; initially it is known that i : PosInteger, j : NegInteger and $k \approx i - j$:

Table 1. Galois Approximation rules for +

$\gamma_\uparrow^+$	τ_1	τ_2	$\lambda\tau_1,\tau_2.$	$\gamma_\downarrow^+\ \tau$	$\lambda\tau.$
0	$\bot$	$\top$	$\bot$	$\bot$	$(\bot,\bot)$
1	$\top$	$\bot$	$\bot$	$\top$	$(\mathsf{Real},\mathsf{Real})$
2	0	$\top$	$\tau_2 \cap \mathsf{Real}$		
3	$\top$	0	$\tau_1 \cap \mathsf{Real}$		
4	c_1	c_2	$c_1 + c_2$		
5	PosInteger	PosInteger	PosInteger		
6	NegInteger	NegInteger	NegInteger		
7	Natural	PosInteger	PosInteger		
⋮	⋮	⋮	⋮		
N - 1	Integer	Integer	Integer		
N	Real	Real	Real		

$$(1)\qquad k:\top \overset{by\ \gamma_\downarrow^-}{\Longrightarrow} i:\mathsf{Real}, j:\mathsf{Real} \overset{by\ \gamma_\uparrow^-}{\Longrightarrow} k:\mathsf{PosInteger}.$$

$$(2)\qquad \text{from initial} \overset{by\ \gamma_\uparrow^-}{\Longrightarrow} k:\mathsf{PosInteger} \overset{by\ \gamma_\downarrow^-}{\Longrightarrow} i:\mathsf{Real}, j:\mathsf{Real}.$$

The first derivation assumes the most general type for k, and then applies the downward approximation to conclude that i and j are reals. However, they are already known to be subtypes of Real, so their most precise type does not change. Next, the upward approximation lowers the type of k from $\top$ to PosInteger. The second derivation applies $\gamma_\uparrow^-$ before $\gamma_\downarrow^-$. In the end, both derivations agree on the must precise types for the variables.

Galois approximations are a natural way to extend interpreted functions and relations into the regular type system. Regular types already incorporate precise Galois approximations.

Theorem 2 (Galois Approximations of Data Constructors)**.** *For any data constructor f, then $\gamma_\uparrow^f, \gamma_\downarrow^f$ are Galois Approximations of f. The approximations are precise, i.e. $\subseteq$ can be replaced with $=$ in Axioms (*L2*)-(*L3*) and $\gamma_\downarrow^f(\gamma_\uparrow^f(\boldsymbol{\tau})) = \boldsymbol{\tau}$ holds. The approximations are defined to be:*

$$\gamma_\uparrow^f(\boldsymbol{\tau}) \overset{def}{=} f(\boldsymbol{\tau}). \quad \gamma_\downarrow^f(f(\boldsymbol{\tau})) \overset{def}{=} \boldsymbol{\tau}. \quad \gamma_\downarrow^f(\beta) \overset{def}{=} \gamma_\downarrow^f(c) \overset{def}{=} \gamma_\downarrow^f(g(\boldsymbol{\tau'})) \overset{def}{=} \bot.$$

In all other cases τ can expressed as a union type $\tau \overset{def}{=} \tau_1 \cup \ldots \cup \tau_n$. For union types let $\gamma_\downarrow^f(\tau) \overset{def}{=} \gamma_\downarrow^f(\tau_1) \cup \ldots \cup \gamma_\downarrow^f(\tau_n)$.

We shall make use of the extra precision for data constructors by handling them specially during type inference.

5.1 Approximations by Tables

Unlike data constructors, interpreted functions do not come with built-in approximations. However, Galois Approximations provide a modular mechanism to soundly integrate new operators into the type system, and to make trade-offs between the precision and speed of type inference. One convenient method of defining Galois Approximations is by tables, as shown in Table 1. Each table contains a sequence of rows $(\boldsymbol{\tau}_i, \lambda_i)$ where λ_i is a function from type terms to type terms. It defines upward / downward approximations as follows:

$$\gamma^o_\uparrow(\boldsymbol{\tau}) \overset{def}{=} \lambda_k(\boldsymbol{\tau}),\ \text{for } k \overset{def}{=} min\{\ i \mid \boldsymbol{\tau} <: \boldsymbol{\tau}_i\ \}.$$
$$\gamma^o_\downarrow(\tau) \overset{def}{=} \lambda_k(\tau),\ \ \text{for } k \overset{def}{=} min\{\ i \mid \tau <: \tau_i\ \}.$$

To compute the upward (or downward) approximation for $\boldsymbol{\tau}$, find the smallest row k of which $\boldsymbol{\tau}$ is a subtype and then compute $\lambda_k(\boldsymbol{\tau})$. In our implementation of FORMULA the type system is parameterized by approximation tables making it easy to introduce new approximations. Note that row 4 is a schema for a an infinite number of rows, one for each possible pair of constants c_1, c_2. These shorthands are also supported in FORMULA.

6 Type Inference

The input to the type inference algorithm is a set of constraints $\mathcal{C}$ understood as a conjunction (minterm) together with a regular type system Σ_{reg}. A constraint is either an equality, disequality, a type constraint, or the application of an interpreted relation r, such as $t < t'$.

$$\mathsf{Constraint}\ \varphi ::= t \approx t' \mid t \not\approx t' \mid t : \tau \mid r(\boldsymbol{t}).$$

Type inference on constraints with disjunctions, which are not minterms, can be be accomplished by first converting the constraints into disjunctive normal form and then applying type inference algorithm to each minterm. If a variable appears in multiple minterms then its final type is the union of its inferred types in each minterm. For the remainder we focus only on minterms. Furthermore, we will consider minterms of the form $\mathcal{C} \wedge \mathcal{J}$, where $\mathcal{J}$ is a finite conjunction of canonized type constraints $t : can(\tau)$ and $\mathcal{C}$ is a conjunction of the other constraints (equalities, disequalities, and interpreted relations). Note that whenever a data constructor f is applied there is an implicit type constraint that the result is a member of α_f.

We now present our type inference procedure. Figure 4 gives saturation rules for inferring the types of subterms in $\mathcal{C} \wedge \mathcal{J}$. The procedure is modulo a decision procedure for satisfiability of the conjunction $\mathcal{C}$. FORMULA uses a combination of *congruence closure* [8] and Z3 [9] to check satisfiability of constraints that combine ADTs and arithmetic. We use this as a black box and concentrate on the rules for saturating with respect to type judgments. Initially all terms occurring in $\mathcal{C} \wedge \mathcal{J}$ are assigned the most permissive type $\top$. We also enforce that data constructors respect the type equations of Σ_{reg}. Equality constraints force two terms to have the same type. If $x : \tau$, $y : \tau'$ and $x \approx y$ is deduced, then $x, y : can(\tau \cap \tau')$ is concluded. Using the canonizer on the $(\cap)$ rule ensures that only a finite number of new types are introduced. On the other hand, if

Init	$\mathcal{C} \wedge \mathcal{J} \Longrightarrow \mathcal{C} \wedge \mathcal{J} \wedge t : \top$	If t is a term in $\mathcal{C}$.
Data	$\mathcal{C} \wedge \mathcal{J} \Longrightarrow \mathcal{C} \wedge \mathcal{J} \wedge f(\boldsymbol{t}) : f(\boldsymbol{\tau})$	If $f(\boldsymbol{t})$ is a term in $\mathcal{C}$ and E contains $\alpha_f \approx f(\boldsymbol{\tau})$.
Const	$\mathcal{C} \wedge \mathcal{J} \Longrightarrow \mathcal{C} \wedge \mathcal{J} \wedge c : c$	If c is a datatype constant in $\mathcal{C}$ or $\mathcal{J}$
Eq	$\mathcal{C} \wedge \mathcal{J} \Longrightarrow \mathcal{C} \wedge t \approx t_g \wedge \mathcal{J}$	If $(t : t_g) \in \mathcal{J}$
$(\cap)$	$\mathcal{C} \wedge \mathcal{J} \Longrightarrow \mathcal{C} \wedge \mathcal{J} \wedge t : \mathit{can}(\tau \cap \tau')$	If $(t : \tau), (t' : \tau') \in \mathcal{J}$ and $\mathcal{C} \models t \approx t'$
$(\Uparrow_f)$	$\mathcal{C} \wedge \mathcal{J} \Longrightarrow \mathcal{C} \wedge \mathcal{J} \wedge f(\boldsymbol{t}) : f(\boldsymbol{\tau})$	If $(\boldsymbol{t} : \boldsymbol{\tau}) \in \mathcal{J}$, $f(\boldsymbol{t})$ occurs in $\mathcal{J}$ or $\mathcal{C}$
$(\Downarrow_f)$	$\mathcal{C} \wedge \mathcal{J} \Longrightarrow \mathcal{C} \wedge \mathcal{J} \wedge \boldsymbol{t} : \boldsymbol{\tau}$	If $(f(\boldsymbol{t}) : f(\boldsymbol{\tau})) \in \mathcal{J}$
$(\Uparrow_o)$	$\mathcal{C} \wedge \mathcal{J} \Longrightarrow \mathcal{C} \wedge \mathcal{J} \wedge o(\boldsymbol{t}) : \gamma^o_\uparrow(\boldsymbol{\tau})$	If $(\boldsymbol{t} : \boldsymbol{\tau}) \in \mathcal{J}$, $o(\boldsymbol{t})$ occurs in $\mathcal{J}$ or $\mathcal{C}$
$(\Downarrow_o)$	$\mathcal{C} \wedge \mathcal{J} \Longrightarrow \mathcal{C} \wedge \mathcal{J} \wedge \boldsymbol{t} : \gamma^o_\downarrow(\tau)$	If $(o(\boldsymbol{t}) : \tau) \in \mathcal{J}$
Unsat	$\mathcal{C} \wedge \mathcal{J} \Longrightarrow \mathsf{Unsat}$	If $\mathcal{C}$ is unsatisfiable or $t : \bot$ is in $\mathcal{J}$.

Fig. 4. Saturation Rules for Type Inference

the type of a term t is a type expression t_g that has only one value (in other words, t_g is a ground expression built from constructors and constants), then the term is necessarily equal to the constant t_g, that is $t \approx t_g$. The $(\Uparrow)$ and $(\Downarrow)$ rules propagate type information from inside to outside of applications and vice versa. The $(\Uparrow_o)/(\Downarrow_o)$ rules use the Galois approximations to propagate types. We can assume that the types produced by the Galois approxiations are canonical.

It follows by inspection of the rules that our procedure infers only valid consequences:

Lemma 3 (Soundness)**.** *Each rule preserves satisfiability of $\mathcal{C} \wedge \mathcal{J}$.*

Lemma 4 (Termination)**.** *Assuming $\mathcal{C}$ has a decidable theory, then saturating $\mathcal{C} \wedge \mathcal{J}$ with respect to the rules in Figure 4 terminates.*

Proof. First we observe that the saturation rules only create judgments for existing terms in $\mathcal{C} \wedge \mathcal{J}$. So there is only a finite set of terms t that are used in judgments $t : \tau$ in $\mathcal{J}$. The set of canonical types used in judgments is also finite because each judgment creates a descending chain of *smaller*, that is more constrained types. There is only a finite set of base types, so this process terminates.

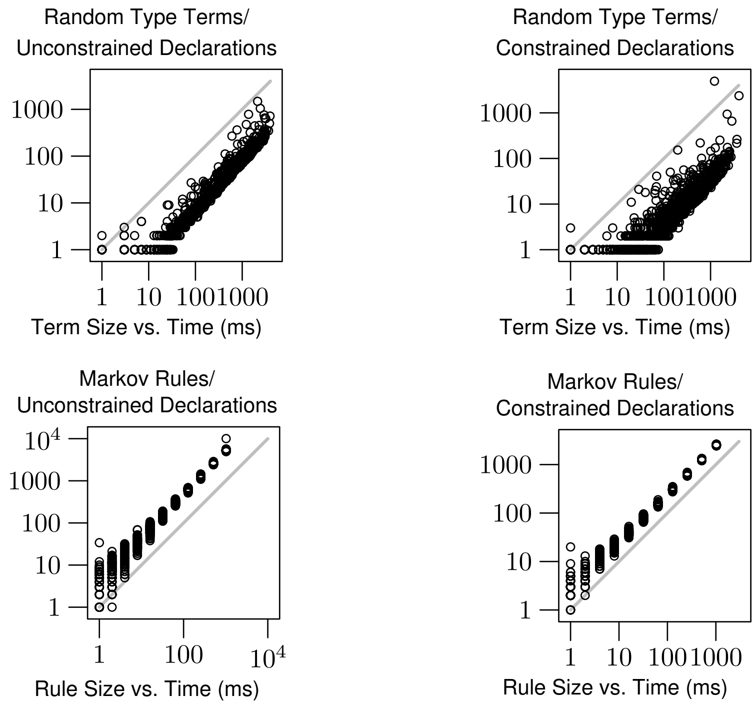
Thus, we have established that the proposed saturation procedure is an effective approach for type checking constraints in combination with regular types. But we can also establish a tighter relation between regular types and constraints in some cases. To formulate this property let us define the notion of *Best Type.*

Definition 4 (Best Type)**.** Given a constraint $\mathcal{C} \wedge \mathcal{J}$ over variables $\boldsymbol{x}$, the best type is a judgment $\boldsymbol{x} : \boldsymbol{\tau}$, such that for every variable x_i with judgment $x_i : \tau_i$, and every value $t \in [\![\tau_i]\!]$, there is a model $\mathcal{M}$ for which $\mathcal{M} \models \mathcal{C} \wedge \mathcal{J} \wedge x_i \approx t$.

Saturation with respect to Figure 4 produces best types in a very useful case:

Theorem 5 (Best types for Data Constructors)**.** *Suppose that $\mathcal{C}$ contains only equality and type constraints over data constructors, then saturation computes the best type (or reports that $\mathcal{C}$ is unsatisfiable).*

Table 2. Top row: Canonization times for random type expressions. Bottom row: Type inference times for LP rules generated by a Markov process.



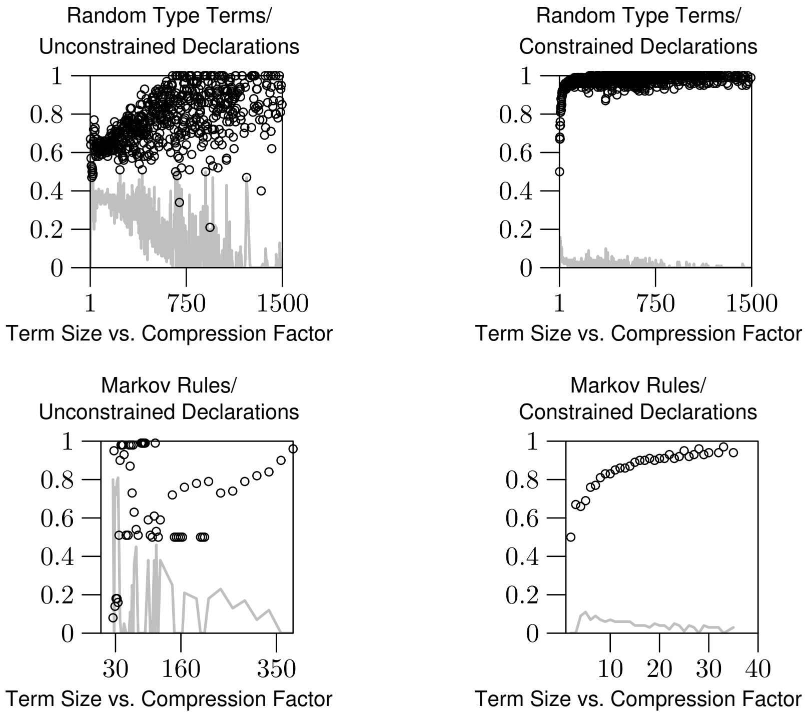
Proof. (Sketch) The theory of Data Constructors is decidable by applying unification with respect to all equalities. The most general unifier produces the set of equalities that must hold in all models. Theorem 2 ensured that $\gamma^f_\downarrow(\gamma^f_\uparrow(\boldsymbol{\tau})) = \boldsymbol{\tau}$, so the saturation rules do not introduce any approximations with respect to the Data Constructor theory.

We presented saturation rules that used tables to resolve type constraints for interpreted functions. This is the basis of type checking in FORMULA, which furthermore also propagates constraints from type judgments over basic constraints. For example t : $\mathsf{PosInteger}$ produces the constraints $\lfloor t \rfloor \approx t \wedge t > 0$.

7 Experimental Results

Type inference using Canoical Regular Types and Galois Approximations has been implemented in our formal specification language FORMULA. In theory, computing canonical forms may take exponential time. In practice, type inference is rarely slow, so we developed several benchmarks to stress-test our algorithms on unusually large type terms and rules. We evaluated our benchmarks using two representative type systems. The first type system, called *constrained*

Table 3. Average compression factors for non-canonical terms of various sizes. Bigger numbers indicate more compression and smaller canonical forms. Gray lines indicate the absolute distances of medians from means (skew).



declarations, came from a typical FORMULA program. There were around 20 distinct data constructors, type equations contained many union types, and there were strong type constraints on constructor arguments. The second type system, called *unconstrained declarations*, was atypically *unconstrained* having only the type equations:

$$E \overset{def}{=} \{\alpha_{f_i} \approx f_i(\top, \top) \mid i = 1 \ldots 8\}. \tag{1}$$

These equations permitted type expressions with arbitrary nestings of f_i applications. We expected longer times for canonization of random type terms over unconstrained declarations because there were more non-equivalent type expressions.

We developed two benchmarks and ran each against the two type systems. The *Random Type Terms* benchmark consisted of generating 1200 random type terms and measuring the time to canonize each term. This benchmark was purely to measure the impact of canonization, which is called many times during type inference. Random type terms had a maximum constructor depth of 12 and were guaranteed not to be trivially empty. Terms were biased towards union types, since unions can lead to an exponential increase in the size of canonical forms.

These random type terms do not occur in typical programs. In the results tables *term size* means the number of subterms.

The *Markov Rules* benchmark generated random specification rules through a Markov process and measured the time of type inference for the entire rule. The goal of this benchmark was to test type inference on large rules. FORMULA rules are based on logic programming and have the structure:

$$f(\boldsymbol{t}) \ :- \ f_1(\boldsymbol{t}_1), \ldots, f_n(\boldsymbol{t}_n), \ \ r_1(\boldsymbol{t}'_1), \ldots, r_m(\boldsymbol{t}'_m).$$

where $f_i(\boldsymbol{t}_i)$ and $r_j(\boldsymbol{t}'_j)$ are constructor applications and constraints. The Markov process for generating these rules was based on key program features we measured from actual logic programs: The rules of an average program perform shallow matchings on terms (an average term depth of 1.5). The average rule has many "don't care" variables; the average ratio of variable occurrences to distinct variables in a rule is also around 1.5. Both of these features impact the type expressions that will be encountered in an average program. We generated 100 rules for each $n = 2^k$ and $1 \leq k \leq 10$.

Table 2 shows the times measured for the combinations of benchmarks and type systems. The results show that canonization scales for complex type terms and for large program fragments respecting observed structure. The average canonization and type inference times for constrained and unconstrained declarations were similar, though constrained declarations exhibited more variance in run-times.

To test the quality of inferred types, we examined the size of non-canonical forms and their canonical forms after each benchmark. Define the compression factor of a non-canoical term τ to be $1 - \frac{size(can(\tau))}{size(\tau)}$. Factors near 1 indicate concise canonical forms that are much smaller than their non-canonical counterparts. Table 3 shows the compression factors for non-canonical terms as a function of term size across the various benchmarks. Each plot summarizes around 10^6 non-canonical terms (canonical terms are not included in these statistics). Note that small terms of size less than five are excluded from the plots, since they appear as noise. In general, the theoretical blow-up of canonical forms is not observed. Instead canonical forms are concise in practice and inferred types can be read by users.

8 Related Work

Regular types have a long history in programming, particularly logic programming (LP) [10], and are non-trivial to implement [11]. However, these approaches do not handle interpreted functions, nor do they produce predictable and comprehensible types. Historically, regular types are used to approximate the outcomes of untyped logic programs for static analysis [12] via abstract interpretation approaches [13]. State-of-the-art implementations use *non-deterministic tree automata* (NDTAs) for representing regular types and optimized automata algorithms for manipulating them [11,14]. The key algorithms for subtype testing are automata product, complement, and emptiness testing [15]. We uses an approach based on canonical forms [7], which has the advantage that type inference produces predictable and comprehensible type terms as opposed to tree automata. In the other direction, generalizations include *feature algebras* [16], extensions with arrow types [17], and refinement types [18]. Though we do not know of effective algorithms for deciding properties.

Regular types (without interpreted functions) exist in some other languages. The work of [19] has been influential on type declarations in LP languages. The authors showed that regular types are a subclass of logic programs and can be viewed as syntactic sugar over untyped logic programs. This is the approach taken by *Ciao-Prolog* [20], which is one of the few extant LP languages supporting regular types. The LP language *NU-Prolog* supports first-class type equations [10]. More recently, regular types have received much attention for statically verifying XML transformations. The languages *XDuce* [11] and *CDuce* [14] use regular types to prove that XML transformations always produced well-formed XML trees. Our earlier example of transforming between between expression languages is an exemplar.

9 Conclusions

We presented an approach to static error detection in declarative languages with constraints. Our approach has the unique property that satisfying assignments are summarized with type judgments, which are intended to be comprehensible by the user. Experimentally, our type judgments a small and can be computed quickly. Furthermore, canonicity ensures the user observes the same type terms whenever type inference infers semantically identical types. This is in contrast to other regular type systems where inference constructs complex tree-automata. Currently, every FORMULA specification passes through type inference during compilation, and we have found it to be a crucial tool to catch difficult specification errors.

Future work includes experimenting with more precise Galois Approximations. We believe the interval-style approximations familiar in abstract interpretation schemes would be particularly useful [21]. On the language side, we would also like better ways to identify which constraints are responsible for type errors or which constraints are responsible for an unexpectedly small inferred types. Because type inference is a saturation procedure over many constraints, it is not necessarily obvious how the constraints interact to result in the final inferred types.

References

1. Chang, F.S.H., Jackson, D.: Symbolic model checking of declarative relational models. In: ICSE, pp. 312–320 (2006)
2. Boronat, A., Meseguer, J.: An algebraic semantics for mof. Formal Asp. Comput. 22(3-4), 269–296 (2010)

3. Jackson, E.K., Levendovszky, T., Balasubramanian, D.: Reasoning about Metamodeling with Formal Specifications and Automatic Proofs. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 653–667. Springer, Heidelberg (2011)
4. Klar, F., Königs, A., Schürr, A.: Model transformation in the large. In: ESEC/SIGSOFT FSE, pp. 285–294 (2007)
5. Winkelmann, J., Taentzer, G., Ehrig, K., Küster, J.M.: Translation of restricted ocl constraints into graph constraints for generating meta model instances by graph grammars. Electr. Notes Theor. Comput. Sci. 211, 159–170 (2008)
6. Horváth, Á., Varró, D.: CSP(M): Constraint Satisfaction Problem over Models. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 107–121. Springer, Heidelberg (2009)
7. Jackson, E.K., Bjørner, N., Schulte, W.: Canonical regular types. In: ICLP (Technical Communications), pp. 73–83 (2011)
8. Nieuwenhuis, R., Oliveras, A.: Fast congruence closure and extensions. Inf. Comput. 205(4), 557–580 (2007)
9. de Moura, L.M., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
10. Dart, P.W., Zobel, J.: A Regular Type Language for Logic Programs. In: Types in Logic Programming, pp. 157–187. MIT Press (1992)
11. Hosoya, H., Vouillon, J., Pierce, B.C.: Regular expression types for XML. ACM Trans. Program. Lang. Syst. 27(1), 46–90 (2005)
12. Gallagher, J.P., Puebla, G.: Abstract Interpretation over Non-deterministic Finite Tree Automata for Set-Based Analysis of Logic Programs. In: Adsul, B., Ramakrishnan, C.R. (eds.) PADL 2002. LNCS, vol. 2257, pp. 243–261. Springer, Heidelberg (2002)
13. Cousot, P., Cousot, R.: Abstract Interpretation and Application to Logic Programs. J. Log. Program. 13(2&3), 103–179 (1992)
14. Benzaken, V., Castagna, G., Frisch, A.: CDuce: an XML-centric general-purpose language. In: Runciman, C., Shivers, O. (eds.) ICFP, pp. 51–63. ACM (2003)
15. Aiken, A., Murphy, B.R.: Implementing Regular Tree Expressions. In: Hughes, J. (ed.) FPCA 1991. LNCS, vol. 523, pp. 427–447. Springer, Heidelberg (1991)
16. Aït-Kaci, H., Podelski, A.: Towards a Meaning of LIFE. J. Log. Program 16(3), 195–234 (1993)
17. Frisch, A., Castagna, G., Benzaken, V.: Semantic subtyping: Dealing set-theoretically with function, union, intersection, and negation types. J. ACM 55(4) (2008)
18. Schäfer, M., de Moor, O.: Type inference for datalog with complex type hierarchies. In: POPL, pp. 145–156 (2010)
19. Fruhwirth, T., Shapiro, E., Vardi, M., Yardeni, E.: Logic programs as types for logic programs. In: LICS 1991, pp. 300–309 (1991)
20. Hermenegildo, M.V., Puebla, G., Bueno, F., López-García, P.: Integrated program debugging, verification, and optimization using abstract interpretation (and the Ciao system preprocessor). Sci. Comput. Program. 58(1-2), 115–140 (2005)
21. Chen, L., Miné, A., Wang, J., Cousot, P.: An Abstract Domain to Discover Interval Linear Equalities. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 112–128. Springer, Heidelberg (2010)

From UML and OCL to Relational Logic and Back

Mirco Kuhlmann and Martin Gogolla

University of Bremen, Computer Science Department
Database Systems Group, D-28334 Bremen
{mk,gogolla}@informatik.uni-bremen.de

Abstract. Languages like UML and OCL are used to precisely model systems. Complex UML and OCL models therefore represent a crucial part of model-driven development, as they formally specify the main system properties. Consequently, creating complete and correct models is a critical concern. For this purpose, we provide a lightweight model validation method based on efficient SAT solving techniques. In this paper, we present a transformation from UML class diagram and OCL concepts into relational logic. Relational logic in turn represents the source for advanced SAT-based model instance finders like Kodkod. This paper focuses on a natural transformation approach which aims to exploit the features of relational logic as directly as possible through straitening the handling of main UML and OCL features. This approach allows us to explicitly benefit from the efficient handling of relational logic in Kodkod and to interpret found results backwards in terms of UML and OCL.

1 Introduction

Creating complete and correct models is a critical concern. Modeling languages like UML [24] and OCL [30] allow for precisely specifying systems which often result in complex models. The analysis of formulated system properties thus requires tool support. Lightweight model validation approaches allow for agile analysis, since they allow modelers to automatically perform multiple validation tasks at any stage of development. The advantage of lightweight approaches, in contrast to interactive verification approaches, is (a) their applicability, as users do not need be familiar with fields like logical deduction, and (b) their immediateness regarding the feedback. As a consequence, those approaches must be *efficient*.

We analyze properties of UML class models annotated with OCL constraints by analyzing model instances [9], since the existence or non-existence of instances with specific properties allows direct conclusions about the model itself. For efficiently searching model instances, we apply SAT-based techniques [2], i. e., solvers for Boolean satisfiability. This approach requires the connection of UML and OCL with Boolean logic resulting in a bidirectional transformation. However, we make use of an intermediate language, *relational logic*, which is automatically and efficiently handled by the sophisticated model instance finder

Kodkod [28]. Kodkod transforms relational models into SAT formulas and translates solutions fulfilling the SAT formulas back into relational instances.

In this paper, we present the transformation of UML and OCL models into relational models, as well as the backward translation from relational instances into UML model instances. We pursue a natural transformation approach which aims to exploit the features of relational logic as directly as possible through straitening the handling of main UML and OCL features. This approach allows us to explicitly benefit from the efficient handling of relational logic in Kodkod. While explaining the transformation, we focus on important modeling aspects and concepts which have not been concerned or adequately treated in other UML and OCL model validation approaches based on relational logic [1,27], e. g., n-ary associations and association classes at the UML side, as well as the undefined value and essential operations like collect and navigation via n-ary associations and association classes at the OCL side. This transformation approach is supported by a tool classified as a model validator which processes a class diagram and OCL invariants as well as information (in form of partial object diagrams and properties like the minimum and maximum number of objects and links, or attribute value domains) determining the search space, that is, the set of model instances to be examined. The transformation is fully automated with respect to both directions, from UML and OCL to relational logic, and back from relational solutions to UML (object diagrams) (for an overview see [15]).

The rest of the paper is structured as follows: Section 2 introduces relevant concepts of relational logic and Kodkod. The main Sect. 3 presents the bidirectional transformation. In Sect. 3.1 we consider the transformation of UML class diagrams into relational models, while Sect. 3.2 discusses the backward translation. The configuration of search spaces is shortly sketched in Sect. 3.3. Section 3.4 covers the OCL part of the transformation. Related work is discussed in Sect. 4 before we conclude with Sect. 5.

2 Background: Relational Logic and Kodkod

Relational logic [10] is based on flat n-ary relations, i. e., sets of tuples of atomic values (atoms). The evaluation result of a relational formula thus depends on concrete instances of relations. Atoms are constants with no specific semantics or inner structure. The individual meaning of an atom emerges from its occurrence in specific relations. However, it is possible to assign a specific semantics to a subset of the available atoms by mapping them to integer values. Thereby, integer calculations are enabled. Relations can express three kinds of values:

Atomic Values: An atomic value is represented by a unary relation including exactly one tuple with one component holding the respective atom. For example, the integer value 3 and an atom symbolizing the name Ada are realized by the relational values `[[3]]` and `[[Ada]]`.

Sets of Atoms: A set of atomic values yields a unary relation with possibly more than one tuple or no tuple, in the case of an empty set. The set of atoms

{Ada,Bob,Cyd}, for example, results in the relational value `[[Ada],[Bob],[Cyd]]`. The atoms `Ada`, `Bob` and `Cyd` do not have a specific meaning, unless they are put into a context, e. g., if we declare a unary relation `fNames`, we consider all tuples within instances of this relation as individual first names.

Sets of Relationships between Atoms: Atomic values are often semantically related to other atomic values. This fact can be described with n-ary relations in which tuples hold sequences of atoms. Each position in a n-ary tuple has a specific meaning. Consider, for example, persons who have a name and possibly younger siblings. In order to relate a person to a name and her younger siblings, we can declare two binary relations `fName` and `ySiblings`, and determine that the first position of the tuples in both relations yields a person atom and the second position yields a name or another person atom, respectively. Possible instances could be `fName=[[p1,Ada],[p2,Cyd],[p3,Bob], [p4,Dan]]` and `ySiblings=[[p1,p3],[p1,p4]]`.

Relational logic provides: (a) relational operations like the relational join, product and transitive closure, as well as multiplicity predicates like 'some' and 'lone', (b) set comprehension, (c) set operations like union and subset, (d) Boolean operations like conjunction and implication, (e) quantifiers of first order logic-like existential and universal quantifiers, and (f) integer operations like addition and comparison predicates. The relational join (expressed by a dot `.`) is a central operation, since it allows for extracting and merging the information provided by relation instances. A join is performed in the context of two relational values `x` and `y` which may be of different arity. The evaluation result of the expression `x.y` is equal to $\{(x_1, \ldots, x_{n-1}, y_2, \ldots, y_m) | (x_1, \ldots, x_n) \in x \wedge (y_1, \ldots, y_m) \in y \wedge x_n = y_1\}$. An example for information extraction with a join is the determination of a person name based on the mentioned relation `fName`. The expression `[[p1]].fName` results in the name related to the person atom `p1`, i. e., in our example `[[Ada]]`. Another example illustrates the merging of two binary relations which in our case results in a set of tuples relating persons to the names of their younger siblings: `ySiblings.fName=[[p1,Bob],[p1,Dan]]`.

Kodkod is a tool which provides an interface to defining relational models and to efficiently finding relational instances fulfilling given relational formulas [28]. A relational model consists of three parts (we will see examples in later sections):

Declarations: A relation declaration determines the name and arity of a relation for which Kodkod searches a valid instance.

Bounds: Kodkod is a *finite* model instance finder, i. e., the *universe* of atoms available for constructing relational values is finite. A relational model includes (a) an a priori, fully determined universe of atoms, and (b) bounds for each declared relation which generally restrict the sets of possible tuples based on available atoms. In this way, a concrete search space is defined.

Constraints: Relational constraints, i. e., formulas, can further restrict the valid instances of the declared relations.

Our approach translates a UML and OCL model into a relational model handled by Kodkod. Results in form of relational model instances presented by Kodkod will be translated back into instances of the UML and OCL model.

3 A Bidirectional Transformation

The aim of translating UML and OCL models into relational models is an efficient search for UML and OCL model instances which fulfill specific user-defined properties. The transformation of UML class diagram and OCL concepts into relational logic is based on three key requirements:

– The transformation of a UML class diagram results in a set of relations. Instances of these relations must structurally allow for representing *all possible* instances of the corresponding UML class diagram.
– Each *valid* instance of the relational model must represent a *valid* instance of the corresponding UML class diagram respecting the given UML and OCL constraints. The same must apply for *invalid* instances.
– The relational model must be formulated as simply as possible, enabling the most efficient processing by the model instance finder (Kodkod).

UML and OCL offer concepts like collections (i. e., sets, bags, ordered sets, sequences, and nested collections) and a three valued logic which are fundamentally different from concepts of relational logic (e. g., solely flat sets and a two-valued logic). For that reason, the first two requirements which concern the completeness and correctness of the translation conflict with the third requirement concerning efficiency. We tackle two different approaches to transforming UML and OCL into relational logic, one giving weight to the first two aspects, the other focussing on the third aspect:

Extrinsic Relational Approach: This approach aims to transform UML and OCL concepts as completely as possible into relational logic enabling, for example, the translation of all kinds of (possibly nested) collections, strings and the associated operations. Furthermore, the three valued-logic of OCL is simulated at the relational level. This virtual *abuse* of relational logic leads to complex relational structures (involving high-arity relations), large search spaces, and hence to losses in efficiency (for details of the extrinsic approach see [14]).

Intrinsic Relational Approach: The intrinsic approach aims to make use of structures directly supported by relational logic, i. e., atomic values, sets of atomic values, and sets of relationships between atomic values (cf. Sect. 2), as well as relational formulas with two truth-values instead of three (as in OCL). On the one hand, this approach naturally results in manageable relational models which can be efficiently processed. On the other hand, it induces several restrictions to the supported UML and OCL features.

In this paper, we present the intrinsic transformation approach and discuss the advantages and disadvantages, practical implications for validation and feasible alternatives. The intrinsic approach has been successfully applied, for example, in the context of role-based access control (RBAC) revealing that the imposed restrictions do not hinder the validation of reasonable models [17].

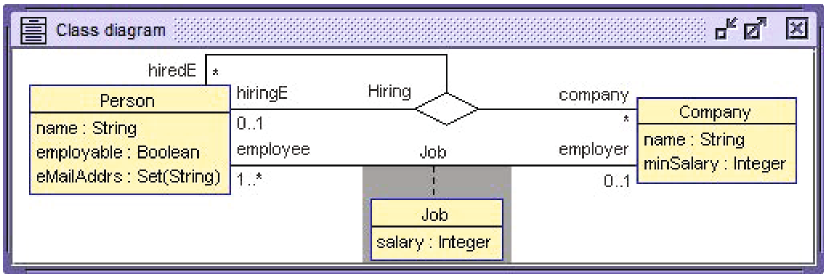


Fig. 1. Example UML Class Diagram

3.1 From UML Class Diagrams to Relational Models

In this section, we focus on the transformation of central UML class diagram features which are frequently used for modeling structural aspects of systems into relational model concepts, i. e., relations and relational constraints. The transformation is illustrated with the help of the example class diagram shown in Fig. 1 which has been designed for explanation purposes covering interesting aspects. It describes persons with a name and a set of email addresses. If employable, persons can have at most one job. A company has a name and defines a minimum salary for its employees. A person can be hired by at most one employee in the context of a specific company. In order to explain both, a binary association and a binary association class, we aim to consider *Job* as an ordinary association (neglecting the grey part), on the one hand, and to consider *Job* as an association class (involving the grey part), on the other hand. The association class adds a salary to each job.

Basic Types. The transformation t uniformly handles the values of the UML basic types *Boolean*, *Integer*, *Real*, and *String* as atomic values. Consequently, basic types result in unary relations whose instances hold the distinctive sets of available basic values, typing the atoms accordingly. Basic type values are needed in the context of UML attribute values, as we will see later.

Boolean $\xrightarrow{t}$ unary relation `Boolean=[[true],[false]]`. The resulting Boolean relation yields a constant instance holding the Boolean values `true` and `false`.

Integer $\xrightarrow{t}$ unary relation `Integer` of structure `[[`i_1`],...,[`$i_{n_{int}}$`]]`.
Example instance: `[[-2],[0],[1],[1000],[1100],[1200],[2000]]`. The integer relation can be variably instantiated, i. e., Kodkod searches an adequate instance. Each integer atom whose name represents an integer literal is bijectively mapped to a corresponding integer value which can be used for calculations within a relational formula. For instance, the atom `1` is mapped to the value 1. Relational logic provides the respective mapping operations (`int` and `Int`). In order to store calculation results in relations, the respective integer values must have an atomic counterpart within the integer relation, e. g., if the result of $1+2$ should be stored as a UML attribute value, the atom `3` must be available.

Real $\xrightarrow{t}$ unary relation `Real` of structure `[[`r_1`],... ,[`$r_{n_{Real}}$`]]`.
Example instance: `[[3.14],[2.71],[1.23]]`. The Real relation is analogously defined to the integer relation, but Real atoms cannot be mapped to processable Real values in relational logic. Thus, Real atoms do not have further meaning except for their comparability, e. g., we can infer that `[[3.14]]` does not equal `[[1.23]]`, but we cannot determine their precedence or apply Real operations.
String $\xrightarrow{t}$ unary relation `String` of structure `[[`s_1`],... ,[`$s_{n_{string}}$`]]`.
Example instance: `[[Ada],[Bob],[Apple],[IBM]]`. Relational logic does not directly support String values with an inner structure, i. e., consisting of sequences of characters. The intrinsic approach handles strings analogously to Real values.
Undef $\xrightarrow{t}$ unary relation `Undef=[[Undef]]`. Primitive values may be undefined. Hence, we need a unary singleton relation holding the undefined value.

Classes and Enumerations. Classes are translated into unary relations with variable instances; enumerations yield unary relations with constant instances:
Class c $\xrightarrow{t}$ unary relation `c` of structure `[[`obj_1`],...,[`obj_{n_c}`]]`, where an atom obj_i (with $1 \leq i \leq n_c$) represents an object identifier.
Example translation: Class *Person* $\xrightarrow{t}$ `Person`.
Example instance: `[[ada],[bob],[cyd]]`.
Enum e=$\{lit_1,\ldots,lit_{n_e}\}$ $\xrightarrow{t}$ unary relation `e=[[`lit_1`],...,[`lit_{n_e}`]]`.
Example translation: Enum *Colors*=$\{r,g,b\}$ $\xrightarrow{t}$ `Colors=[[r],[g],[b]]`.

Associations and Association Classes. The intrinsic transformation fully supports n-ary associations and association classes with multiplicities. An n-ary association has n association ends, where association end i (with $1 \leq i \leq n$) is of type class c_i, i. e., a navigation to this end results in objects of c_i. For translating associations into relational logic we determine a specific order of the association ends in such a way that end i is mapped to tuple position i. Hence, we obtain the following transformation for n-ary associations:
n**-ary Association** a $\xrightarrow{t}$ n-ary relation `a` of structure `[[`obj_{11}`,...,`obj_{1n}`],...,[`obj_{m1}`,...,`obj_{mn}`]]`, where obj_{ij} describes the object occurring in the ith link at the jth association end, plus typing and multiplicity constraints.
Example translation: Association *Hiring* with association end order: *hiringE*, *hiredE*, *company* $\xrightarrow{t}$ `Hiring` plus constraints shown below.
Example instance: `[[ada,bob,apple]]`. n-ary associations result in n typing constraints requiring each association end, i. e., each tuple position, to hold objects of the related class, i. e., atoms of the respective class relation. The universe relation `univ` provided by relational logic including all existing atoms allows us to navigate to the desired tuple positions by cutting off the unneeded tuple positions. Consider the following typing constraints for association relation `Hiring`:

`(Hiring.univ).univ in Person`	the hiring employee (first position) is a person
`(univ.Hiring).univ in Person`	the hired employee (second position) is a person
`univ.(univ.Hiring) in Company`	a person is hired for a company (third position)

Furthermore, each association end yielding a constraining multiplicity differing from *0..** results in a multiplicity constraint. Consider for example the constraint for association end *hiringE* which demands that each pair of objects belonging to the opposite association ends *hiredE* and *company* is connected to at most one object of association end *hiringE*:

```
all c2:Person, c3:Company | #((Hiring.c3).c2)<=1
```

If the lower bound of a multiplicity is greater than 0, the constraint is extended accordingly. Generally we see that the absence of a link is indicated by the absence of a corresponding tuple in the association relation. In this way, the navigation to an association end directly results in set values. Objects not linked to another object do not occur in the set. If no object is connected, the navigation results in an empty set. Binary associations are an exception to this rule if an association end is single-valued, i. e., if it yields the multiplicity *1* or *0..1*. In this case, a navigation to this end results in exactly one object. Multiplicity *0..1* allows this object to be undefined. Thus, the absence of a link is expressed by tuples having the `Undef` atom at the respective position. That is, in contrast to general association relations the absence of a link is not indicated by the absence of the respective tuple, but by the explicit occurrence of the undefined value:

Binary association $a \xrightarrow{t}$ binary relation `a` of structure `[[`obj_{11}`,`obj_{12}`],...,[`obj_{m1}`,`obj_{m2}`]]`, where obj_{ij} may be undefined, if association end j yields multiplicity *0..1*, plus special relational constraints for typing and multiplicities.
Example translation: Association *Job* with association end order: *employee*, *employer* (dismissing the grey association class part) $\xrightarrow{t}$ `Job`.
Example instance: `[[ada,ibm],[bob,ibm],[cyd,Undef]]`. Constraints:

`Job.univ in Person`	the employee is a person
`univ.Job in Company+Undef`	the employer is a defined company or undefined
`all c1:Person|#(c1.Job)=1`	a person is connected to one atom via relation `Job`
`all c2:Company|#(Job.c2)>=1`	a company is connected to at least one person

If we respect the grey part in Fig. 1, we obtain an association class. Association classes yield two relations. One relation represents the class perspective following the same translation rules as relations for ordinary classes. In every respect, the class relations of association classes can be handled like class relations of ordinary classes. The relation representing the association part is translated analogously to ordinary associations, except for an additional column at the first tuple position holding the participating association class objects:

n**-ary Association class** $ac \xrightarrow{t}$ unary relation `ac` of structure `[[`ac_obj_1`],...,[`ac_obj_m`]]`, n+1-ary relation `ac_assoc` of structure `[[`ac_obj_1`,`obj_{11}`,...,`obj_{1n}`],...,[`ac_obj_m`,`obj_{m1}`,...,`obj_{mn}`]]`, plus typing and multiplicity constraints.
Example translation: Association class *Job* with association end order: *job* (implicit), *employee*, *employer* (respecting the grey part) $\xrightarrow{t}$ `Job`, `Job_assoc`.
Example instance of `Job`: `[[job1],[job2]]`. Example instance of `Job_assoc`: `[[job1,ada,ibm],[job2,bob,ibm],[Undef,cyd,Undef]]`. As ordinary association ends, association class ends are typed:

```
(Job_assoc.univ).univ in Job+Undef
```

Furthermore, the association class end requires two multiplicity constraints for ensuring that (a) each permutation of objects corresponding to the opposite ends is connected to at most one association class object, and (b) each association class object is connected to exactly one permutation of *defined* objects:

(a) `all c2:Person, c3:Company|#((Job_assoc.c3).c2)<=1`

(b) `all c1:Job | #(c1.Job_assoc)=1 && (c1.Job_assoc) in (Person->Company)`

Analogously to binary associations, binary association classes need a special handling if single-valued association ends are involved. In the case of an object-valued association end like *employer*, the opposite end (i. e., *employee* in our example) is always related to one object which may be undefined:

(c) `all c2:Person | #(c2.(univ.Job_assoc))=1`

In the case of set-valued association ends like *employee* with multiplicity *1..**, the opposite end (*employer*) is never linked to an undefined association class object because, in this case, the navigation to the association class end results in a set of objects (i. e., one or more jobs in our example):

(d) `all c3:Company|!(Undef in ((Job_assoc.c3).univ)) && #(Job_assoc.c3)>=1`

Attributes. Independent from their types, UML attributes are always translated into binary relations. Attribute relations relate objects with attribute values. If an attribute is not defined, the respective objects are related to the undefined value. In the case of set-valued attributes, we use the special atom `Undef_Set` to indicate the absence of a defined set. This way, we can distinguish between undefined set values (object related to `Undef_Set`), defined set values including the undefined value (object related to `Undef`) and an empty sets (the corresponding object does not participate in the attribute relation instance). Regarding this detail, the translation of attributes and binary associations differ.

Attribute *Class::attr* $\xrightarrow{t}$ binary relation `Class_attr` of structure `[[`obj_1`,` val_{11}`]`$,\ldots,$`[`obj_1`,`val_{1n_1}`]`$,\ldots,$`[`obj_m`,`val_{m1}`]`$,\ldots,$`[`obj_m`,`val_{mn_m}`]]`, where n_i is the number of atoms representing the attribute value related to obj_i $(1 \leq i \leq m)$, plus typing and multiplicity constraints. Basic, object and enumeration type attributes require $n_i = 1$ for all i. Set type attributes allow any positive value including 0 for n_i, also n_i and n_j $(1 \leq j \leq m$ and $i \neq j)$ may differ.
Example translation: Attribute *Person::name*, *Person::eMailAddrs*, *Job::salary* $\xrightarrow{t}$ `Person_name`, `Person_eMailAddrs`, `Job_salary`.
Example instance (`Person_name`): `[[ada,Ada],[bob,Bob],[cyd,Undef]]`.
Example instance (`Person_eMailAddrs`):
`[[ada,ada@apple.com],[ada,ada@gmail.com],[cyd,Undef_Set]]`.
Example instance (`Job_salary`): `[[job1,2000],[job2,1200]]`.

Attribute relations are constrained by formulas for determining the attribute domain, type and multiplicity. The attribute domain is always a class relation. The undefined value is not involved at the domain side. However, the undefined value always participates in the attributes type definition. Let us consider the constraints for the basic type attribute relation `Person_name`:

```
Person_name.univ in Person          the domain is Person
univ.Person_name in String+Undef    the type is String including Undef
all c:Person | #(c.Person_name)=1   the attribute relates a person to one atom
```

Set-valued attributes yield different constraints:

```
Person_eMailAddrs.univ in Person       the domain is Person
univ.Person_eMailAddrs in              the type is a set of String values in-
  String+Undef+Undef_Set               cluding undefined values
all c:Person |                         an undefined set is not accompanied
  Undef_Set in c.Person_eMailAddrs =>  by other values
    #(c.Person_eMailAddrs)=1
```

3.2 From Relational Instances to Class Diagram Instances

In this section, we consider the straightforward backward translation of a valid relational model instance provided by Kodkod into instances of UML class diagram concepts. We illustrate the transformation with the help of instances of relations resulting from the example class diagram shown in Fig. 1 including the grey association class part:

```
Boolean=[[true],[false]],           Integer=[[1000],[1100],[1200],[2000]],
String=[[Ada],[Bob],[Apple],[IBM]], Undef=[[Undef]],
Person=[[ada],[bob],[cyd]],         Company=[[apple],[ibm]],
Job=[[job1],[job2]],                Hiring=[[ada,bob,apple]],
Job_assoc=[[job1,ada,apple],[job2,bob,apple],[Undef,cyd,Undef]],
Person_name=[[ada,Ada],[bob,Bob],[cyd,Undef]],
Person_employable=[[ada,true],[bob,true],[cyd,false]],
Person_eMailAddrs=[[ada,ada@apple.com],[ada@gmail.com],[cyd,Undef_Set]],
Company_name=[[apple,Apple],[ibm,IBM]],
Job_minSalary=[[apple,1000],[ibm,1100]],
Job_salary=[[job1,2000],[job2,1200]]
```

These relation instances directly result in the class diagram instance visualized in the object diagram shown in Fig. 2.

3.3 User-Defined Search Space Configuration

For searching valid instances of relational models, Kodkod requires a restricted search space, i. e., a predetermined universe of atoms and bounds to the declared relations. Upper bounds determine the set of all possible tuples for each relation. Lower bounds, instead, declare sets of tuples which *must* occur in a valid instance, i. e., a partial solution. A comfortable way for specifying partial solutions is the translation of a partial user-defined object diagram into the lower bounds of the concerned relations. This forward translation can be done analogously to the backward translation illustrated in Sect. 3.2.

Since the search space directly influences the search efficiency of Kodkod, the aim is to minimize the upper bounds. Respective optimizations are in particular possible in the context of partial solutions, since the existence of specific tuples in

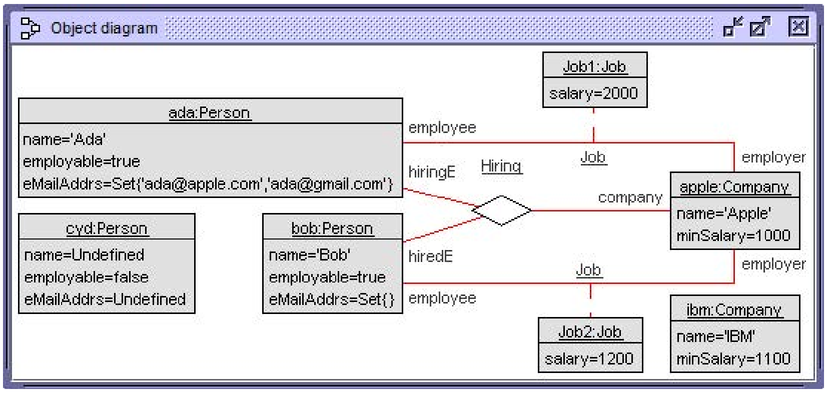


Fig. 2. Translation Result from Relation Instances to Class Diagram Instances

the lower bounds often preclude the existence of other tuples in a valid instance. Those tuples can be removed from the upper bounds, e. g., if a partial solution assigns the name `Ada` to object `ada`, the upper bounds of relation `Person_name` can be filtered with respect to tuples assigning other names to this object.

The search space configuration can be extended by relational constraints which, for example, determine the minimum and maximum numbers of defined links of a specific association, or attribute values of a specific attribute. Those properties cannot be configured by bounds, as they do not concern specific tuples.

An implementation of the considered transformation should provide means for easy configurations while hiding the particularities of relational logic, e. g., allowing the user to determine the minimum and maximum number of objects, forbidding specific links, or defining ranges of available attribute values.

3.4 From OCL Constraints to Relational Constraints

Class diagrams can be annotated with OCL invariants which constrain the set of valid class diagram instances. OCL invariants, representing Boolean OCL expressions, are transformed into relational formulas. Additionally, in our validation approach user-defined validation tasks specifying properties the searched model instance must fulfill are made available in form of temporary OCL constraints. In this section, we consider the translation of individual interesting and important OCL operations. The transformation of operations not discussed in this section can be inspected in [13].

Boolean Operations. The intrinsic transformation approach makes use of the two-valued relational logic. Consequently, Boolean OCL expressions result in relational formulas, in contrast to non-Boolean OCL expressions which result in relational expressions, i. e., relation instances. For example, consider the Boolean operation *xor* which is the only Boolean operation with no direct counterpart in relational logic:

$expr_1$ ***xor*** $expr_2 \xrightarrow{t}$ `(`$\overset{t}{\overrightarrow{expr}}_1$ `&& !`$\overset{t}{\overrightarrow{expr}}_2$`) || (!`$\overset{t}{\overrightarrow{expr}}_1$ `&&` $\overset{t}{\overrightarrow{expr}}_2$`)`, with $\overset{t}{\overrightarrow{e}}$ denoting the transformation result of OCL expression e into a relational expression or formula, respectively.

Since the Boolean values of relational formulas cannot be stored in relations, we define two relational operations (a) for mapping Boolean atoms (`true` and `false` which can occur as Boolean attribute values or as Boolean literals in OCL expressions) to relational truth values, and (b) for mapping relational truth values into atomic values:

(a) `expr2formula(e)`:`Formula` = `e=[[true]]`
(b) `formula2expr(f)`:`Expression` = `f => [[true]] else [[false]]` (if-then-else)

Operation (a) reveals that the three-valued logic of OCL is encoded into two-valued relational logic by mapping the undefined value to the value *false*. This realization can influence the validity of OCL invariants. Consider, for example, the OCL constraint $expr_1$ *and* $expr_2$ *implies* $expr_3$ which would evaluate to *Undefined*, and thus would be violated, if $expr_1$ evaluates to *Undefined* and the other expressions to *false*. The corresponding relational constraint, however, would be fulfilled. This disadvantage can be avoided by explicitly treating possible undefined values within a constraint, e. g., by applying explicit case distinctions and the OCL operation *oclIsUndefined*. As a consequence, the modeler has to be aware of situations in which an OCL expression can be undefined (which is anyway a preferable modeling style).

Integer, OclAny and Other Operations. Except for the explicit handling of the undefined value, integer operations are directly translated into their counterparts provided by relational logic. OclAny operations like *equality*, *inequality* or *oclIsUndefined* result in Boolean values, hence, requiring the application of the expression, formula mapping operations discussed before. However, their transformation is also straightforward. The distinct operations and statements *allInstances*, *let*, *if-then-else*, and the access of attribute values also yield plain relational constructs. For details see [13].

Set Operations. In the majority of cases, OCL set operations like *union*, *including*, *includes*, *forAll* or *exists* can be directly transformed into equivalent relational logic expressions or formulas, respectively. In this subsection, we consider the prominent set operation *collect* which, on the one hand, is often used for comfortably collecting specific (possibly calculated) values, on the other hand, is not handled in other works on translating OCL into relational logic. Furthermore, *collect* is implicitly applied for navigating a UML class diagram using the dot shortcut which we will consider later.

src->***collect***(v | *body(v)*) $\xrightarrow{t}$ $\overset{t}{\overrightarrow{src}}$`=[[Undef_Set]] => [[Undef_Set]] else`

`rflattenUndef(rcollect(v,`$\overset{t}{\overrightarrow{src}}$`,`$\overset{t}{\overrightarrow{body}}$`(v)))`,

where *body(v)* represents an arbitrary OCL expression in which variable *v* may occur, `rflattenUndef` and `rcollect` are relational operations which we have defined for transforming the OCL *collect*. The case distinction ensures that an undefined source collection (*src*) again results in an undefined collection.

The operation `rcollect` requires three arguments; a variable `v`, the translated source expression, and the translated body expression in which `v` may occur. First, this operation creates a binary relation via comprehension which relates each element of the source collection to the evaluation result of the respective body expression. For instance, the OCL expression *Set{1,2,3}->collect(i|i*i)* would yield the intermediate relation `[[1,1],[2,4],[3,9]]`. The transformation respects the fact that the result of *collect* must be flattened. If the body expression, results in a set of values, each element of the source collection is related to each element of this set via an individual tuple, i. e., the result is automatically flattened. After that, the first tuple position is cut off to obtain the desired evaluation result, e. g., with respect to the current example `[[1],[4],[9]]`:

$$\texttt{rcollect(v,}\overset{t}{\overrightarrow{src}}\texttt{,}\overset{t}{\overrightarrow{body}}\texttt{(v))} = \texttt{univ.}\{\texttt{v:}\overset{t}{\overrightarrow{src}}\texttt{, res:}\overset{t}{\overrightarrow{body}}\texttt{(v) | true}\}$$

The body of a *collect* expression can result in collection values which are implicitly flattened in the context of the OCL *collect*, e. g., the expression *Set{Undefined,Set{1}}->collect(i|i)* evaluates to *Bag{Undefined,1}* of type *Bag(Integer)*, while the source collection is of type *Set(Set(Integer))*. That is, undefined set-valued body expressions evaluate to an undefined value in the flattened result. For this reason, we need the operation `rflattenUndef` which checks if undefined collections (expressed by the atom `Undef_Set`) occur, and transforms them into `Undef` representing undefined single-values:

$$\texttt{rflattenUndef(e)} = \texttt{Undef\_Set in e => (e-Undef\_Set)+Undef else e}$$

Please note that the relational representation of *collect* always results in *sets* of values, while its OCL counterpart either results in bags or sequences, possibly yielding duplicate values and specific orders. The intrinsic approach thus restricts the expressiveness of *collect*. However, in many circumstances, not a specific order or the number of duplicate values is crucial, but the collection of distinct values. Let us consider this fact with the help of two concrete OCL invariants based on the class diagram shown in Fig. 1:

```
context c:Company
inv MinimumSalaryMaintained: c.job.salary->min() > c.minSalary
inv HiringPersonEmployed:
  c.hiringE->notEmpty() implies c.hiringE.employer->asSet()=Set{c}
```

The first invariant ensures in the context of a company the lowest paid job to yield a salary higher than the minimum salary determined by the company. The expression *c.job.salary* implicitly applies a *collect* via the dot shortcut, collecting all salaries for each job. The aim is to obtain the lowest salary. The number of employees yielding the lowest salary is irrelevant. The other invariant ensures that persons can only hire employees for their own company. Again, the only purpose of expression *c.hiringE.employer->asSet* is to collect the *distinct* employers of persons who hire for company *c*. Consequently, despite the restrictions, the intrinsical approach supports a large variety of practical models.

Navigation. Our transformation approach allows for navigating arbitrary reflexive and non-reflexive n-ary associations and association classes. We consider the general OCL navigation expression *expr.role* representing the navigation via association *assoc* from the evaluation result of *expr* (which yields a defined or undefined object), i. e., from association end i, to the *role* at association end j. For keeping the translation clear, we introduce the auxiliary operations `univ_r` and `univ_l` which represent multiple applications of universe joins from the right or the left side, respectively:

$univ\_r(\mathtt{e}, n) = \mathit{if}\ \ n > 0\ \mathit{then}\ univ\_r(e, n-1)\mathtt{.univ}\ \mathit{else}\ \mathtt{e}$
$univ\_l(\mathtt{e}, n) = \mathit{if}\ \ n > 0\ \mathit{then}\ \mathtt{univ.}\,univ\_l(e, n-1)\ \mathit{else}\ \mathtt{e}$
Example: $univ\_r(\mathtt{e}, 3) = \mathtt{e.univ.univ.univ}$

expr.role (via n-ary association *assoc* from association end i to end j) $\xrightarrow{t}$

$\overset{t}{\overrightarrow{expr}}\mathtt{=[[Undef]]\ =>\ [[}uv\mathtt{]]\ else}$

if $i < j$ then $univ\_r(univ\_l(\overset{t}{\overrightarrow{expr}}\,\mathtt{.}\,univ\_l(\mathtt{assoc}, i-1), j-i-1), n-j)$

else $univ\_l(univ\_r(univ\_r(\mathtt{assoc}, n-i)\,\mathtt{.}\,\overset{t}{\overrightarrow{expr}}, i-j-1), j-1)$,
where uv is equal to `Undef_Set` if association end j is set-valued, and uv is equal to `Undef` if end j is object-valued.

Let us consider some example navigation expressions based on association *Hiring* and association class *Job* shown in Fig. [1]:
apple.hiringE (from association end 3 to end 1) $\xrightarrow{t}$ `(Hiring.[[apple]]).univ`.
apple.hiredE (from end 3 to end 2) $\xrightarrow{t}$ `univ.(Hiring.[[apple]])`.
bob.company[hiredE][1] (from end 2 to end 3) $\xrightarrow{t}$ `[[bob]].(univ.Hiring)`.
ada.job (from end 2 to end 1) $\xrightarrow{t}$ `(Job_assoc.univ).[[ada]]`

As we have mentioned before, the dot shortcut, i. e., an implicit *collect*, provided by OCL allows us to easily collect objects while navigating through a class diagram, i. e., via more than association. Consider, for instance, the expression *apple.hiringE.employer* including an ordinary navigation starting from an object (*apple*), as well as an implicit *collect* based on the navigation result which further navigates to association end *employer*. This shortcut expression is equivalent to *apple.hiringE*`->`*collect(p|p.employer)*. A (complete) transformation of this expression is shown at the end of this section.

Our transformation approach allows us to differentiate between three distinctive cases which is required by OCL. (a) If *expr* within *expr.hiringE.employer* is undefined, the whole expression results in an *undefined set*. (b) If *expr.hiringE* results in a defined set including at least one unemployed person, the whole shortcut expression results in a *set including the undefined value*. (c) If *expr.hiringE* results in an empty set, the whole expression results in an *empty set*. These meaningful cases cannot be expressed by approaches like [1] due to language restrictions with respect to Alloy.

[1] Since the association is reflexive, i. e., persons can participate in *Hiring* links in different roles, the association end from which the navigation starts must be determined within brackets if ambiguous.

`rflattenUndef(rcollect(p,`$\overset{\xrightarrow{t}}{apple.hiringE}$`,`$\overset{\xrightarrow{t}}{p.employer}$`))` $=$

$\overset{\xrightarrow{t}}{apple.hiringE}$`=[[Undef_Set]] => [[Undef_Set]] else`

  `(Undef_Set in univ.{p:`$\overset{\xrightarrow{t}}{apple.hiringE}$`, res:`$\overset{\xrightarrow{t}}{p.employer}$` | true} =>`

   `((univ.{p:`$\overset{\xrightarrow{t}}{apple.hiringE}$`, res:`$\overset{\xrightarrow{t}}{p.employer}$` | true})-Undef_Set)+Undef`

  `else univ.{p:`$\overset{\xrightarrow{t}}{apple.hiringE}$`, res:`$\overset{\xrightarrow{t}}{p.employer}$` | true})`, with

$\overset{\xrightarrow{t}}{apple.hiringE}$ $=$
`[[apple]]=[[Undef]] => [[Undef_Set]] else (Hiring.[[apple]]).univ`, and

$\overset{\xrightarrow{t}}{p.employer}$ $=$ `p=[[Undef]] => [[Undef]] else p.(univ.Job_assoc)`

4 Related Work

While there are many important approaches in the field of UML and OCL model validation, in particular for information system validation [20], there is currently only one work following our approach to directly translating UML models into pure relational models [27]. The approach focuses on automatic resolution of model inconsistencies by translating basic class diagram concepts into relations and formulas. OCL as a whole and important UML features like n-ary associations, association classes, and undefined values have not yet been explicitly concerned.

OCLexec [12,11] makes use of Kodkod in order to generate Java method bodies by animating OCL operations constrained by OCL postconditions and invariants. In this approach, OCL expressions are translated into arithmetic expressions with bounded quantifiers and uninterpreted functions, i. e., pure integer expressions. The efficient mechanisms of Kodkod [28] are applied to transform those expressions into SAT problems. However, this approach has a loose connection to our work, since the authors of OCLexec '*do not make use of higher-level features of Kodkod such as encoding of relations*'. Thus, our transformation of UML and OCL concepts into relations and relational formulas is fundamentally different from the transformation result of OCLexec.

Our work is related to approaches which translate UML and OCL into the specification language Alloy [10] which is also based on relational logic. The so-called Alloy Analyzer transforms Alloy specifications into relational models supported by Kodkod. However, the modeling concepts provided by Alloy, e. g., signatures and fields, purposefully restrict the structure of specification components. That is, on the one hand, structures of Alloy specifications result in specific relational structures, but, on the other hand, not all relational structures supported by Kodkod can be modeled with Alloy. Consequently, several aspects of UML and OCL like the adequate handling of undefined values are not supported by Alloy, and thus are not directly realizable by approaches like UML2Alloy [1].

While UML2Alloy is an elaborated tool for validating UML and OCL models, it does not handle UML concepts like n-ary associations and association classes, or OCL operations like *collect*. The authors of CD2Alloy [18] pursue a deep embedding by defining class diagram constructs as new concepts within Alloy, enabling, for example, the comparison of two class diagrams. The work discussed in [19] aims to check the consistency between class and object diagrams by explicitly modeling object diagram concepts in Alloy. A backward transformation from original Alloy specifications into UML and OCL models is presented in [8]. The authors in [4] translate conceptual models described in OntoUML for validation purposes into Alloy.

Kodkod has been successfully applied in different fields, e. g., for executing declarative specifications in case of runtime exceptions in Java programs [25], reasoning about memory models [29], or generating counterexamples for Isabelle/HOL a proof assistant for higher-order logic (Nitpick) [3].

There are many other works concerning the validation of UML and OCL models which do not base on Alloy or Kodkod. For instance, a direct translation of UML and OCL concepts into SAT has been addressed in [26]. However, a direct translation cannot benefit from existing translation mechanisms like the sophisticated symmetry detection and breaking scheme which enables an efficient handling of partial solutions, or the detection and exploitation of redundant structures in formulas which are implemented in Kodkod. A translation of specific UML and OCL features into constraint satisfaction problems (CSP) is done in [6]. Answer set programming (ASP) [21], the constructive query containment (CQC) method [22], or rewriting-based techniques [23,7] are applied for analyzing static and dynamic model aspects. The named approaches differ from more interactive approaches like [5] involving verification by theorem proving.

5 Conclusion

In this paper we have presented the details of a bidirectional transformation from UML and OCL into relational logic and back, while focussing on the essential concepts of UML models and central OCL operations. Our so-called intrinsic approach implies restrictions at the UML and OCL side, but, on the one hand, enables the direct use of relational constructs, and, on the other hand, does still support a large variety of practically useful models.

Future work will comprise the finalization of our extrinsic approach which has been developed parallel to the current intrinsic approach. We will discuss a detailed comparison of (a) the intrinsic and extrinsic approach, and (b) our approaches and other relational and non-relational UML and OCL model validation approaches. A comparison will consider the supported UML and OCL features based on the OCL benchmark [16] as well as the efficiency with respect to models of different scale and purpose. Furthermore, the transformation will be extended regarding dynamic aspects, e. g., involving OCL pre- and postconditions, UML state machines, and sequence diagrams, and the mechanisms for specifying and optimizing the search space of model instances will be consolidated.

References

1. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. Software and System Modeling 9(1), 69–86 (2010)
2. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
3. Blanchette, J.C., Nipkow, T.: Nitpick: A Counterexample Generator for Higher-Order Logic Based on a Relational Model Finder. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 131–146. Springer, Heidelberg (2010)
4. Braga, B.F.B., Almeida, J.P.A., Guizzardi, G., Benevides, A.B.: Transforming OntoUML into Alloy: towards conceptual model validation using a lightweight formal method. ISSE 6(1-2), 55–63 (2010)
5. Brucker, A.D., Wolff, B.: HOL-OCL: A Formal Proof Environment for UML/OCL. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 97–100. Springer, Heidelberg (2008)
6. Cabot, J., Clariso, R., Riera, D.: Verification of UML/OCL Class Diagrams using Constraint Programming. In: IEEE International Conference on Software Testing Verification and Validation Workshop, ICSTW 2008, pp. 73–80 (April 2008)
7. Clavel, M., Egea, M.: ITP/OCL: A Rewriting-Based Validation Tool for UML+OCL Static Class Diagrams. In: Johnson, M., Vene, V. (eds.) AMAST 2006. LNCS, vol. 4019, pp. 368–373. Springer, Heidelberg (2006)
8. Garis, A.G., Cunha, A., Riesco, D.: Translating Alloy Specifications to UML Class Diagrams Annotated with OCL. In: Barthe, G., Pardo, A., Schneider, G. (eds.) SEFM 2011. LNCS, vol. 7041, pp. 221–236. Springer, Heidelberg (2011)
9. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based specification environment for validating UML and OCL. Sci. Comput. Program. 69(1-3), 27–34 (2007)
10. Jackson, D.: Software Abstractions - Logic, Language, and Analysis. MIT Press (2006)
11. Krieger, M.P., Brucker, A.D.: Extending OCL Operation Contracts with Objective Functions. ECEASST 44 (2011)
12. Krieger, M.P., Knapp, A.: Executing Underspecified OCL Operation Contracts with a SAT Solver. ECEASST 15 (2008)
13. Kuhlmann, M., Gogolla, M.: Intrinsic Relational Approach: Transformation of OCL Operations, http://www.db.informatik.uni-bremen.de/publications/intern/IntrinsicApproachOCL2012.pdf
14. Kuhlmann, M., Gogolla, M.: Strengthening SAT-Based Validation of UML/OCL Models by Representing Collections as Relations. In: Tolvanen, J.P., Vallecillo, A. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 32–48. Springer, Heidelberg (2012)
15. Kuhlmann, M., Hamann, L., Gogolla, M.: Extensive Validation of OCL Models by Integrating SAT Solving into USE. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 290–306. Springer, Heidelberg (2011)
16. Kuhlmann, M., Hamann, L., Gogolla, M., Büttner, F.: A benchmark for OCL engine accuracy, determinateness, and efficiency. Software and System Modeling 11(2), 165–182 (2012)
17. Kuhlmann, M., Sohr, K., Gogolla, M.: Comprehensive Two-Level Analysis of Static and Dynamic RBAC Constraints with UML and OCL. In: Baik, J., Massacci, F., Zulkernine, M. (eds.) SSIRI 2011, pp. 108–117. IEEE (2011)
18. Maoz, S., Ringert, J.O., Rumpe, B.: CD2Alloy: Class Diagrams Analysis Using Alloy Revisited. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 592–607. Springer, Heidelberg (2011)

19. Maoz, S., Ringert, J.O., Rumpe, B.: Semantically Configurable Consistency Analysis for Class and Object Diagrams. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 153–167. Springer, Heidelberg (2011)
20. Olivé, A.: Conceptual Modeling of Information Systems. Springer (2007)
21. Ornaghi, M., Fiorentini, C., Momigliano, A., Pagano, F.: Applying ASP to UML Model Validation. In: Erdem, E., Lin, F., Schaub, T. (eds.) LPNMR 2009. LNCS, vol. 5753, pp. 457–463. Springer, Heidelberg (2009)
22. Queralt, A., Teniente, E.: Reasoning on UML Class Diagrams with OCL Constraints. In: Embley, D.W., Olivé, A., Ram, S. (eds.) ER 2006. LNCS, vol. 4215, pp. 497–512. Springer, Heidelberg (2006)
23. Roldán, M., Durán, F.: Dynamic Validation of OCL Constraints with mOdCL. ECEASST 44 (2011)
24. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual, 2nd edn. The Pearson Higher Education (2004)
25. Samimi, H., Aung, E.D., Millstein, T.D.: Falling Back on Executable Specifications. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 552–576. Springer, Heidelberg (2010)
26. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL models using Boolean satisfiability. In: DATE, pp. 1341–1344. IEEE (2010)
27. Straeten, R.V.D., Puissant, J.P., Mens, T.: Assessing the Kodkod Model Finder for Resolving Model Inconsistencies. In: France, R.B., Küster, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 69–84. Springer, Heidelberg (2011)
28. Torlak, E., Jackson, D.: Kodkod: A Relational Model Finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
29. Torlak, E., Vaziri, M., Dolby, J.: MemSAT: checking axiomatic specifications of memory models. In: Zorn, B.G., Aiken, A. (eds.) PLDI, pp. 341–350. ACM (2010)
30. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. The Addison-Wesley Object Technology Series. Addison-Wesley (2003)

On Verifying ATL Transformations Using 'off-the-shelf' SMT Solvers

Fabian Büttner[1], Marina Egea[2], and Jordi Cabot[1]

[1] AtlanMod Research Group, INRIA / Ecole des Mines de Nantes
{fabian.buettner,jordi.cabot}@inria.fr
[2] Atos, Madrid
marina.egea@atosresearch.eu

Abstract. MDE is a software development process where models constitute pivotal elements of the software to be built. If models are well-specified, transformations can be employed for various purposes, e.g., to produce final code. However, transformations are only meaningful when they are 'correct': they must produce valid models from valid input models. A valid model has conformance to its meta-model and fulfils its constraints, usually written in OCL. In this paper, we propose a novel methodology to perform automatic, unbounded verification of ATL transformations. Its main component is a novel first-order semantics for ATL transformations, based on the interpretation of the corresponding rules and their execution semantics as first-order predicates. Although, our semantics is not complete, it does cover a significant subset of the ATL language. Using this semantics, transformation correctness can be automatically verified with respect to non-trivial OCL pre- and postconditions by using SMT solvers, e.g. Z3 and Yices.

1 Introduction

In Model-Driven Engineering (MDE), models constitute pivotal elements of the software to be built. When they are sufficiently well specified, model transformations can be employed for different purposes, e.g., to produce actual code. However, it is essential that such transformations are *correct* if they are to play their key role. Otherwise, errors introduced by transformations will be propagated and may produce more errors in the subsequent MDE steps. Thus, well-founded and, at the same time, practical verification methods and tools are important to guarantee this correctness.

Our work focuses on checking partial correctness of *declarative, rule-based transformations* between *constrained metamodels*. More specifically, we regard the ATL transformation language [17] and MOF [22] style metamodels that employ OCL constraints to precisely describe their domain. These ingredients are very popular due to their sophisticated tool support, and because OCL is employed in almost all OMG specifications. Several notions of correctness apply to such model transformations, like termination or confluence (see, e.g., [19,14]). In this paper, we are interested in a Hoare-style notion of partial correctness, i.e., in the correctness of a transformation with respect to certain sets of pre- and postconditions. In other words, we are interested in whether the output model produced by an ATL transformation for any valid input model is valid, too. A valid model is one that has conformance to its metamodel and fulfils its constraints, usually written in OCL [21], that is, the OMG standard constraint language for models. We present a novel approach that targets *automatic* and *unbounded* verification of this property for ATL transformations. Our aim is to provide a 'push button' technology that can be applied regularly in model transformation development by developers lacking of a formal background.

The key components of our approach are a novel first order semantics for a declarative subset of ATL, based on the interpretation of ATL rules as first-order functions and predicates, and the use of automatic decision procedures for Satisfiability Modulo Theories problems in SMT solvers. Such solvers, e.g. Z3 [12,28] and Yices [13,27], have been significantly improved in the past years and can automatically and efficiently decide several important fragments of first-order logic (FOL). Our approach combines the advantages of formal verification (in the sense that we aim to provide formal proofs) and automatic verification (in the sense that we do not require the transformation developer to operate, for example, interactive theorem provers). To our knowledge, we are the first ones in proposing a first order semantics for a declarative subset of ATL which, in particular, allows the automatic unbounded verification of transformations between metamodels that may be constrained using OCL (more precisely, the subset of OCL that is considered in [11]).

In addition to the running example that we use in this paper, we have made also available at [8], for the interested reader, the formalization, according to our FOL semantics, of a larger and more complex example. In this example we illustrate how we deal with type inheritance hierarchies and more complex and overlapping input patterns that have to be resolved by type checking and filtering conditions resolution. Also, we deal with more complex bindings statements in the output patterns for this transformation. This example was borrowed from [5].

Organization. In Sect. 2 we present our running example. Sect. 3 and Sect. 4 describe our FOL formalization for metamodels and ATL rules. In Sect. 5 we formalize our Hoare-style notion of partial correctness and how it can be checked using SMT solvers. We discuss related work in Sect. 6, and we conclude and outline future work in Sect. 7.

2 Running Example

Figure 1 depicts the ER and REL metamodels that are (resp.) the source and target metamodels for the ER2REL transformation, which is depicted in Fig. 2. In the ER metamodel, a schema may have entities and relationships (relships), both may contain attributes, and attributes may be keys; in the REL metamodel, a schema may have relations, which may have again attributes.[1] We only provide here an informal description of ER2REL, its precise semantics is discussed later in Sect. 4. Additional information on ATL can be found at [17,3]. In a nutshell, the ER2REL transformation takes an instance of the ER metamodel as input and produces an instance of the REL metamodel following the transformation in Fig. 2. This transformation is described by *matched rules*, which are

[1] For simplicity, we refer to the metamodel elements as schemas, entities, relationships, etc., instead of using schema type, entity type, etc..

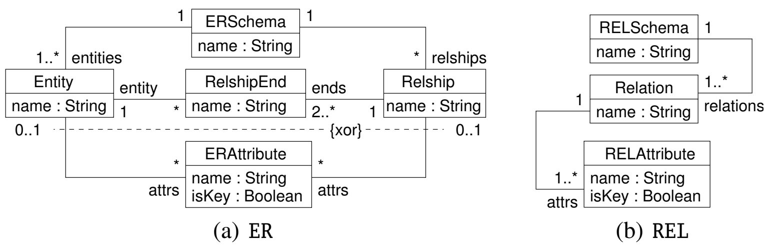


Fig. 1. ER and REL metamodels

```
module ER2REL; create OUT : REL from IN : ER;

rule S2S { from s : ER!ERSchema to t : REL!RELSchema (name <- s.name) }

rule E2R { from s : ER!Entity
           to t : REL!Relation (name<-s.name, schema<-s.schema) }

rule R2R { from s : ER!Relship
           to   t : REL!Relation (name <-s.name, schema<-s.schema) }

rule EA2A { from att : ER!ERAttribute, ent : ER!Entity (att.entity=ent)
            to   t   : REL!RELAttribute
                       (name<-att.name, isKey<-att.isKey, relation<-ent ) }

rule RA2A { from att : ER!ERAttribute, rs  : ER!Relship (att.relship=rs)
            to   t   : REL!RELAttribute
                       (name<-att.name, isKey<-att.isKey, relation<-rs) }

rule RA2AK { from att : ER!ERAttribute, rse : ER!RelshipEnd
                        (att.entity=rse.entity and att.isKey=true)
             to   t   : REL!RELAttribute
                        (name<-att.name, isKey<-att.isKey, relation<-rse.relship)}
```

Fig. 2. The ATL transformation ER2REL

the workhorse of ATL. Matched rules define a pattern of input types and possibly a filter expression (the `from`-clause). Each rule is applied to each matching set of objects in the input model to create the objects in the target model that are described in the `to`-clause, assigning values to their properties (typically) based on the input objects' properties.

The first rule in Fig. 2, S2S, maps ER schemas to REL schemas, the second rule E2R maps each entity to a relation, and the third rule R2R maps each relationship to a relation. The remaining three rules generate attributes for the relations. Both, entity and relationship attributes are mapped to relation attributes (rules EA2A and RA2A). Furthermore, the key attributes of the participating entities are mapped to relation attributes as well (rule RA2AK). Notice that in the property assignment, a so-called *implicit resolution* step is needed to resolve source objects to target objects: For example the binding `schema<-s.schema` in E2R and R2R 'silently' replaces the ERschema value of `s.schema` by the RELSchema object that is created for `s.schema` by S2S. Fig. 3 shows a list with the OCL constraints for the source and target metamodels. The constraints require the expected uniqueness of names within their scopes (e.g., for the entities in a schema), and the existence of key attributes in entities and relations. In addition to the constraints in Fig. 3, multiplicity constraints are encoded as OCL constraints for both metamodels, too. For every lower bound different from 0 and each upper bound different from * we add one OCL constraint named ⟨qualified rolename⟩.lo (for lower) resp. ⟨qualified rolename⟩.up (for upper). E.g., the constraint `Entity::schema.lo` is `context Entity inv: self.schema->size()>=1`, and the constraint for the upper bound is `Entity::schema.up` is `context Entity inv: self.schema->size()<=1`.

```
context ERSchema inv pre1:        -- unique schema names
ERSchema.allInstances()->forall(s1,s2| s1<>s2 implies s1.name<>s2.name)

context ERSchema inv pre2:        -- relship names are unique in schema
self.relships->forAll(r1,r2 | r1<>r2  implies  r1.name<>r2.name)

context ERSchema inv pre3:        -- entity names are unique in schema
self.entities->forAll(e1,e2 | e1<>e2 implies e1.name<>e2.name)

context ERSchema inv pre4:        -- disjoint entity and relship names
self.relships->forAll(r | self.entities->forAll(e |  r.name<>e.name))

context EREntity inv pre5:        -- attr names are unique in entity
self.attrs->forAll(a1,a2 | a1.name=a2.name implies a1=a2)

context ERRelship inv pre6:       -- attr names are unique in relship
self.attrs->forAll(a1,a2 | a1.name = a2.name implies a1=a2)

context Entity inv pre7:          -- entities have a key
self.attrs->exists(a | a.isKey)
-- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
context RELSchema inv post1:      -- unique schema names
RELSchema.allInstances()->forall(s1,s2| s1<>s2 implies s1.name<>s2.name)

context RELSchema inv post2:      -- relation names are unique in schema
relations->forall(r1,r2| r1<>r2 implies r1.name<>r2.name)

context Relation inv post3:       -- attribute names  unique in relation
self.attrs->forAll(a1,a2 | a1.name=a2.name implies a1=a2)

context Relation inv post4:        -- relations have a key
self.attrs->exists(a | a.isKey)
```

Fig. 3. OCL constraints for ER and REL

Problem Statement. A developer who is designing a model transformation typically wonders the following question several times during the designing process: *Do the constraints imposed on the source model plus the transformation specification guarantee that these other constraints are fulfilled by the target models?* When the answer to this question is *‘yes’* for certain properties, we would say that the transformation which is being designed is correct with respect to the given sets of pre and postconditions. Namely, in our view, a model transformation is *correct* if and only if executing it using a constrained-valid input model as argument always results on a constrained-valid output model, where a constrained-valid input model is a model that satisfies the model transformation’s *preconditions* and a constrained-valid output model is a model that satisfies the model transformation’s *postconditions*. Notice that the ATL model transformations that we consider here are always executed to populate target models which are initially empty. For our approach to be practical, we are implementing a tool that automatically maps ATL matched rules to first order logic files and employs the Z3 solver to check whether the implications between pre and postconditions hold. A tool that automatically maps the OCL constrained metamodels to FOL is already implemented.

We can read each row in Table 2 (left hand side) as follows: For each input model which satisfies the named preconditions, the respective postcondition will hold for the output model. These implications were proven automatically to hold for the ER2REL transformation by Z3 and Yices.[2] Moreover, the SMT solvers can also determine the minimal set of preconditions that are required to prove a given postcondition. The table shows that every target postcondition of the REL metamodel can be inferred except post3 – for which Z3 can find a counter example even if all preconditions are assumed, e.g. an input model containing reflexive relationships on the source side. In the following sections, we will explain our FOL semantics for ATL matched rules that allow these implications to be automatically proven by the SMT solvers.

Table 1. Implications that hold for ER2REL (and that can be proven automatically using Z3 and Yices* using our translation into first-order logic). (QI=Quantifier Instantiations). (Class names abbreviations: E = Entity, RS = Relship, RSE = RelshipEnd, RE= Relation, RA = RELAttribute).

Proofs found automatically by Z3 and Yices*					
Preconditions	Postcondition	Unsat core total = 69	QI (C)	QI (U)	QI (P)
pre1	post1	4	22	18	22
E::schema.lo, RS::schema.lo	R::schema.lo	9	186	30	60
E::schema.up, RS::schema.up	R::schema.up	9	310	118	200
pre2, pre3, pre4, E::schema.lo+up, RS::schema.lo+up	post2	16	10274	888	423
RSE:relship.lo	RA::relation.lo	11	359	50	105
RSE::relship.lo+up	RA::relation.up	11	2864	72	247
pre4, RSE::type.lo, RS::ends.lo	post4	14	493	141	235

3 Mapping Metamodels and OCL Constraints to First Order Logic

Since ATL transformations are always defined from a source to a target metamodel, we will first explain how we map metamodels' elements to first order logic. We had already used this first order formalization in [11].

– *Type-predicates:* Metamodels' classes are mapped to unary boolean functions. E.g., the class ERSchema is mapped to a unary boolean function ERSchema: $int \rightarrow bool$;
– *Objects variables* are mapped to integer variables, e.g, an object variable cl of type Entity is mapped to an integer variable cl, such that Entity(cl) holds;
– *Attribute-functions:* Attributes are mapped to either boolean or integer functions, e.g., the attribute name is mapped to a function name: $int \rightarrow int$;[3]

[2] We put a '$*$' to Yices in the table since it requires to assume some previously proven postconditions as lemmas in order to find a proof for R::schema.lo and RA::relation.lo

[3] We do not considered attributes' values of type object or collection. Also, strings are currently treated as Integers. Thus, no string-specific operations are supported. The reason is that there are no such theories and decision procedures available yet for SMT solvers (although this is ongoing work).

– *Association-predicates:* Association-ends of a given association, e.g. `erschema` are mapped to binary boolean functions, e.g., erschema: $int\ int \rightarrow bool$.

Notice that we assume that a function $\lceil . \rceil$ exists to generates unique function symbols for metamodel elements, and that metamodel elements are uniquely named (without losing generality). Next, we outline how we can ensure disjointness of types in the type system, and how to consider inheritance relationships, that were not covered before in our approach. Our extension includes formulas

– to ensure that those type-predicates mapping classes that are not subclasses in any inheritance relationship are pairwise disjointly interpreted. I.e., if c and c' are disjoint classes, we include a formula $\forall(x)\ \neg(\lceil c \rceil(x) \wedge \lceil c' \rceil(x))$ to ensure that their corresponding predicates are disjointly interpreted;
– to map class inheritance relationships. Namely, for each direct subclass relations between c' and c, i.e., c' is a (direct) subclass of c, we include a formula $\forall(x)(\lceil c' \rceil(x) \Rightarrow \lceil c \rceil(x))$. For each abstract superclass c, the following formula would also be included: $\forall(x)(\lceil c \rceil(x) \Rightarrow \bigvee_{1 \leq h \leq k} \lceil c_h \rceil(x))$ for all c_h subclass of c.

Notice that we need to add these axioms explicitly because we do not use sorted logic as, e.g., [25]. In the case of inheritance, only for the immediate subtypes below a superclass in different branches of the tree, the formulas to guarantee the pairwise disjoint interpretation of these types are included.

Remark 1. Neither superclass attributes nor associations ends are specified explicitly for the subclasses, but they are mapped and used according to how they are specified in the metamodel for the superclass. We do not consider multiple inheritance relationships.

From OCL to First-Order Logic. As we mentioned before, this work focuses on the correctness of transformations defined between OCL constrained metamodels. We use our previous work [11] to translate OCL constraints into FOL. The operators that are listed in the examples below are those covered in [11]. [4] Our mapping is both simple, in the sense that the resulting FOL formulas closely mirror the original OCL constraints; and practical, in the sense that, using this mapping we can also employ automated theorem provers and/or SMT solvers (e.g., Z3 and Yices) to automatically perform unsatisfiability checks on non-trivial sets of OCL constraints. In a nutshell, our mapping is defined recursively over the structure of OCL expressions. Attributes, classes and association ends that may be part of OCL expressions are mapped as we explained for metamodel elements.

– `Boolean`-expressions are translated to formulas, which essentially mirror the logical structure of the OCL expressions, e.g., for the operations `or`, `and`, `implies`, `not`, `isEmpty()`, `notEmpty()`, `includes`, `excludes`, $<$, $>$, $\leq$, $\geq$, $=$, $\neq$;
– `Integer`-expressions are basically copied, e.g. $+$, $-$, $*$; currently, we do not cover `String`-expressions.

[4] Our mapping is not yet complete but it does cover a sufficiently significant subset of the OCL language.

- `Collection`-expressions are translated to *fresh* predicates that augment the specification signature and whose meaning is defined by additional formulas generated by the mapping. E.g. to map `select`, `reject`, `collect`, `forAll`, `exists`, `including` or `excluding` operations.

Example 1. Mapping precondition 1 using our previous translation from OCL to FOL. Precondition 1 in the example presented in Sect. 2 is:

```
inv pre1:
ERSchema.allInstances()->forall(x,y| x<>y implies x.name<>y.name)
```

It is mapped to:

$$\forall(x)(\mathsf{ERSchema}(x) \Rightarrow \forall(y)(\mathsf{ERSchema}(y) \Rightarrow ((x \neq y) \Rightarrow (\mathsf{name}(x) \neq \mathsf{name}(y)))))$$

4 First-Order Semantics for ATL Transformations

There are two things that need to be considered in order to understand the meaning of a model-to-model transformation and how it works. One is the language in which it is specified, the other is how the transformation definition is executed by a transformation engine. Therefore, to be able to reason about pre- and postconditions that may hold for an ATL transformation, our first order interpretation of ATL transformations is capturing in addition to the definition of the rules, how the ATL engine executes them. For the work presented here, we assume that the ATL transformation parses and type checks correctly regarding the source and target metamodels, and that its execution does not end in abortion or error, i.e., a valid output model is produced from any valid input model. Currently, we only regard matched rules, in a slightly restricted form that allows only one output pattern element per rule and three kinds of bindings, as it is captured in Fig. 4. But our mapping can be extended to deal with more than one output pattern elements and to cover OCL collection expressions that can be used on the right-hand side of binding statements. Lazy rules will be included (with some restrictions), in future work. Matched rules' patterns (up to name uniqueness) always compose ATL transformations that are terminating and confluent [18]. Furthermore, we only support the subset of OCL that is supported in [11]. In particular, we do not support recursively defined OCL operations that would be the only source of non-termination. Last, we do not allow that both ends of an association are used as target of bindings at the same time, because ATL does not guarantee confluence in this case. The structure of ATL matched rules was briefly explained in Sect. 2. Fig. 4 shows the pattern of matched rules that our mapping currently supports.

The object variables and the OCL expression appearing in the `from`-clause is called the rule's source pattern. The object variable and the binding statements appearing in the `to`-clause are called the rule's target pattern. Recall that the $oclexp$ in the source pattern is a boolean filtering condition (if there is not filtering condition, it is assumed to be $true$). The expressions $bindstm_i$ are binding statements and s_j and o are object variables of types t_j (of the source metamodel) and t' (of the target metamodel), resp.. The properties $attname'_j$, $assocend'_k$ or $assocend'_l$ are (resp.) attribute's names and association ends that belong to t' objects according to the target metamodel definition. Analogously, $attname_{r_k}$ and $assocend_{f_p}$ are attribute's names and association ends that belong to t_r and t_f objects (resp.) according to the source metamodel definition. Moreover, attributes must be of integer or boolean types and their type must conform when they are bound by a rule, e.g. $attname'_j \leftarrow s_r.attname_{r_k}$. We assume that the function $\lceil . \rceil$ introduced in Sect. 3 also produces unique function symbols for ATL rules. These functions are declared with the arity that corresponds to the rules they mirror. Next we give a closer look on how the ATL engine executes these rules. We take advantage of this description to explain how the properties of the execution semantics of ATL rules are captured by our formalization in first order logic. As expected the ATL engine interprets ATL rules oriented from source to target.

```
rule $rlname$
   from $s_1 : t_1, \dots, s_n : t_n$ $(oclexp)$
   to $o : t'$ $(bindstm_1, \dots, bindstm_m)$
where each $bindstm_i$ can have one of the following shapes:
   shape I: $attname'_j \leftarrow s_r.attname_{r_k}$,
   shape II: $assocend'_l \leftarrow s_f.assocend_{f_p}$
   shape III: $assocend'_k \leftarrow s_v$
```

Fig. 4. ATL matched rule's pattern currently supported by our mapping

1. Objects in the target metamodel exists only if they have been created by an ATL rule since the ATL transformations that we consider are always initially executed on an empty target model. Namely, the ATL transformation considered as an operation from a source to a target model is surjective on rules' target object variables' types. When an object type can be generated by the execution of more than one rule of an ATL transformation, then a disjunction considering all these possibilities is made in the consequent of the assertion. E.g., if an object o of type t' can be created by any of the rules $rlname_1$ of input parameters $s_{11} : t_{11}, \dots, s_{1v_1} : t_{1v_1}$ and filtering expression $oclexp_1$, $rlname_2$ of input parameters $s_{21} : t_{21}, \dots, s_{2v_2} : t_{2v_2}$ and filtering expression $oclexp_2$, ..., and $rlname_k$ of input parameters $s_{k1} : t_{k1}, \dots, s_{kv_k} : t_{kv_k}$ and filtering expression $oclexp_k$, formulas shaped as shown in Fig. 5, pattern (i), are generated. This type of formulas is inserted for each target object type of rules in the transformation.[5] Corresponding formulas instantiated for the ER2REL example presented in Sect. 2 are shown in Figs. 7, 9, 10, assertions (i).

2. A rule's source pattern is matched (taking into account the filtering condition) against the elements of the source model (see assertion pattern (ii) in Fig. 6). Elements in the target model are created by the execution of at most one rule using a tuple of input objects that cannot be matched by two different rules (see assertion pattern (iii) in Fig. 6). Namely, an ATL transformation from source to target is executed as a function and, in addition, it is globally injective. To ensure that a target object can only be produced

[5] The function ocl2fol represents the mapping from OCL to first order logic defined in [11] and described in Sect. 3. It will produce a conjunction of boolean formulas when applied to an OCL boolean expression.

$$\begin{aligned}
&(i)\ \forall(o)\ (\lceil t' \rceil(o)) \Rightarrow \exists(s_{11},\ldots,s_{1v_1})\ (\lceil t_{11}\rceil(s_{11}) \wedge \ldots \wedge \lceil t_{1v_1}\rceil(s_{1v_1})\ \wedge \mathsf{ocl2fol}(oclexp_1) \wedge \\
&\quad (\lceil rlname_1\rceil(s_{11},\ldots,s_{1v_1}) = o)) \vee \\
&\quad \exists(s_{21},\ldots,s_{2v_2})\ (\lceil t_{21}\rceil(s_{21}) \wedge \ldots \wedge \lceil t_{2v_2}\rceil(s_{2v_2})\ \wedge \mathsf{ocl2fol}(oclexp_2) \wedge \\
&\qquad (\lceil rlname_2\rceil(s_{21},\ldots,s_{2v_2}) = o)) \vee \\
&\qquad \ldots\ldots\ldots\ldots \\
&\quad \exists(s_{k1},\ldots,s_{kv_k})\ (\lceil t_{k1}\rceil(s_{k1}) \wedge \ldots \wedge \lceil t_{kv_k}\rceil(s_{kv_k})\ \wedge \mathsf{ocl2fol}(oclexp_k) \wedge \\
&\qquad (\lceil rlname_k\rceil(s_{k1},\ldots,s_{kv_k}) = o))
\end{aligned}$$

Fig. 5. Formulas to capture that ATL transformations are surjective on target object variables' types

by one rule on a fixed tuple of arguments, we introduce a function creation. It assigns to each target object the rule identifier (which is a constant defined exactly for this purpose) and the input object pattern that created it. In order to have an homogeneous signature of this function, we assume the maximum input pattern arity u (plus 1 to insert the rule identifier). For rules with a lesser arity $v < u$, the tuple is completed with arbitrary object variables that appear existentially quantified. This definition, shown in Fig. 6, represents a simple way for ensuring global injectivity for the transformation. These type of formulas are also inserted for every ATL rule in the transformation. These formulas instantiated for the ER2REL example presented in Sect. 2 are shown in Figs. 7, 9, 10, assertions (ii)-(iii).

$$\begin{aligned}
&(ii)\ \forall(s_1,\ldots,s_n)(\lceil t_1\rceil(s_1) \wedge \ldots \wedge \lceil t_n\rceil(s_n) \wedge \mathsf{ocl2fol}(oclexp)) \Rightarrow \\
&\quad \exists(o)\ (\lceil t'\rceil(o) \wedge (\lceil rlname\rceil(s_1,\ldots,s_n) = o)) \\
&(iii)\ \forall(s_1,\ldots,s_n,o)(\lceil t_1\rceil(s_1) \wedge \ldots \wedge \lceil t_n\rceil(s_n) \wedge \mathsf{ocl2fol}(oclexp) \wedge \\
&\quad (\lceil rlname\rceil(s_1,\ldots,s_n) = o)) \Rightarrow \\
&\quad \exists(y_1,\ldots,y_d)(\lceil creation\rceil(o) = \langle idrlname, s_1,\ldots,s_n,y_1,\ldots,y_d\rangle),\ \text{with } d\text{=}(u\text{-}1)\text{-}n
\end{aligned}$$

Fig. 6. Formulas to capture the rules' source patterns' matching and rule's inyectivity

3. The *bindings* of the target patterns are performed straight-forwardly for attribute's values (binding pattern shape I in Fig. 4) of primitive types. However, an implicit resolution strategy is applied by the ATL engine to resolve source objects to target objects. This mechanism is in place for shapes II and III of the binding statements given in Fig. 4). Recall that the transformations we assume as our subject of study can be successfully executed. In particular, this means that all the bindings declared in the target model are well defined, i.e., can be performed. To mirror the binding mechanism, auxiliar functions are defined. There will be as many of these functions as different rules' arities are in the transformation. Fig. 8 illustrates how they are used, but we do not further explain here how they are defined for space reasons). Let us just say that we represent these functions with the symbol $\mathsf{resolve_u}$, where u is the arity of the rules it is based on.

$$
\begin{aligned}
&(i)\ \forall(s)(\mathsf{RELSchema}(s) \Rightarrow \exists(p)(\mathsf{ERSchema}(p) \wedge (\mathsf{S2S}(p) = s)),\\
&(ii)\ \forall(p)(\mathsf{ERSchema}(p) \Rightarrow \exists(s)\ (\mathsf{RELSchema}(s) \wedge (\mathsf{S2S}(p) = s)),\\
&(iii)\forall(p,s)(\mathrm{ERSchema}(p) \wedge \mathrm{RELSchema}(s) \wedge (\mathrm{S2S}(p) = s)) \Rightarrow\\
&\qquad \exists(y)(\mathrm{creation}(s) = \langle idS2S, p, y\rangle),\\
&(\mathrm{SI})\ \forall(p,s)(\mathsf{ERSchema}(p) \wedge \mathsf{RELSchema}(s) \wedge (\mathsf{S2S}(p) = s)) \Rightarrow\\
&\qquad (\mathsf{name}(s) = \mathsf{name}(p))
\end{aligned}
$$

Fig. 7. Formulas (i), (ii) and (iii) for the rule S2S

(Shape I)

$$
\begin{aligned}
&\forall(s_1,\ldots,s_n,o)(\lceil t_1\rceil(s_1) \wedge \ldots \wedge \lceil t_n\rceil(s_n) \wedge \mathsf{ocl2fol}(oclexp) \wedge (\lceil rlname\rceil(s_1,\ldots,s_n) = o))\\
&\quad \Rightarrow (\lceil attname'_j\rceil(o) = \lceil attname_{r_k}\rceil(s_r))
\end{aligned}
$$

(Shape II)

$$
\begin{aligned}
&\forall(s_1,\ldots,s_n,o)(\lceil t_1\rceil(s_1) \wedge \ldots \wedge \lceil t_n\rceil(s_n) \wedge \mathsf{ocl2fol}(oclexp) \wedge (\lceil rlname\rceil(s_1,\ldots,s_n) = o))\\
&\qquad \Rightarrow \forall(w)(\lceil \mathtt{type}(assocend_{f_p})\rceil(w) \wedge \lceil assocend_{f_p}\rceil(s_f, w)) \Rightarrow\\
&\qquad\quad \exists(w')(\lceil \mathtt{type}(assocend'_l)\rceil(w') \wedge \lceil assocend'_l\rceil(o, w') \wedge resolve_1(w, w'))\\
&\quad\ \forall(w')(\lceil \mathtt{type}(assocend'_l)\rceil(w') \wedge \lceil assocend'_l\rceil(o, w'))\\
&\qquad\quad \Rightarrow \exists(w)(\lceil \mathtt{type}(assocend_{f_p})\rceil(w) \wedge \lceil assocend_{f_p}\rceil(s_f, w) \wedge resolve_1(w, w'))
\end{aligned}
$$

(Shape III)

$$
\begin{aligned}
&\forall(s_1,\ldots,s_n,o)(\lceil t_1\rceil(s_1) \wedge \ldots \wedge \lceil t_n\rceil(s_n) \wedge \mathsf{ocl2fol}(oclexp) \wedge \lceil rlname\rceil(s_1,\ldots,s_n) = o)\\
&\qquad\quad \Rightarrow \exists(w')(\lceil \mathtt{type}(assocend'_k)\rceil(w') \wedge \lceil assocend'_k\rceil(o, w') \wedge resolve_1(s_v, w')) \wedge\\
&\quad\ \forall(w')((\lceil \mathtt{type}(assocend'_k)\rceil(w') \wedge \lceil assocend'_k\rceil(o, w')) \Rightarrow resolve_1(s_v, w')))
\end{aligned}
$$

Fig. 8. Formulas for the bindings of the rules

It is defined as boolean function with $u+1$ arity, and it helps to distinguish which is the rule resolving source to target objects. For the ATL transformation presented in Fig. 2, the function $\mathtt{resolve}_1$ defined as it is shown in Fig. 11 is used to resolve the binding patterns `erschema`←`er.relschema` of the rules E2R and R2R, and the binding patterns `relation`←`ent` of the rule EA2A, `relation`←`rs` and `relation`←`rse.relship` of the rules RA2A and RA2AK (resp.).

Formulas mapping binding statements are inserted *on-demand* for every ATL rule in the transformation. We formalized the mapping of binding statements of shape I-III in Fig. 8. Although we do not provide in this paper the definition of the function `type` over the metamodel (and it is used in the definition shown in Fig. 8), notice that it simply returns the type of the association end it is applied to. These formulas instantiated for the ER2REL example presented in Sect. 2 are shown in Fig. 7, 9, 10, assertions headed by (SI) to (SIII) for (Shape I) to (Shape III).

$$
\begin{aligned}
&(i)\ \forall(t)(\mathsf{Relation}(t) \Rightarrow \\
&\quad \exists(e)(\mathsf{Entity}(e) \wedge (\mathsf{E2R}(e) = t))\ \vee\ \exists(rh)(\mathsf{Relship}(rh) \wedge (\mathsf{R2R}(rh) = t)), \\
&(ii)\ \forall(e)(\mathsf{Entity}(e) \Rightarrow \exists(t)\ (\mathsf{Relation}(t) \wedge (\mathsf{E2R}(e) = t)), \\
&(iii)\ \forall(t,e)(\mathsf{Entity}(e) \wedge \mathsf{Relation}(t) \wedge (\mathsf{E2R}(e) = t)) \Rightarrow \\
&\qquad \exists(y)(\mathrm{creation}(t) = \langle idE2R, e, y\rangle), \\
&(\mathrm{SI})\ \forall(e,t)(\mathsf{Entity}(e) \wedge \mathsf{Relation}(t) \wedge (\mathsf{E2R}(e) = t)) \Rightarrow \\
&\qquad (\mathsf{name}(e) = \mathsf{name}(t)), \\
&(\mathrm{SII})\ \forall(e,t)\ (\mathsf{Entity}(e) \wedge \mathsf{Relation}(t) \wedge (\mathsf{E2R}(e) = t)) \Rightarrow \\
&\qquad (\forall(p)\ (\mathsf{ERSchema}(p) \wedge \mathsf{erschema}(e,p)) \Rightarrow \\
&\qquad\quad \exists(s)\ (\mathsf{RELSchema}(s) \wedge \mathsf{relschema}(t,s) \wedge \mathsf{resolve}_1(p,s))) \wedge \\
&\qquad (\forall(s)\ (\mathsf{RELSchema}(s) \wedge \mathsf{relschema}(t,s)) \Rightarrow \\
&\qquad\quad \exists(p)\ (\mathsf{ERSchema}(p) \wedge \mathsf{erschema}(e,p) \wedge \mathsf{resolve}_1(p,s)))
\end{aligned}
$$

Fig. 9. Formulas (i), (ii) and (iii) for the rule `E2R`. Map of its binding statements of shape I-II.

5 Verifying Model Transformations

In this section we formalize the Hoare-style notion of correctness (i.e. Hoare triples) that we use to verify ATL model transformations. [6] In particular, Def. 1 follows standard Hoare logic in that it deals only with partial correctness, while termination would need to be proven separately. Notice however that the matched rules' patterns considered in this work (up to name uniqueness) always compose ATL transformations that are terminating and confluent since this type of rules do not contain any possible source of non-termination. Namely, they do not contain recursive calls, neither recursively defined OCL helper operations. This claim is further supported and explained in [18]. In addition, notice that for the ATL rules that we consider in this work, only two conditions can get an execution aborted: (a) Two rules match the same tuple of objects; (b) A binding of shape III (Fig. 4) cannot be resolved because the required object was not matched by any rule's source pattern. Neither (a) nor (b) happen in our examples.

Assuming that *ocl2fol* represents our mapping from OCL to first-order logic [11] described in Sect. 3, and that *atl2fol* is the mapping from ATL to first-order logic that we partially described in Sect. 4, we are able to formalize our notion of ATL model transformations correctness in Def. 1 in two alternative shapes. The former definition of correctness is usually more convenient for using theorem provers to prove postconditions and the latter definition of correctness allows us to reduce the problem of checking the correctness of an ATL model transformation to the problem of checking the unsatisfiability of a set of first-order sentences, which can be checked using an SMT solver. In fact, all correctness checks shown in Table 2 were automatically proven by Z3 and Yices, two modern SMT solvers. However, let us remark again that Yices required some postconditions previously proven as lemmas to find a proof for postconditions

[6] Let us recall, informally, that these triples, i.e. $\{\Phi\}\ \mathcal{Q}\ \{\Psi\}$, with $\{\Phi\}$ and $\{\Psi\}$ being formulas in first order logic, mean that if $\{\Phi\}$ holds before the execution of $\mathcal{Q}$ and, if $\mathcal{Q}$ terminates, then $\{\Psi\}$ will hold upon termination.

$$\begin{aligned}
&(i)\ \forall(t)\ (\mathsf{RELAttribute}(t) \Rightarrow \\
&\quad \exists(at,e)\ (\mathsf{ERAttribute}(at) \wedge \mathsf{Entity}(e) \wedge \mathsf{entity}(at,e) \wedge (\mathsf{EA2A}(e,at)=t)) \vee \\
&\quad \exists(at,rh)\ (\mathsf{ERAttribute}(at) \wedge \mathsf{Relship}(rh) \wedge \mathsf{relship}(at,rh) \wedge (\mathsf{RA2A}(rh,at)=t)) \vee \\
&\quad \exists(at,rhe,e)\ (\mathsf{ERAttribute}(at) \wedge \mathsf{RelshipEnd}(rhe) \wedge \mathsf{Entity}(e) \wedge \mathsf{entity}(at,e) \wedge \\
&\quad \mathsf{type}(rhe,e) \wedge (\mathsf{isKey}(at)=true) \wedge (\mathsf{RA2AK}(rhe,at)=t)), \\
&(ii)\ \forall(e,at)(\mathsf{Entity}(e) \wedge \mathsf{ERAttribute}(at) \wedge \mathsf{entity}(at,e)) \Rightarrow \\
&\quad \exists(t)(\mathsf{RELAttribute}(t) \wedge (\mathsf{EA2A}(e,at)=t)), \\
&(iii)\ \forall(e,at,t)(\mathsf{Entity}(e) \wedge \mathsf{ERAttribute}(at) \wedge \mathsf{RELAttribute}(t) \wedge (\mathsf{EA2A}(e,at)=t)) \Rightarrow \\
&\quad (\mathrm{creation}(t) = \langle idEA2A, e, at\rangle), \\
&(\mathrm{SI})\ \forall(e,at,t)(\mathsf{Entity}(e) \wedge \mathsf{ERAttribute}(at) \wedge \mathsf{RELAttribute}(t) \wedge (\mathsf{EA2A}(e,at)=t)) \Rightarrow \\
&\quad (\mathsf{name}(at) = \mathsf{name}(t)), \\
&(\mathrm{SIII})\ \forall(e,at,t)(\mathsf{Entity}(e) \wedge \mathsf{ERAttribute}(at) \wedge \mathsf{RELAttribute}(t) \wedge (\mathsf{EA2A}(e,at)=t)) \Rightarrow \\
&\quad \exists(w')\ (\mathsf{Relation}(w') \wedge \mathsf{relation}(t,w') \wedge \mathsf{resolve}_1(e,w'))) \wedge \\
&\quad \forall(w')\ (\mathsf{Relation}(w') \wedge \mathsf{relation}(t,w') \Rightarrow \mathsf{resolve}_1(e,w')))
\end{aligned}$$

Fig. 10. Formulas (i), (ii) and (iii) for the rule `EA2A`, taking into account that `RELAttributes` can also be created by the rules `RA2A` and `RA2AK`. Map of its binding statements of shape I and III

$$\begin{aligned}
&\mathsf{resolve}_1(x,y) =^{def.} ((\mathsf{ERSchema}(x) \wedge \mathsf{RELSchema}(y) \wedge (\mathsf{S2S}(x)=y)) \vee \\
&\quad (\mathsf{EREntity}(x) \wedge \mathsf{RELRelation}(y) \wedge (\mathsf{E2R}(x)=y)) \vee \\
&\quad (\mathsf{ERRelship}(x) \wedge \mathsf{RELRelation}(y) \wedge (\mathsf{R2R}(x)=y)))
\end{aligned}$$

Fig. 11. Definition of the auxiliar function $\mathtt{resolve}_1$

`R::Schema.lo` and `RA::relation.lo`, whereas the decision procedures of Z3 were able to fully handle these cases without further help (we further discuss this generality aspect below).

Definition 1. *Let $\mathcal{Q} = \{r_1, \ldots, r_n\}$ be an ATL model transformation composed of matched rules (and free from OCL recursive helper operators). Then, $\mathcal{Q}$ is correct with respect to preconditions $\{\varsigma_1 \ldots \varsigma_l\}$ and postconditions $\{\tau_1, \ldots, \tau_w\}$ if and only if, upon termination of $\mathcal{Q}$, for every τ_i, $i = 1, \ldots, w$, the following formula always hold:*

$$\left(\bigwedge_{j=1}^{l} ocl2fol(\varsigma_j)\right) \wedge \left(\bigwedge_{j=1}^{n} atl2fol(r_j)\right) \Rightarrow ocl2fol(\tau_i) \tag{1}$$

or, equivalently, the following formula is unsatisfiable

$$\left(\bigwedge_{j=1}^{l} ocl2fol(\varsigma_j)\right) \wedge \left(\bigwedge_{j=1}^{n} atl2fol(r_j)\right) \wedge \neg(ocl2fol(\tau_i)) \tag{2}$$

The correctness of our approach obviously depends on the correctness of the mappings from OCL to FOL and from ATL to FOL, that is, on whether they correctly capture the semantics of OCL constraints and of ATL rules and rules' execution. These are certainly two challenging theoretical problems, whose solutions will require, first of all, well-defined, commonly accepted, formal semantics for OCL and ATL: none of these are currently available. Still notice that our translation yields very intuitive formulas for anyone familiar with ATL, and so they are suited for validation by humans against the expected behaviour of ATL.

Automatic verification of transformation correctness. For the ER2REL example presented in Sect. 2, all implications summarized in Table 2 were automatically (and directly) proven by Z3 and (partly by) Yices in less than 1 second (in a standard 2.2 Ghz office laptop running Windows 7). Similarly, for the more complex example that we borrowed from [5], all postconditions that we required were proven automatically in less than a minute using Z3. Actually, they were proven in less than 10 seconds when previously proven postconditions (multiplicity constraints in the target metamodel) were used as lemmas. The preconditions that are required to find a proof can be automatically determined by the solvers as the *unsatisfiable cores*. The column '*unsat core*' in Table 2 shows for the ER2REL example the number of assertions (from the FOL specification that contains the ATL semantics and the mapped OCL constraints used as preconditions) that are required to prove a postcondition. The other columns show the number of *quantifier instantiations* required to perform the proof. These numbers are directly correlated to the numbers of ground terms created by Z3 by instantiating the universally quantified variables in our formulas. QI(C) is the number of instantiations when all assertions are enabled, QI(U) is the number of instantiation when we only leave active the assertions that belong to the unsatisfiable core (others are 'disabled'). Finally, QI(P) is the number of instantiations made when all assertions for the transformation semantics are considered together with only the required preconditions. The relation between the three columns shows that the solver benefits from reduced precondition sets. In total, the specification of the ER2REL example in FOL (accounting pre-conditions and ATL semantics assertions) consists of 69 formulas.

Z3 can also work as a counter example finder but, in general, its algorithms seem to be slower for that goal. We performed several experiments to test the efficiency of Z3 for counter example finding. For instance, since we knew that post3 for ER2REL did not hold from the assumed preconditions, we ask Z3 to find a counter example, however, Z3 took more than 2 hours for such task when we did not specify a maximum model extent. Nevertheless, we think that tools specially dedicated to finite model finding such as Alloy are better suited to perform exactly that task in less time. Our experience [9] leads us in this direction also, and we consider a matter of future work tailoring our semantics for these tools. They would provide the required complement to SMT solvers, for the satisfiable case.

Generality of our approach. Both examples (ER2REL and the one borrowed from [5]) can be automatically verified using Z3 and (partly) Yices (the interested reader can find the files containing the formalization of both examples ready to feed Z3 and Yices at [8]). But, as FOL is not decidable we cannot claim full generality for our approach. However, we expect our FOL mapping for ATL matched rules to fall in the scope of what can be solved by the model-based quantifier instantiation (MBQI) decision procedure of Z3 [15] that is refutationally complete for quantified formulas of uninterpreted and 'almost uninterpreted' functions. Yet, which part of the OCL language is decidable needs to be investigated.

6 Related Work

Several works address automated verification of model-to-model transformations for the same Hoare-style notion of partial correctness. Nevertheless, as it happens with any other modelization process, there are several possible translations from a given source language, e.g. from ATL, both to different formalisms and following different strategies depending on the correctness properties that we want to verify and the desired properties of the verification procedure itself (complete, automatic, etc.). Next, we will distinguish two groups of related works: 1) automatic unbounded verification approaches; 2) automatic bounded verification approaches.

In the first group, [16] type checks transformations with respect to a schema (i.e., a metamodel) by using the MONA solver. Only the typing of the graph can be checked in this approach. Other properties, e.g., the uniqueness of names, cannot be expressed in this approach while they can be checked using OCL. In [2], they propose novel deduction rules to be used for automatically deriving properties of model transformations. On the contrary, we do not propose new deduction rules but rely on the deduction systems implemented in the SMT solvers. In [24], unbounded model checking is used to check first-order linear temporal properties for graph transformation systems. Standard OCL does not capture temporal properties (nor does our mapping). In the same vein, [20] map transformations into the DSLTrans language, and pattern-based properties into a model-checking problem (using Prolog). To our knowledge, there is no approach in this group dealing with ATL or with a constraint language similar to OCL.

In the second group, [26], provides a rewriting logic semantics for ATL, and uses Maude to simulate and verify transformations. In the same logic, [7] formalizes QVT-like transformations. [1,4] translate pattern-based model-transformations into the relational logic of Alloy. In [6] they extend a verification technique capable of checking statically that graph based refactoring rule applications preserve consistency with regards to graph constraints by automatically performing counterexample finding. The consistency notion used in [6] is analogous to the partial correctness notion that we use in this paper when applied to 'in-place' transformations. The same notion is used also in [23] for the verification of graph programs. Finally, we have also translated ATL transformations into corresponding transformation models to capture its execution semantics by OCL constraints, and we have used Alloy to find counterexamples [9]. Notice that the generated transformation models, which have a nice intuitive interpretation as a trace model, could be further translated to FOL using [11]. However, the translation obtained is not adequate for SMT solvers since the resulting FOL assertions are overly complex for efficient e-pattern matching (neither Yices nor Z3 could perform any of the proofs in our examples using the resulting specification of this approach). In this sense, our works complement each other, i.e., [9] is well-suited for bounded counter example finding, whereas the approach presented in this paper can provide proofs for the unsatisfiable cases. In [10], we showed how TGG and QVT-R transformations can be translated into OCL-based transformation models, yielding a different kind of models than [9] due to the different execution semantics. We expect that a direct first-order semantics for QVT-R is also required in order to employ SMT solvers for verifying these transformations following the approach presented in this paper.

7 Conclusions and Future Work

We summarize our contributions in this paper as follows: (i) We provide a novel (and the only, so far) first-order semantics for a declarative subset of ATL; (ii) we propose an automatic, unbounded approach to formally verify a Hoare style notion of partial correctness for ATL transformations with respect to OCL pre- and postconditions; (iii) we have successfully used SMT solvers to perform that verification, i.e., to automatically prove constraints that will always hold on target models. Our approach is suited for 'black box' application by non-formal developers, because we do not require interaction with a theorem prover. For our approach to be practical, we are implementing a tool that automatically maps ATL matched rules to first order logic files and employs the Z3 solver to check whether the implications between pre and postconditions hold. A tool that automatically maps to FOL the OCL constrained metamodels is already implemented.

Our work complements those on bounded verification of model transformations (e.g., using SAT-based tools such as Alloy). Whereas bounded approaches are generally more efficient (and complete within the bounds) in finding counterexamples, our approach provides evidence in the cases when no counterexample could be found by the bounded approach. Furthermore, SMT solvers provide the information about which are the assertions producing unsatisfiability, i.e., 'what implies what', since they can extract the unsatisfiable core. This is particularly useful in terms of guaranteeing which preconditions imply which postconditions. For both the example presented in this paper and the larger one provided on-line, all assertions of interest could be proven in less than a few seconds. We expect our FOL mapping for ATL matched rules to fall in the scope of what can be handled by the model-based quantifier instantiation (MBQI) decision procedure of Z3 [15] that is refutationally complete for quantified formulas of uninterpreted and 'almost uninterpreted' functions (presuming that the OCL constraints of the metamodels fall into the same fragment). Yet, even if the generated first-order specification falls into a fragment for which the SMT decision procedure is refutationally complete, termination is not guaranteed for the case when a counterexample exists. Finally, we must say that most of the work in all examples that we considered so far, could be done by the incomplete (but more efficient) standard e-pattern matching procedure. In future work we will also generalize our translation to consider broader rule patterns, in particular, to deal with more than one output pattern elements and to cover OCL collection expressions that can be used on the right-hand side of binding statements. Lazy rules (in a restricted way) will be also considered.

Acknowledgements. This research was partially funded by the EU project NESSoS (FP7 256890) and, by the Nouvelles Eq́uipes Program of the Pays de la Loire Region (France).

Moreover, we would like to thank Salvador Martinez Pérez (AtlanMod Research group-INRIA) for his helpful comments on previous versions of the work presented here.

References

1. Anastasakis, K., Bordbar, B., Küster, J.M.: Analysis of Model Transformations via Alloy. In: Proceedings of MoDeVVa 2007 (2007), http://www.modeva.org/2007/modevva07.pdf
2. Asztalos, M., Lengyel, L., Levendovszky, T.: Towards automated, formal verification of model transformations. In: Proceedings 3rd International Conference on Software Testing, Verification and Validation, ICST 2010, pp. 15–24. IEEE Computer Society (2010)
3. ATL User Guide (2012) http://wiki.eclipse.org/ATL/User_Guide_-_The_ATL_Language
4. Baresi, L., Spoletini, P.: On the Use of Alloy to Analyze Graph Transformation Systems. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 306–320. Springer, Heidelberg (2006)
5. Baudry, B., Ghosh, S., Fleurey, F., France, R.B., Traon, Y.L., Mottu, J.-M.: Barriers to systematic model transformation testing. Communications of ACM 53(6) (2010)
6. Becker, B., Lambers, L., Dyck, J., Birth, S., Giese, H.: Iterative Development of Consistency-Preserving Rule-Based Refactorings. In: Cabot, J., Visser, E. (eds.) ICMT 2011. LNCS, vol. 6707, pp. 123–137. Springer, Heidelberg (2011)
7. Boronat, A., Heckel, R., Meseguer, J.: Rewriting Logic Semantics and Verification of Model Transformations. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 18–33. Springer, Heidelberg (2009)
8. Büttner, F., Egea, M., Cabot, J.: On verifying ATL transformations using 'off-the-shelf' SMT solvers: Examples (2012), http://www.emn.fr/z-info/atlanmod/index.php/MODELS_2012_SMT
9. Büttner, F., Egea, M., Cabot, J., Gogolla, M.: Verification of ATL transformations using transformation models and model finders. In: Proceedings of 14th International Conference on Formal Engineering Methods, ICFEM 2012, Kyoto, Japan, November 12-16. LNCS, Springer (in press, 2012)
10. Cabot, J., Clariso, R., Guerra, E., Lara, J.: Verification and validation of declarative model-to-model transformations through invariants. Journal of Systems and Software 83(2) (2010)
11. Clavel, M., Egea, M., de Dios, M.A.G.: Checking unsatisfiability for OCL constraints. Electronic Communications of the EASST 24 (2009)
12. de Moura, L.M., Bjørner, N.: Satisfiability modulo theories: Introduction and applications. Communications of ACM 54(9), 69–77 (2011)
13. Dutertre, B., Moura, L.D.: The Yices SMT solver. Technical report, Computer Science Laboratory, SRI International (2006), http://yices.csl.sri.com/tool-paper.pdf
14. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. An EATCS Series. Springer (2006)
15. Ge, Y., de Moura, L.M.: Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009)
16. Inaba, K., Hidaka, S., Hu, Z., Kato, H., Nakano, K.: Graph-transformation verification using monadic second-order logic. In: Schneider-Kamp, P., Hanus, M. (eds.) Proceedings of ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, PPDP 2011, pp. 17–28. ACM (2011)

17. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1-2) (2008)
18. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006), http://sosym.dcs.kcl.ac.uk/events/mtip05/submissions/jouault_kurtev_transforming_models_with_atl.pdf
19. Lano, K., Kolahdouz-Rahimi, S.: Model-Driven Development of Model Transformations. In: Cabot, J., Visser, E. (eds.) ICMT 2011. LNCS, vol. 6707, pp. 47–61. Springer, Heidelberg (2011)
20. Lucio, L., Barroca, B., Amaral, V.: A Technique for Automatic Validation of Model Transformations. In: Petriu, D.C., Rouquette, N., Haugen, O. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 136–150. Springer, Heidelberg (2010)
21. OMG. The Object Constraint Language Specification v. 2.2 (Document formal/2010-02-01). Object Management Group, Inc. (2010), http://www.omg.org/spec/OCL/2.2/
22. OMG. Meta Object Facility (MOF) Core Specification 2.4.1 (Document formal/2011-08-07). Object Management Group, Inc. (2011), http://www.omg.org
23. Poskitt, C.M., Plump, D.: Hoare-style verification of graph programs. Fundamenta Informaticae 118(1-2), 135–175 (2012)
24. Rensink, A.: Explicit State Model Checking for Graph Grammars. In: Degano, P., De Nicola, R., Meseguer, J. (eds.) Concurrency, Graphs and Models. LNCS, vol. 5065, pp. 114–132. Springer, Heidelberg (2008)
25. Richters, M., Gogolla, M.: On Formalizing the UML Object Constraint Language OCL. In: Ling, T.-W., Ram, S., Li Lee, M. (eds.) ER 1998. LNCS, vol. 1507, pp. 449–464. Springer, Heidelberg (1998)
26. Troya, J., Vallecillo, A.: A Rewriting Logic Semantics for ATL. Journal of Object Technology 10 (2011)
27. Yices, http://yices.csl.sri.com/
28. Z3, http://research.microsoft.com/en-us/um/redmond/projects/z3/

ATLTest: A White-Box Test Generation Approach for ATL Transformations

Carlos A. González and Jordi Cabot

École des Mines de Nantes - INRIA - LINA, Nantes, (France)
{carlos.gonzalez,jordi.cabot}@mines-nantes.fr

Abstract. MDE is being applied to the development of increasingly complex systems that require larger model transformations. Given that the specification of such transformations is an error-prone task, techniques to guarantee their quality must be provided. Testing is a well-known technique for finding errors in programs. In this sense, adoption of testing techniques in the model transformation domain would be helpful to improve their quality. So far, testing of model transformations has focused on black-box testing techniques. Instead, in this paper we provide a white-box test model generation approach for ATL model transformations.

1 Introduction

Model-Driven Engineering (MDE) is a software engineering paradigm where models play a fundamental role. They are used to specify, simulate, test, verify and generate code for the application to be built. Most of these activities are model manipulations, thus, model transformation becomes a crucial activity. Nevertheless, writing model transformations is a complex and error-prone task, specially when using MDE to develop complex systems that usually involve chains of large transformations. Therefore, having mechanisms to make model transformations more reliable has become a matter of utmost importance.

Among the possible strategies to improve the quality of model transformations, several testing techniques for model transformations have been recently proposed (see [4] for a recent survey). So far, most of the techniques follow a black-box approach (i.e. transformations are regarded as a black-box so the generation of test input models does not take into account the internals of the transformation) while only a few attempt partial white-box testing strategies (but oriented towards the coverage of the input metamodel and not that of the transformation).

In this sense, the contribution of this paper is to define a new white-box testing mechanism to generate test input models out of ATL model transformations. Our goal is to optimize the generation of the tests by maximizing the coverage of the internal transformation structure. We have chosen ATL [14] as target transformation language due to its popularity (both in academia and industry). However, many of the ideas presented herein could be applied to other transformation languages following the rule-based paradigm in which the OCL is broadly used such as the QVT transformation language family. Our approach can be used in isolation or could be integrated with black-box testing techniques to provide an hybrid test transformation framework.

This paper is structured as follows: Section 2 provides some background on model transformation testing and motivates our approach. Section 3 describes at a high level the foundations of our approach and introduces the running example used throughout the paper. Section 4 goes into detail on how to analyze an ATL transformation to extract the information needed to generate the test input models. Section 5 describes the generation of test input models using the information extracted in Section 4. Finally, Section 6 reviews the related work, and some conclusions and further work are drawn in Section 7.

2 Background and Motivation

Software testing, also known as program testing, can be viewed as the destructive process of trying to find the errors (whose presence is assumed) in a program or piece of software, of course, with the intent of establishing some degree of confidence that the program does what it is expected to do [19]. A common methodology to test a piece of software generally comprises a number of well known steps, namely the creation of input test cases, running the software with the test cases, and finally, using an oracle to analyze the results yielded to determine whether errors came up or not. An oracle is any program, process or body of data that specifies the expected outcome for a set of test cases as applied to a tested object [5] and it can be as simple as a manual inspection or as complex as a separate piece of software.

It is generally accepted that the more input tests are created and the more time is spent running the software, the higher is the probability of finding errors and therefore end up with a more reliable software. However, since finding all the errors presented in a piece of software is impossible [5], and the number of test cases that can be created to test a piece of software can be potentially infinite, it is necessary to establish some strategy to carry out testing in an effective way. Two of the most prevalent strategies are black-box testing and white-box testing. The main difference betweeen them is that in black-box testing only the program specification is taken into account at the time of designing test cases, whereas in white-box testing test cases are created out of the analysis of program internals. Mixed strategies combining both approaches are usually encouraged, though, in order the get a better testing experience.

The methodology to test a model transformation is essentially the same as for software testing and, therefore the same conclusions can be applied. However, compared to program testing, model transformation testing must face an additional challenge [3]: the complex nature of model transformation inputs and outputs. Models can be large structures and must conform to a meta-model (possibly extended with OCL well-formedness rules) thus making even harder the generation of test models and the analysis of the results. Fig. 1 shows what

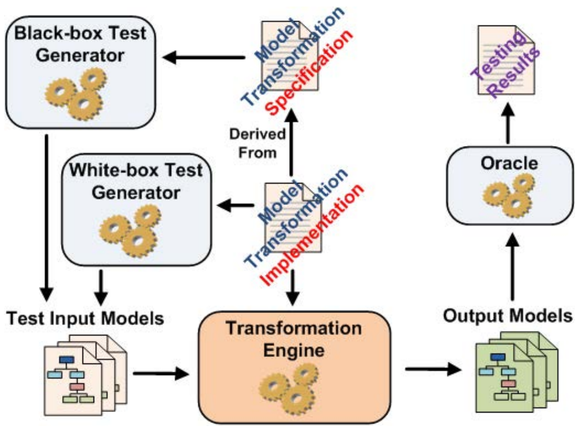


Fig. 1. Mixed approach to model transformation testing

a mixed strategy to test model transformations looks like, in which black-box testing approaches derive test cases from the transformation specification and white-box approaches do it out of its implementation.

So far, the generation of input test models by means of black-box techniques has become more popular since, unlike white-box approaches, they do not need to deal with the technology or transformation language employed in the implementation of the model transformation. In relation to this, our motivations to present a white-box testing approach are twofold: On the one hand, our approach could be combined with black-box approaches to facilitate the creation of mixed testing strategies. On the other hand, at the time of implementing a model transformation, a formal specification is not always available, thus making difficult or even impossible the application of black-box approaches to generate input test models. In these scenarios, white-box testing techniques can be of special relevance.

3 ATLTest: Test Input Models for ATL Transformations

3.1 Overall Picture

ATLTest is a white-box test generation approach for ATL transformations. In traditional white-box testing, test generation is a 2-step process in which, typically a control flow graph or a data flow graph is generated in the first place, out of an analysis of the source code, and then, a set of test cases is obtained from traversing the graph a specific number of times, usually determined by some coverage criteria, like for example decision coverage. Although essentially the same, compared to traditional white-box approaches, the test generation process in ATLTest exhibits some differences, basically due to the mixed declarative and imperative model transformation language constructs of ATL and the complex nature of model transformation inputs.

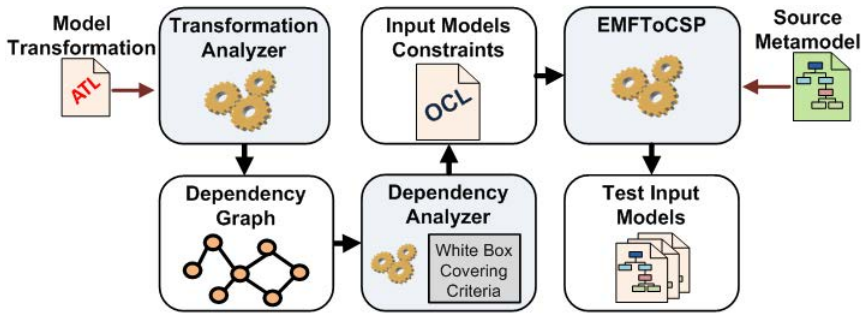


Fig. 2. ATLTest: Overall picture

More specifically, the test generation process in ATLTest, depicted in Fig. 2, consists of three separate steps. In the first one, the ATL transformation is analyzed and a graph abstracting the relevant information for the test generation phase is produced. This graph, called "dependency graph", plays the same role in ATLTest that control flow graphs or data flow graphs play in other traditional approaches, although it is substantially different in nature. For now it suffices to say that the dependency graph represents groups of interrelated conditions expressed in the OCL, that must be hold (totally or partially) by the test input models.

Once the analysis of the ATL transformation is done, the second step is to traverse the dependency graph a number of times which, as for traditional approaches, is determined by some coverage criteria. Traversing the dependency graph implies setting truth values for the different conditions in the graph and, therefore, each traversal will yield a set of constraints that symbolizes a family of relevant test cases for the transformation (i.e. the constraints characterize the structure/values of possible sample input models corresponding to that test case).

In the last step, the actual test cases (i.e. the test input models to be used when executing the transformation) are created by computing models conforming to the source metamodel and satisfying the constraints for the test case. This computation can be performed using any of the SAT-based or CSP-based solvers available. In particular, we use EMFtoCSP[1] [12] to generate the input test models. EMFtoCSP is an Eclipse[2]-integrated tool for the automatic verification of UML models and EMF models annotated with OCL constraints by means of reexpressing them as a constraint satisfaction problem. In the context of model transformation testing, EMFtoCSP will generate solutions (i.e. sample models) that satisfy both the source metamodel and the additional OCL expressions resulting from the graph traversal. A single sample model suffices to cover the corresponding test case.

In the next sections we will describe in more detail the foundations and rationale behind ATLTest.

[1] `http://code.google.com/a/eclipselabs.org/p/emftocsp/`

[2] `http://www.eclipse.org/`

3.2 Running Example

To illustrate our approach we will be using as a running example the following transformation that converts *publications* into *books* (see Fig. 3). In a nutshell, the model transformation contains two rules (*Publication2Book* and *PubSection2Chapter*) to respectively transform "Publication" and "PubSection" input elements into "Book" and "Chapter" output elements. Those elements are only transformed if the respective flags "isBook" and "isChapter" are activated.

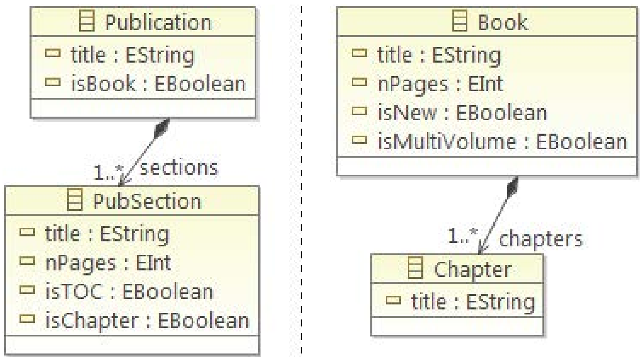


Fig. 3. Source (left) and target (right) metamodels for the running example

```
module Publication2Book;
create OUT : Book from IN : Publication;
rule Publication2Book {
  from p: Publication!Publication (p.isBook)
  to   b: Book!Book (
          title<-p.title,
          isMultiVolume<-p.sections->select(s| s.isChapter)->
          size()>25 and p.sections->select(s| s.isTOC)->size()>2,
          chapters<-p.sections->select(s| s.isChapter),
          nPages<-p.sections->collect(s| s.nPages)->sum()   )
}
rule PubSection2Chapter {
  from ps: Publication!PubSection (ps.isChapter)
  to    c: Book!Chapter ( title<-ps.title )
}
```

4 Dependency Graph Generation

The ATL language includes a variety of constructs (matched rules, lazy rules, helpers, etc) but in most of them OCL plays a key role. Therefore any white-box testing approach for ATL must devote a special attention to the OCL expressions appearing in the transformation.

In fact, OCL expressions are at the heart of the mechanism to create the dependency graph. In a nutshell, the majority of nodes and arcs are generated out of the analysis of certain OCL expressions found in the rules and helpers making up the ATL transformation, thus forming the building blocks of the dependency graph. The analysis of the rules and helpers containing those OCL expressions extends and interconnects those building blocks. The process is described in more detail in the following subsections.

4.1 Analysis of OCL Expressions

OCL expressions have a clear impact on the number and structure of interesting input models to use as tests for the model transformation. To ensure the coverage of the model transformation we should make sure that the test models evaluate to a different result the several OCL expressions in the transformation.

Let's consider the OCL expression `p.sections->select(s|s.isChapter)` extracted from the running example. The expression is part of a binding in the first rule, aimed at generating as many "Chapter" elements in the output model as "PubSection" elements with the flag "isChapter" set to "True" are present in the input model. Clearly, when looking at this expression we immediately think of different situations that should be tested, e.g. "What happens if there are no "PubSection" elements in the input model?" or "What happens if none of the "PubSection" elements are flagged as chapters?". Therefore, input models that test each situation (i.e. an input model with no "PubSections", a model with "PubSections", a model with "PubSections" in which none of them is flagged as a chapter,...) should be generated by our method.

Each question above can be characterized by means of a boolean OCL expression (for the former example `PubSection::allInstances()->notEmpty()` and `PubSection::allInstances()->select(s|s.ischapter)->notEmpty()` could be those expressions). Each expression would constitute a node in the dependency graph (meaning that the generated tests may include the condition in the node depending on how the graph is traversed as explained in the next section). It is also worth noting that it does not make much sense to check the second condition if the first one does not hold (we cannot create at the same time a model with no "PubSection" elements and a non-empty list of "PubSection" elements, some of them flagged as chapters), which means that the two conditions are somehow interrelated. This interrelation is the reason why we call the graph, dependency graph. There is a dependency between the two conditions, expressed as an arc between the two nodes. Obviously, these arcs play a key role in the traversal of the graph during the test generation phase.

In the rest of the section we generalize this discussion to arbitrary OCL expressions. We have identified three different big groups of OCL expressions relevant to the process sketched above, namely, expressions in the context of collections (Table 1), iterative operations (Table 2) and boolean expressions (Table 3). Each row in the tables show how the dependency graph is extended when finding an expression of that type in an ATL construct. The dependency graph is expressed as two ordered sets that contain the nodes (V) and the arcs (E) in the order they are created, where nodes are described with an OCL expression, and arcs are expressed as "(x,y)", "x" and "y" being the positions of the source and target nodes in the corresponding set. In this regard, "last" is used to make reference to the last position in a set, and in the case of complex OCL expressions, "$G_x(V)$" and "$G_x(E)$" make reference to the respective sets of nodes and arcs obtained from the analysis of the source expression "x". Similar for "$G_{body}(V)$" and "$G_{body}(E)$" in table 2.

One important remark is that, in order to be considered for analysis, all these OCL expressions must reference at least one element of the input metamodel, since these are the most relevant for test generation. The identification of the OCL expressions suitable for analysis can be done by traversing the abstract syntax tree of the OCL expressions in the ATL transformation.

To finish this subsection, we illustrate how to create nodes 3, 4, 5, 6 and 7 of the dependency graph in Fig. 6 by applying the information in the tables to the following expression from the running example:

```
isMultiVolume<-p.sections->select(s| s.isChapter)->size() > 25
and p.sections->select(s| s.isTOC)->size() > 2              (exp1)
```

To begin with, the OCL expression at the right of "<-", matches entry 10 in table 3 using "and" as "Op". According to this entry, the 2-step process depicted in Fig. 4 must be carried out. That is, subexpressions at the left and at the right of "and" must be analyzed, thus yielding several nodes and arcs, and then some of those nodes are merged. Finally all the nodes are interconnected.

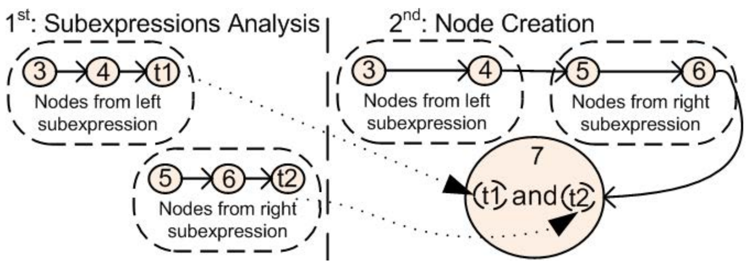

1st: Subexpressions Analysis
2nd: Node Creation
3
4
t1
Nodes from left subexpression
5
6
t2
Nodes from right subexpression
3
4
Nodes from left subexpression
5
6
Nodes from right subexpression
7
t1 and t2

Fig. 4. Actions to carry out when applying entry 10 in table 3 to the running example

The expression at the left of "and" in (exp1) is

```
p.sections->select(s| s.isChapter)->size() > 25             (exp2)
```

that matches entry 11 in table 3 where "CompOp" is ">" and "LitValue" is "25". Fig. 5 illustrates the process to be carried out when instructions in this entry are followed. The subexpression on the left side is analyzed in the first place, this way yielding nodes 3 and 4, and then, the node "t1" is created.

Table 1. Nodes and arcs generated out of OCL operations in the context of a collection

	OCL Expression	**G=(V,E)**
1	$Obj_c.[nav \mid nav \rightarrow notEmpty()]$	$V = \{C :: allInstances() \rightarrow select(c \mid c.nav \rightarrow notEmpty()) \rightarrow notEmpty()\}$
2	$C :: allInstances()[\rightarrow notEmpty()]$	$V = \{C :: allInstances() \rightarrow notEmpty()\}$
3	$Obj_c.nav \rightarrow isEmpty()$	$V = \{C :: allInstances() \rightarrow select(c \mid c.nav \rightarrow isEmpty()) \rightarrow notEmpty()\}$
4	$C :: allInstances() \rightarrow isEmpty()$	$V = \{C :: allInstances() \rightarrow isEmpty()\}$
5	$c \rightarrow isEmpty()$	$V = \{c \rightarrow isEmpty()\} \cup G_c(V)$, $E = \{(G_c(V)[last], 1)\} \cup G_c(E)$
6	$c \rightarrow notEmpty()$	$V = \{c \rightarrow notEmpty()\} \cup G_c(V)$, $E = \{(G_c(V)[last], 1)\} \cup G_c(E)$
7	$c \rightarrow [size() \mid last() \mid sum() \mid append(o) \mid flatten() \mid first() \mid including(o) \mid prepend(o)]$	$V = G_c(V)$, $E = G_c(E)$
8	$c \rightarrow [includes(o) \mid count(o) \mid indexOf(o) \mid excluding(o)]$	$V = \{c \rightarrow includes(o)\} \cup G_c(V)$, $E = \{(G_c(V)[last], 1)\} \cup G_c(E)$
9	$c \rightarrow excludes(o)$	$V = \{c \rightarrow excludes(o)\} \cup G_c(V)$, $E = \{(G_c(V)[last], 1)\} \cup G_c(E)$
10	$c \rightarrow includesAll(cl)$	$V = \{c \rightarrow includesAll(cl)\} \cup G_c(V) \cup G_{cl}(V)$, $E = \{(G_c(V)[last], G_{cl}(V)[1]), (G_{cl}(V)[last], 1)\} \cup G_c(E) \cup G_{cl}(E)$
11	$c \rightarrow excludesAll(cl)$	$V = \{c \rightarrow excludesAll(cl)\} \cup G_c(V) \cup G_{cl}(V)$ $E = \{(G_c(V)[last], G_{cl}(V)[1]), (G_{cl}(V)[last], 1)\} \cup G_c(E) \cup G_{cl}(E)$
12	$c \rightarrow union(cl)$	$V = G_c(V) \cup G_{cl}(V)$, $E = \{(G_c(V)[last], G_{cl}(V)[1])\} \cup G_c(E) \cup G_{cl}(E)$
13	$c \rightarrow [insertAt(n, o) \mid at(n)]$	$V = \{c \rightarrow size() \geq n\} \cup G_c(V)$ $E = \{(G_c(V)[last], 1)\} \cup G_c(E)$
14	$c \rightarrow subSequence(l, u)$	$V = \{c \rightarrow size() \geq u\} \cup G_c(V)$ $E = \{(G_c(V)[last], 1)\} \cup G_c(E)$
15	$c \rightarrow [intersection(cl) \mid symetricDifference(cl)]$	$V = \{c \rightarrow includesAll(cl)\ or\ cl \rightarrow includesAll(c)\} \cup G_c(V) \cup G_{cl}(V)$, $E = \{(G_c(V)[last], G_{cl}(V)[1]), (G_{cl}(V)[last], 1)\} \cup G_c(E) \cup G_{cl}(E)$

Now let's see in detail how nodes 3 and 4 are generated. The expression on the left of (exp2) is

```
p.sections->select(s| s.isChapter)->size()                    (exp3)
```

that matches entry 7 in table 1. According to this entry, it is necessary to analyze the source collection of (exp3), that is:

```
p.sections->select(s| s.isChapter)                            (exp4)
```

It matches entry 6 in table 2. This entry indicates that (exp4) has the form `c->select(body)` and therefore "c" and "body" expressions must be analyzed.

Table 2. Generation of nodes and arcs out of OCL iterative operations

	OCL Expression	**G=(V,E)**
1	$c \to exists(body)$	$V = \{c \to exists(body)\} \cup G_c(V) \cup G_{body}(V)$, $E = \{(G_c(V)[last], G_{body}(V)[1]), (G_{body}(V)[last], 1)\} \cup G_c(E) \cup G_{body}(E)$
2	$c \to forAll(body)$	$V = \{c \to forAll(body)\} \cup G_c(V) \cup G_{body}(V)$, $E = \{(G_c(V)[last], G_{body}(V)[1]), (G_{body}(V)[last], 1)\} \cup G_c(E) \cup G_{body}(E)$
3	$c \to isUnique(body)$	$V = \{c \to isUnique(body)\} \cup G_c(V) \cup G_{body}(V)$, $E = \{(G_c(V)[last], G_{body}(V)[1]), (G_{body}(V)[last], 1)\} \cup G_c(E) \cup G_{body}(E)$
4	$c \to one(body)$	$V = \{c \to one(body)\} \cup G_c(V) \cup G_{body}(V)$, $E = \{(G_c(V)[last], G_{body}(V)[1]), (G_{body}(V)[last], 1)\} \cup G_c(E) \cup G_{body}(E)$
5	$c \to [collect(body) \vert sortedBy(body)]$	$V = G_c(V)$, $E = G_c(E)$
6	$c \to [reject(body) \vert any(body) \vert select(body)]$	$V = G_c(V) \cup G_{body}(V)$, $E = \{(G_c(V)[last], G_{body}(V)[1])\} \cup G_c(E) \cup G_{body}(E)$

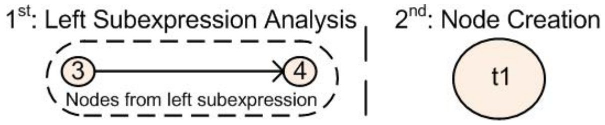

1st: Left Subexpression Analysis
3
4
Nodes from left subexpression
2nd: Node Creation
t1

Fig. 5. Actions to carry out when applying entry 11 in table 3 to the running example

In (exp4), “c” is `p.sections` and “body” is `s.isChapter`. They match respectively entry 1 in table 1 and entry 2 in table 3. This way, we finally obtain nodes 3 and 4 that can be seen in Fig. 6. It is important to remember that the creation of nodes 3 and 4 is just the first step in the analysis of (exp2), as exposed in Fig. 5. Now it is time to complete the analysis of this expression by creating the node “t1”. This node is made up by the following OCL expression:

```
p.sections->select(s| s.isChapter)->size() > 25              (t1)
```

It is the time to remember that the analysis of (exp2) is just the analysis of the left subexpression of (exp1). As can be seen in Fig. 4 the analysis of (exp1) continues with the analysis of its right subexpression. We omit a detailed description of this analysis, though, since it is very similar to the one just described. It suffices to say that the analysis of the right subexpression of (exp1) yields nodes 5 and 6 that can be seen in Fig. 6, as well as node “t2”, made up by the following OCL expression:

```
p.sections->select(s| s.isTOC)->size() > 2                    (t2)
```

Finally, applying last step shown in Fig. 4, node 7 is created out of the union of nodes “t1” and “t2”, expressed in terms of the “allInstances()” operator, and the

Table 3. Generation of nodes and arcs out of boolean OCL operations

	Boolean OCL Expression	**G=(V,E)**
1	$[True\|False]$	$V = \varnothing, E = \varnothing$
2	$[not]Obj_A.boolAttr$	$V = \{A :: allInstances() \rightarrow select(a\|[not]a.boolAttr) \rightarrow notEmpty()\}$
3	$Obj_A.[attr\|nav].oclIsUndefined()$	$V = \{A :: allInstances() \rightarrow select(a\|a.[attr\|nav].oclIsUndefined()) \rightarrow notEmpty()\}$
4	$expr.oclIsUndefined()$	$V = \{expr \rightarrow oclIsUndefined()\} \cup G_{expr}(V)$, $E = \{(G_{expr}(V)[last], 1)\} \cup G_{expr}(E)$
5	$expr.oclIsKindOf(t)$	$V = \{expr \rightarrow oclIsKindOf(t)\} \cup G_{expr}(V)$, $E = \{(G_{expr}(V)[last], 1)\} \cup G_{expr}(E)$
6	$expr.oclIsTypeOf(t)$	$V = \{expr \rightarrow oclIsTypeOf(t)\} \cup G_{expr}(V)$, $E = \{(G_{expr}(V)[last], 1)\} \cup G_{expr}(E)$
7	$Obj_A.attr\ CompOp\ LitValue$	$V = \{A :: allInstances() \rightarrow select(a\| a.attr\ CompOp\ LitValue) \rightarrow notEmpty()\}$
8	$Obj_A.attr\ Op\ Obj_B.attr$	$V = \{A :: allInstances() \rightarrow select(a\| B :: allInstances() \rightarrow exists(b\| a.attr\ CompOp\ b.attr)) \rightarrow notEmpty()\}$
9	$Obj_A.attr\ Op\ Obj_B.attr\ Op\ ...\ Op\ Obj_N.attr$	$V = \{A :: allInstances() \rightarrow select(a\|B :: allInstances() \rightarrow exists(b\|... \rightarrow exists(n\|a.attr\ Op\ b.attr\ Op\ ...\ Op\ n.attr)...)) \rightarrow notEmpty()\}$
10	$expr_1\ Op\ expr_2$	$V = \{G_{expr_1}(V)[last]\ Op\ G_{expr_2}(V)[last]\} \cup \{G_{expr_1}(V)[1], ..., G_{expr_1}(V)[last-1]\} \cup \{G_{expr_2}(V)[1], ..., G_{expr_2}(V)[last-1]\}$, $E = \{(G_{expr_1}(V)[last-1], G_{expr_2}(V)[1]), (G_{expr_2}(V)[last], 1)\} \cup \{G_{expr_1}(E)[1], ..., G_{expr_1}(E)[last-1]\} \cup \{G_{expr_2}(E)[1], ..., G_{expr_2}(E)[last-1]\}$
11	$expr\ CompOp\ LitValue$	$V = \{expr\ CompOp\ LitValue\} \cup G_{expr}(V)$, $E = \{(G_{expr}(V)[last], 1)\} \cup G_{expr}(E)$

different nodes created during the process are interconnected. The final result can be seen in Fig. 6.

The analysis of the rest of OCL expressions in the sample model transformation can be carried out in the same way.

4.2 Analysis of Rules and Helpers

As we have seen, the analysis of OCL expressions yields the building blocks of the dependency graph. In this subsection we cover the analysis of rules and helpers, coarse-grained elements of ATL transformations.

There are different types of rules in ATL, namely, matched rules, lazy rules and called rules. The first two are declarative rules while the last one is an imperative type of rule.

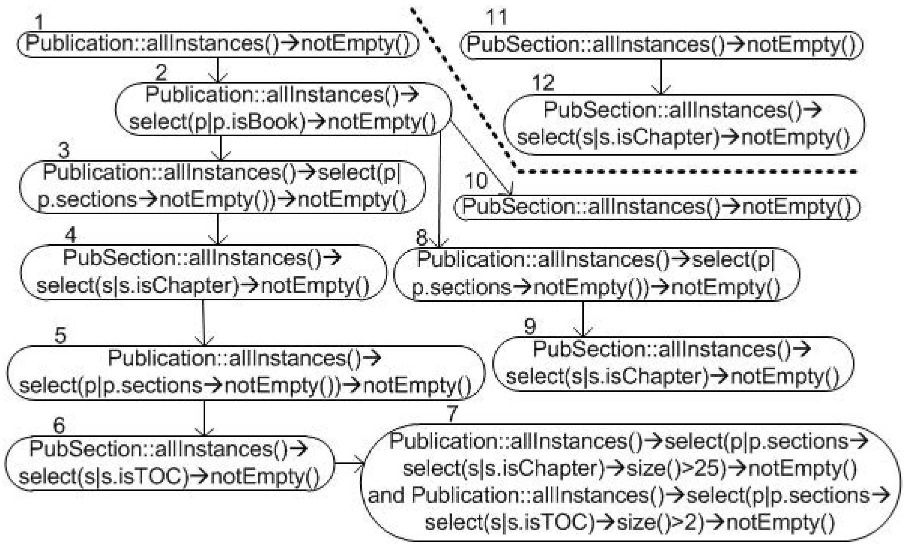

1
Publication::allInstances()→notEmpty()
2
Publication::allInstances()→select(p|p.isBook)→notEmpty()
3
Publication::allInstances()→select(p|p.sections→notEmpty())→notEmpty()
4
PubSection::allInstances()→select(s|s.isChapter)→notEmpty()
5
Publication::allInstances()→select(p|p.sections→notEmpty())→notEmpty()
6
PubSection::allInstances()→select(s|s.isTOC)→notEmpty()
7
Publication::allInstances()→select(p|p.sections→select(s|s.isChapter)→size()>25)→notEmpty() and Publication::allInstances()→select(p|p.sections→select(s|s.isTOC)→size()>2)→notEmpty()
8
Publication::allInstances()→select(p|p.sections→notEmpty())→notEmpty()
9
PubSection::allInstances()→select(s|s.isChapter)→notEmpty()
10
PubSection::allInstances()→notEmpty()
11
PubSection::allInstances()→notEmpty()
12
PubSection::allInstances()→select(s|s.isChapter)→notEmpty()

Fig. 6. Dependency graph of the example, made up by two connected components

The analysis of a declarative rule focuses on the `from` section of the rule, that indicates the conditions that trigger the rule, the `to` section of the rule, that describes how elements of the target model are created, and the optional `do` section of the rule, used to enable the specification of imperative statements.

The analysis of the `from` section produces a node with the OCL expression `in_type::allInstances()->notEmpty()`, where "in_type" refers to the model element that will be matched by the rule. Optionally, this section can include a boolean OCL expression, as a filter to limit the "in_type" elements that can trigger the rule. When present, this filter is analyzed according to the instructions of subsection 4.1 and, in this case, the node created in the first place is connected to the first node rendered by the filter analysis.

Returning to the running example, the analysis of the `from` section of the rule "Publication2Book", that includes the condition `p.isBook`, produces nodes 1 and 2 in Fig. 6. Analogously, the `from` section of the rule "PubSection2Chapter" generates nodes 11 and 12 that made up the second connected component of the dependency graph.

The `to` section of a declarative rule is, essentially, a collection of bindings describing how elements of the target metamodel are created. Each binding has the form `feature-name <- exp`, being "exp" an OCL expression. The result of analyzing this section is a number of interconnected nodes, obtained from the analysis of each "exp" element as explained in subsection 4.1. Finally, the first node in each of the groups of nodes rendered is connected to the last node in the group of nodes obtained from the analysis of the `from` section of the rule.

The `do` section of a declarative rule allows the specification of imperative statements. This section is analyzed by looking for OCL expressions suitable for analysis. When found, those expressions are analyzed according to the directions of subsection 4.1. This approach is also applied at the time of analyzing called rules.

To finish the description of the dependency graph generation process, one word about ATL helpers. Helpers can be viewed as the ATL equivalent to methods and can be called from different points in an ATL transformation. Each helper has a body, specified as an OCL expression. If during the analysis of the elements described above and in subsection 4.1, a call to a helper is found, then its body is analyzed like any other OCL expression and the rendered nodes are included as resulting from the analysis of the element where the call was found.

One last remark that is worth mentioning is that depending on the complexity of the ATL transformation under analysis (number of rules, presence of imperative sections, etc.), the resulting dependency graph can be made up by more than one connected component.

5 Test Input Models Generation

Once the dependency graph is created, the next step consists in traversing it a number of times, each time determining the set of constraints a new test case must fulfill. The process is directed by a coverage criterion, which eventually determines the number of traversals, and consequently, the number of test cases to be generated.

In white-box testing, coverage criteria help designers to select the structural elements of the software (model transformations in this case) that will be the focus of the testing and to determine the desired intensity of the testing efforts. The coverage criteria drive the creation of the tests to make sure the tests cover the selected parts of the transformation and do it enough to gain the desired confidence on their correctness. The fact that a test suite covers an element means that it exists at least one test case that exercises that element. This is known as coverage analysis.

Branches in the program logic are elements typically selected as object of coverage analysis in white-box testing. There are a number of classical white-box coverage criteria that follow this approach, like for example, “condition coverage” or “multiple-condition coverage” [19]. Both focus on making sure that all branches in the program are covered, but they differ on how they exercise conditional branches where the condition is not atomic. In the case of “condition coverage”, complete coverage is achieved by simply ensuring that the test cases exercise each branch with all possible outcomes at least once (i.e. for a boolean branch, the test suite must include a test case where the branch evaluates to “False” and one where it evaluates to “True”). However, “multi-condition coverage” requires the test suite to include a test case for each individual combination of truth values of the subconditions conforming the branch condition.

These and other similar criteria can be easily adapted to our approach. Since in the dependency graph each node contains a boolean expression, condition coverage and multi-condition coverage can be applied by considering each node as a branch, with the particularity that every time the condition in the node evaluates to "False" the traversal of the actual connected component ends and goes on with the next one. In other case, a neighbour node is visited and the traversal continues.

This way, the application of the two coverage criteria consists on traversing the dependency graph a number of times, each time asigning either different output values to each OCL expression (condition coverage), or different combinations of truth values to each component of a complex OCL expression (multi-condition coverage). After "n" traversals, "n" sets of constraints to characterize "n" test cases will have been obtained.

Eventually, once the sets of constraints have been obtained, the execution of EMFtoCSP over each set will yield the set of input models to test the model transformation[3].

Retaking our example, we are going to show what the results of one traversal of the graph shown in Fig. 6 would be in every approach. Let's suppose that the sequence of truth values assigned to the nodes of the first connected component is <1,True>, <2,True>, <3,True>, <4,True>, <5,True>, <6,True>, and then, in the case of "condition coverage" node 7 is set to <7,True>, and in the case of "multi-condition coverage" is set to <7,(False,True)>. In the second connected component, the expressions will be set as <11,True>, <12,True>, for both approaches.

Applying "condition coverage", the constraints obtained are:

```
Publication::allInstances()->notEmpty()=true
Publication::allInstances()->select(p|p.isBook)->notEmpty()=true
Publication::allInstances()->select(p|p.sections->notEmpty())
   ->notEmpty()=true
PubSection::allInstances()->select(s|s.isChapter)->notEmpty()=true
Publication::allInstances()->select(p|p.sections->notEmpty())
   ->notEmpty()=true
PubSection::allInstances()->select(s|s.isTOC)->notEmpty()=true
Publication::allInstances()->select(p|p.sections->
   select(s|s.isChapter)->size()>25)->notEmpty() and
   Publication::allInstances()->select(p|p.sections->
   select(s|s.isTOC)->size()>2)->notEmpty()=true
PubSection::allInstances()->notEmpty()=true
PubSection::allInstances()->select(s|s.isChapter)->notEmpty()=true
```

Running EMFtoCSP over the input metamodel constrained with the expressions above yields the model that can be seen in Figure 7 a).

[3] Some assignments can cause contradictory sets of OCL expressions (e.g. if the same subexpressions are used in two connected components and they are assigned different truth values in the same iteration). In those situations, EMFtoCSP will return an empty result and the test case will be discarded.

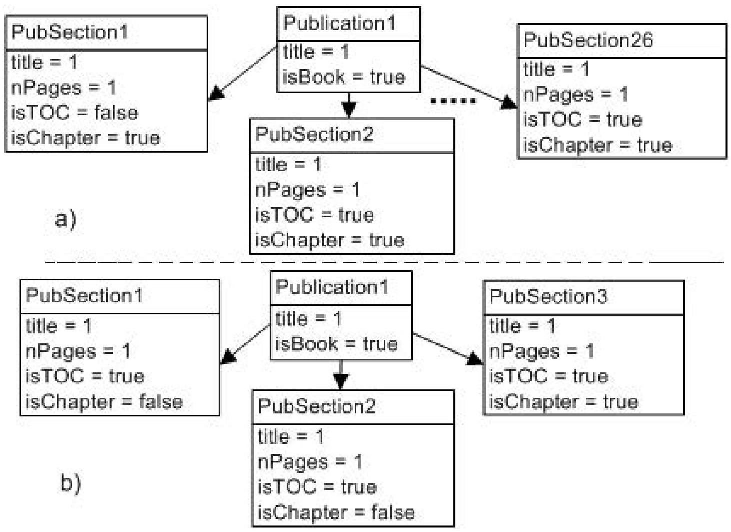


Fig. 7. Results of the example

For "multi-condition coverage", only the expression of node 7 changes:

```
Publication::allInstances()->select(p|p.sections->
   select(s|s.isChapter)->size()> 25)->notEmpty()=false and
   Publication::allInstances()->select(p|p.sections->
   select(s|s.isTOC)->size()>2)->notEmpty()=true
```

Running again EMFtoCSP, we obtain the model of Figure 7 b).

6 Related Work

One of the most important tasks when testing a model transformation is the creation of an adequate set of test input models. Currently, the majority of approaches facing this challenge are based on black-box techniques [11, 9, 10, 16, 21, 22, 3, 6, 20, 8, 13].

As far as we know only two white-box approaches for transformation testing have been proposed [9, 15]. Both address the identification of the relevant parts of the input metamodel to be exercised by the tests: by looking at the transformation definition they detect the subset of the metamodel (and possible relevant values for the metamodel attributes) that is accessed during the transformation and thus focus the generation of tests on that subset. In our case, the coverage of the input metamodel is derived from the test cases generated when addressing the coverage of the model transformation internal structure. This analysis of the internal transformation structure also guarantees that our tests exercise all branches in the transformation, this way maximizing their effectiveness.

White-box techniques can also be used in coverage analysis, to measure the quality of the generated test models. Regarding this, [17] proposes a number of white-box coverage measures for ATL transformations, namely rule coverage, instruction coverage and decision coverage, that are used to check how a number of test cases cover ATL transformations. This could be useful to check the quality of the tests generated with our approach, especially for model transformations where the designer may want to limit the number of tests generated.

It is worth noting that the generation of test cases out of OCL expressions is not exclusive of model transformation testing, on the contrary, it is also an important method for the verification and validation of UML/OCL specifications. Regarding this, [7] and [2] propose approaches to generate test data from OCL specifications, based on the utilisation of Higher-Order Logic and constraint solving techniques, respectively. Another approach based on the utilisation of constraint solving techniques is proposed in [1] to generate test cases out of UML specifications, although in this case only a limited subset of the OCL is supported. Finally, [23] proposes an approach to evaluate the quality of test cases generated from OCL expressions based on the utilization of several coverage criteria.

7 Conclusions and Future Work

We have presented ATLTest, a white-box testing approach for the generation of test input models for ATL transformations. Our approach tries to optimize the effectiveness of the generated tests by maximizing the coverage of the internal structure of the model transformation under analysis. ATLTest could be combined with black-box testing techniques to create mixed test generation approaches. In ATLTest, each test case is characterized by a set of OCL expressions that define the possible structure of the test input models for that test case. Sample test models satisfying the OCL constraints are created automatically using the EMFtoCSP tool.

As further work, we plan to extend our to approach to cover other transformation languages like QVT. We would also like to study complexity metrics like cyclomatic complexity [18] to establish a limit on the number of test cases that need to be created, something that can be specially useful when testing large transformations. Finally, ATLTest is a first step in the development of a full model transformation testing framework called ATLUnit, where different test cases generation approaches could be combined.

References

1. Aertryck, L.V., Jensen, T.: Uml-casting: Test synthesis from uml models using constraint resolution. In: Proceedings of AFADL 2003 (Approches Formelles Dans L'Assistance Au Dévelopment De Logiciel) (2003)
2. Aichernig, B.K., Salas, P.A.P.: Test case generation by ocl mutation and constraint solving. In: QSIC, pp. 64–71. IEEE Computer Society (2005)
3. Baudry, B., Dinh-Trong, T., Mottu, J.M., Simmonds, D., France, R., Ghosh, S., Fleurey, F., Le Traon, Y.: Model transformation testing challenges. In: Proceedings of IMDT Workshop in conjunction with ECMDA 2006 (2006)

4. Baudry, B., Ghosh, S., Fleurey, F., France, R.B., Traon, Y.L., Mottu, J.M.: Barriers to systematic model transformation testing. Commun. ACM 53(6), 139–143 (2010)
5. Beizer, B.: Software Testing Techniques, 2nd edn. Int. Thomson Computer Press (1990)
6. Brottier, E., Fleurey, F., Steel, J., Baudry, B., Traon, Y.L.: Metamodel-based test generation for model transformations: an algorithm and a tool. In: ISSRE, pp. 85–94. IEEE Computer Society (2006)
7. Brucker, A.D., Krieger, M.P., Longuet, D., Wolff, B.: A Specification-Based Test Case Generation Method for UML/OCL. In: Dingel, J., Solberg, A. (eds.) MODELS 2010. LNCS, vol. 6627, pp. 334–348. Springer, Heidelberg (2011)
8. Fiorentini, C., Momigliano, A., Ornaghi, M., Poernomo, I.: A Constructive Approach to Testing Model Transformations. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 77–92. Springer, Heidelberg (2010)
9. Fleurey, F., Steel, J., Baudry, B.: Validation in model-driven engineering: testing model transformations. In: Proceedings of first Int. Workshop on Model, Design and Validation, pp. 29–40 (November 2004)
10. Fleurey, F., Baudry, B., Muller, P.A., Traon, Y.L.: Qualifying input test data for model transformations. Software and System Modeling 8(2), 185–203 (2009)
11. Gogolla, M., Vallecillo, A.: *Tract*able Model Transformation Testing. In: France, R.B., Kuester, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 221–235. Springer, Heidelberg (2011)
12. González, C.A., Büttner, F., Clarisó, R., Cabot, J.: Emftocsp: A tool for the lightweight verification of emf models. In: Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA), pp. 44–50 (June 2012)
13. Guerra, E.: Specification-Driven Test Generation for Model Transformations. In: Hu, Z., de Lara, J. (eds.) ICMT 2012. LNCS, vol. 7307, pp. 40–55. Springer, Heidelberg (2012)
14. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
15. Küster, J.M., Abd-El-Razik, M.: Validation of Model Transformations – First Experiences Using a White Box Approach. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 193–204. Springer, Heidelberg (2007)
16. Lamari, M.: Towards an automated test generation for the verification of model transformations. In: SAC, pp. 998–1005. ACM (2007)
17. Mc Quillan, J.A., Power, J.F.: White-box coverage criteria for model transformations. Department of Computer Science. National University of Ireland (July 2009)
18. McCabe, T.J.: A complexity measure. IEEE Trans. Software Eng. 2(4), 308–320 (1976)
19. Myers, G.J.: The Art of Software Testing, 2nd edn. John Wiley & Sons, Inc. (2004)
20. Sen, S., Baudry, B., Mottu, J.M.: On combining multi-formalism knowledge to select models for model transformation testing. In: ICST, pp. 328–337. IEEE Computer Society (2008)
21. Sen, S., Baudry, B., Mottu, J.M.: Automatic Model Generation Strategies for Model Transformation Testing. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 148–164. Springer, Heidelberg (2009)
22. Wang, J., Kim, S.K., Carrington, D.A.: Automatic generation of test models for model transformations. In: Australian Software Engineering Conference, pp. 432–440. IEEE Computer Society (2008)
23. Weißleder, S., Schlingloff, B.H.: Quality of automatically generated test cases based on ocl expressions. In: ICST, pp. 517–520. IEEE Computer Society (2008)

Empirical Evaluation on FBD Model-Based Test Coverage Criteria Using Mutation Analysis

Donghwan Shin, Eunkyoung Jee, and Doo-Hwan Bae

Dept. of Computer Science, KAIST, Daejeon, Republic of Korea
{donghwan,ekjee,bae}@se.kaist.ac.kr

Abstract. Function Block Diagram (FBD), one of the PLC programming languages, is a graphical modeling language which has been increasingly used to implement safety-critical software such as nuclear reactor protection software. With increased importance of structural testing for FBD models, FBD model-based test coverage criteria have been introduced. In this paper, we empirically evaluate the fault detection effectiveness of the FBD coverage criteria using mutation analysis. We produce 1800 test suites satisfying the FBD criteria and generate more than 600 mutants automatically for the target industrial FBD models. Then we evaluate mutant detection of the test suites to assess the fault detection effectiveness of the coverage criteria. Based on the experimental results, we analyze strengths and weaknesses of the FBD coverage criteria, and suggest possible improvements for the test coverage criteria.

Keywords: Function block diagram, mutation analysis, test coverage criteria.

1 Introduction

Function Block Diagram (FBD) is a graphical modeling language for Programmable Logic Controller (PLC) programs [1]. Recently, FBD has been used to implement safety-critical system software such as nuclear reactor protection software [2]. For such safety-critical software, structural testing is demanded by regulation authorities such as Nuclear Regulatory Commission (NRC) [3].

With the growing importance of structural testing for FBD models, Jee et al. [4] have developed three FBD model-based test coverage criteria: Basic Coverage (BC), Input Condition Coverage (ICC), and Complex Condition Coverage (CCC) criteria. The proposed test coverage criteria are useful in the aspects of reflecting data flow-centric characteristics of FBD and giving testers intuitive structural coverage concepts. However, evaluation for the coverage criteria in terms of fault detection has not been done and important questions remain: How effective is each of the three test coverage criteria in fault detection? What types of faults are more likely detected by those criteria?, and so on. Answers to these questions would bring important outcomes on the validity of the FBD coverage criteria and therefore, on the model-based testing for FBD models.

To investigate test coverage criteria in terms of fault detection, we need to prepare experiments with many faults. Mutation analysis measures the fault detection ability of a test suite by seeding artificial defects, *mutants*, into a model. If a test suite *kills* a mutant, it means that the mutant is detected by the test suite. Since defects are made by simple variants of a model, operators to be used to make the variants are called *mutation operators*. Mutation analysis provides a well-defined fault-seeding process and gives potentially large number of faults which increase the statistical significance of results [5]. Mutation analysis has been widely used to compare and evaluate the fault detection effectiveness of test suites or test coverage criteria in many studies [5–7].

In this paper, we evaluate three FBD model-based test coverage criteria in terms of the fault detection effectiveness using mutation analysis. We automatically generate many variants (mutants) from the original target FBD models and generate test suites with respect to the three coverage criteria. By investigating in which situation the faults are detected and which test suites trigger such detections, we evaluate the fault detection effectiveness of the coverage criteria. We analyze strengths and weaknesses of the coverage criteria and suggest possible improvements based on the experiment results.

The remainder of the paper is organized as follows: Section 2 presents related work for the mutation analysis and coverage criteria assessment. Section 3 explains basic concepts of FBD and definitions for the FBD model-based coverage criteria. Section 4 describes research questions and experimental strategies. Section 5 reports the results of the experiment followed by analysis and discussion. We conclude the paper at Section 6.

2 Related Work

Evaluation on the fault detection effectiveness of test coverage criteria has been studied in many literatures [5, 8, 9].

Andrews et al. [5] assessed and compared Block, Decision, C-Use, and P-Use coverage criteria using mutation analysis. They investigated the relative cost, fault detection effectiveness and cost-effectiveness of each of the four criteria using mutation analysis. The fault detection effectiveness was measured by mutant detection ratio and the cost was measured by test suite size. They randomly extracted and generated test suites from the test pool for the four coverage criteria with 45-95% coverage levels. Although the paper is similar to our work in the aspect of the evaluation of coverage criteria using mutation analysis, they experimented on the code coverage whereas our work is on the model-based test coverage. In test suite generation, we purpose achieving 100% coverage level using a constraint solver. They reported the relationship among fault detection, test suite size, and coverage criteria. However, they didn't analyze strengths and weaknesses for each criterion as we do.

Hutchins et al. [8] conducted experiments to investigate the effectiveness of Def-Use (DU) coverage and Decision coverage. The fault detection effectiveness was measured by the ratio of test suites which detect faults. They generated test suites with the aid of a test script generation tool and manually seeded a number of faults in seven small programs. In our work, mutation operators are used to automatically generate faults. While they reported the relationship among coverage, test suite size, and fault detection, they did not investigate strengths and weaknesses of the coverage criteria. Moreover, they were based on the source code whereas we are based on the FBD models.

Li et al. [9] reported the experimental comparison of edge-pair coverage, all-use coverage, prime path coverage, and mutation coverage, in terms of effectiveness and efficiency. The effectiveness was measured by the number of faults detected by test suites and the efficiency was measured by the ratio of the number of test cases over the number of found faults. Total 88 faults were seeded by hand. They manually generated all the test suites from the same collection of values to reduce the "test value noise", i.e., the noise of different values satisfying the same test requirement. They purposed achieving 100% coverage level for all the subject criteria. We used a constraint solver to achieve 100% coverage level and mutation operators to analyze the fault detection effectiveness of the model-based coverage criteria systematically and statistically.

While various coverage criteria have been investigated, most studies are based on the code level or the control-flow-graph based coverage criteria. There also have been many studies on model-based coverage criteria[10–13]. Among them, most relevant one to our research is structural test coverage criteria for Lustre models. Lakehal and Parissis [13] defined structural test coverage criteria for Lustre which is a synchronous data-flow modeling language. They compared the proposed test coverage criteria with other existing test coverage criteria qualitatively, but didn't provide quantitative evaluation for the coverage criteria.

To the best of our knowledge, there has been no previous work for evaluation of FBD model-based coverage criteria using mutation analysis.

3 FBD Model-Based Coverage Criteria

We summarize basic concepts and definitions for the FBD model test coverage criteria pertinent to this work. Formal definitions and detailed descriptions for the FBD test coverage criteria are presented in [4].

3.1 Function Block Diagram and Basic Concepts

The main characteristics of PLC models are indefinite and cyclic execution. A PLC program reads inputs, computes internal values, and generates outputs in each scan cycle [14]. Because of such characteristics, PLC is suitable for continuous system environments.

FBD is one of standard PLC modeling languages. FBD is featured in its graphical notations and expressiveness for high degree of data-flows among the components. Figure 1 shows an FBD model example. FBD models consist of data flow signals and processing elements. An FBD model is considered a directed graph which consists of multiple inputs and outputs. In Figure 1, the FBD model consists of 6 blocks and 13 edges. It has 7 input edges and one output edge.

A *d-path* in an FBD model is defined as a finite sequence $\langle e_1, e_2, ..., e_n \rangle$ of edges. For example, one of d-path with length 5 in Figure 1 is $p_5 = \langle f\_X$, $GE3$,

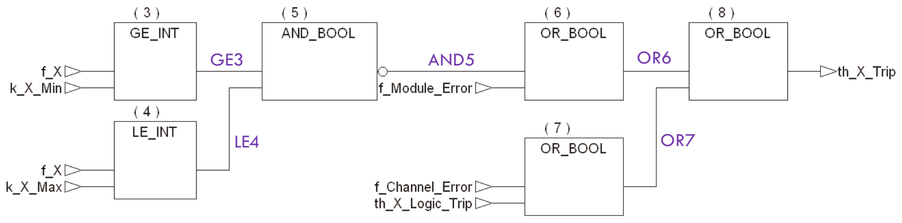


Fig. 1. A small FBD model for calculating *th_X_trip*

$AND5$, $OR6$, $th\_X\_Trip\rangle$. The number of d-paths in an FBD model is finite because FBD models have no internal feedback loops.

A *d-path condition* (DPC) is defined for each d-path. DPC for a d-path p is a condition under which the input of p influences the output of p along the d-path. If DPC(p) is *true* by an input test data vector $t = (v_1, v_2, ..., v_n)$ for inputs $i_1, i_2, ..., i_n$ (i.e., $|DPC(p)|_t = true$), it means that t is an effective input to make the input edge of p influence the output edge of p. In other words, an input value vector which makes DPC(p) true is considered to cover the d-path p. We can deal with only "meaningful" input values, which influence to the concerned output, through DPCs. DPCs are composed of conjunction of function conditions (FCs) and function block conditions (FBCs) along the corresponding d-path. DPCs are represented by logical formulas including Boolean and non-Boolean variables. For example, $DPC(p_5)$ is conjunction of four FCs as follows: $DPC(p_5) = DPC(\langle f\_X, GE3, AND5, OR6, th\_X\_Trip\rangle) = FC(f\_X, GE3) \wedge FC(GE3, AND5) \wedge FC(AND5, OR6) \wedge FC(OR6, th\_X\_Trip)$. Each FC or FBC in DPC formula is replaced by corresponding logical formulas until including only input or internal variables of the FBD model. The detailed information of FCs and FBCs is described in [4].

3.2 Structural Test Coverage Criteria for FBD Models

Three different coverage criteria for FBD models have been defined based on the DPC concept. They are Basic Coverage (BC), Input Condition Coverage (ICC), and Complex Condition Coverage (CCC). Let DP be the set of all d-paths from input edges to output edges.

Definition 1 (Basic Coverage). *A set of test data T satisfies the basic coverage criterion if and only if $\forall p \in DP$, $\exists t \in T$ $|DPC(p)|_t = true$.*

BC criterion requires covering every d-path at least once. All DPCs for all the d-paths are test requirements for achieving BC.

Definition 2 (Input Condition Coverage). *A set of test data T satisfies the input condition coverage criterion if and only if $\forall p \in DP$, $\exists t \in T$ $|in(p) \wedge DPC(p)|_t = true$ and $\exists t' \in T$ $|\neg in(p) \wedge DPC(p)|_{t'} = true$ where $in(p)$ is a Boolean input edge of the d-path p.*

To satisfy the ICC criterion, both *true* and $false$ values for every Boolean input edge should be considered in addition to satisfying all DPCs. When there is an output edge in a target FBD model, if α is the number of d-paths stating with a Boolean input and β is the number of d-paths starting with a non-Boolean input, then the number of test requirements for ICC is $((\alpha + \beta) + (\alpha \times 2))$ while the number of test requirements for BC is $(\alpha + \beta)$. Every test set satisfying the ICC criterion also satisfies the BC criterion, i.e., ICC *subsumes* BC.

Definition 3 (Complex Condition Coverage). *A set of test data* T *satisfies the complex condition coverage criterion if and only if* $\forall p \in DP$, $\exists t \in T \; | e_i \wedge DPC(p)|_t = true$ *and* $\exists t' \in T \; | \neg e_i \wedge DPC(p)|_{t'} = true$ *where* e_i *is a Boolean edge in the d-path* p *of length* n *and* $1 \leq i \leq n$.

The CCC criterion is strongest among the three FBD model-based test coverage criteria. This criterion requires covering not only every d-path as BC but also every Boolean edges with both *true* and $false$ values. The CCC criterion subsumes BC and ICC criteria by definition because input edges are included in all edges of FBD models.

4 Evaluation Strategy with Mutation Analysis

In order to evaluate the FBD model-based test coverage criteria, we investigate two main research questions:

- Q1: How effective is each of the three test coverage criteria in fault detection?
- Q2: What types of faults are likely to be found by the coverage criteria?

To answer these questions, we designed our experiments as described in Figure 2. We used industrial FBD models in our experiments. For each subject model, we generated a number of test suites satisfying the FBD model-based test coverage criteria using an automated tool, *FBDTester*. Jee [15] developed *FBDTester* to automatically generate test suites satisfying the FBD coverage criteria. Meanwhile, we also generated a number of mutants by applying mutation operators to the subject models. We defined FBD mutation operators and developed *FBDMutantGenerator* for generating mutants automatically. We simulated the mutants, i.e., FBD models including faults, with the test suites using *FBDMutantSimulator* which is an automatic execution tool for mutants with test suites. Finally, *FBDMutantSimulator* reported mutant kill information for each coverage criterion.

4.1 Subject Models

We conducted experiments using a preliminary version of the real-world industrial FBD models which have been developed to implement Bistable Processor (BP) of Reactor Protection System (RPS) in Korea Nuclear Instrumentation and Control System R&D Center (KNICS) project [2]. Originally 18 trip logic modules and two monitoring logic modules were implemented in the FBD models for

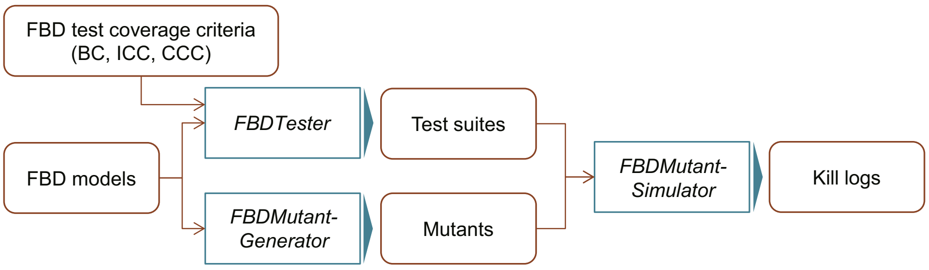


Fig. 2. Evaluation strategy overview

the BP system. Each trip logic decides whether trip (reactor stop) should occur or not by observing input signals such as pressure, temperature, and volume. Monitoring logic checks liveness of the system.

Among 18 trip modules and two monitoring modules, we classified them in five groups according to model size and structural complexity and selected five representative FBD models from each group. The selected modules are Fix-Rising (FR), Heart-Beat (HB), Manual-Rate-Calculation (MRC), Manual-Rate-Falling (MRF), and Trip-Decision (TD). The other modules are similar to the selected subject modules. For example, one of unselected module Fix-Falling is the same as Fix-Rising except using a LT block instead of a GT block.

Table 1 shows the number of blocks, inputs, outputs, and d-paths of each model. Table 1 also shows the number of feasible test requirements (TRs) with respect to BC, ICC, and CCC, for each model. In our experiments, we consider feasible test requirements only.

Table 1. Size information for five subject models

	HB	MRF	FR	MRC	TD
Blocks	38	26	26	15	7
Inputs	12	11	10	13	8
Outputs	4	4	4	2	2
d-paths	118	235	142	113	16
TRs for BC	110	215	142	113	16
TRs for ICC	110	273	182	165	32
TRs for CCC	1028	1675	958	553	124

4.2 Test Suite Generation

We generated a number of test suites with the aim of achieving 100% coverage level for each of three FBD coverage criteria. Such test suites are referred to *C-suites* for a coverage criterion *C*. For example, a test suite achieving 100% coverage for BC is called BC-suite. Coverage level is the number of satisfied test requirements over the total test requirements.

Since test requirements for FBD models were composed of complex propositional formulas, pure-random test data generation was not scalable for achieving over 95% coverage level. We generated test suites using *FBDTester* [15], an automatic test data generation tool for FBD models with respect to the BC, ICC, and CCC criteria. In *FBDTester*, an SMT (Satisfiability Modulo Theories) solver [16] is used to generate test cases since finding a test case satisfying test requirements is considered an SMT problem. *FBDTester* is designed to generate a test suite satisfying given test requirements maximally.

Preliminary studies [5, 8, 9] indicated that generally a number of randomly generated test suites are needed to gain unbiased fault detection effectiveness. Ideally, a test suite satisfying a criterion would assure a specific level of fault detection regardless of the methods used in generating test suites. However, in reality, two test suites satisfying the same coverage criterion with same coverage level may differ widely in their fault detection effectiveness. For meaningful experiments without this noise, we generated 100 independent test suites per each subject model and statistically analyzed the results.

We generated test suites using guided-random methods because, as noted above, pure-random generation method was not suitable for achieving high level of coverage. In order to introduce randomness in test suite generation and generate many test suites, we slightly modified *FBDTester*, which originally generated a test suite for a test coverage criterion, to be able to generate various test suites for the same set of test requirements. By shuffling the given test requirements, the SMT solver could generate a number of different test suites while all the generated test suites have the same coverage level for the same test coverage criterion. With this guided-random approach, we successfully generated a number of test suites satisfying 100% coverage level per each test coverage criterion.

Table 2. The average test suite size, and the number of generated mutants for the subject modules

	HB	MRF	FR	MRC	TD
BC-suite	5.22	5.04	5.00	6.00	1.00
ICC-suite	5.30	6.04	6.00	6.00	2.00
CCC-suite	8.06	13.18	9.00	11.28	2.00
Mutants	190	102	102	51	36

Table 2 summarizes the average number of the generated test cases for each subject model according to each criterion. For example, BC-suites for the FR module have about five test cases and CCC-suites for the TD module have about two test cases in average. Table 2 also shows the number of the generated mutants which will be explained in Section 4.3.

4.3 Mutant Generation

Mutation Analysis. Given a test suite S_M for a model M, let $T(M)$ be the total number of mutants for the model M and $K(S_M)$ be the number of mutants killed by S_M. Then the *mutant score* is defined as follows:

$$mutant\ score = \frac{K(S_M)}{T(M)} \times 100 \quad (1)$$

We measured mutant scores for each test suite and deduced the fault detection effectiveness of the test suite from the mutant score. Maximum mutant score is 100 which means 100% mutants are detected. The fault detection effectiveness of a coverage criterion C can be evaluated by the fault detection effectiveness of the C-*suites*.

Mutation Operators. The general principle underlying mutation analysis is that the faults used by mutation testing represent the mistakes that programmers often make [7]. Since there have been no previous studies on mutation operators for FBD models, we defined FBD mutation operators by reflecting frequently occurring FBD faults surveyed in [17].

Computational complexity is one of issues in mutation analysis. To successfully reduce the number of mutants without significant loss of the fault detection effectiveness, we adopted *selective mutation* concept. Offutt et al. [18] suggested five selective mutation operators achieving 99.5 mutation score. The five selected mutation operators were ABS(Absolute Value Insertion), AOR(Arithmetic Operator Replacement), LCR(Logical Connector Replacement), ROR(Relational Operator Replacement), and UOI(Unary Operator Insertion). These operators have been widely accepted [5, 7].

By considering FBD-specific faults and selective mutation operators, we defined five representative mutation operators for FBD as follows:

– CVR(Constant Value Replacement): replace a integer constant C with $(C-2)$, $(C-1)$, $(C+1)$, or $(C+2)$.
– IID(Inverter Insertion or Deletion): negate a boolean edge.
– ABR(Arithmetic Block Replacement): replace an arithmetic block with another block from the same class.
– CBR(Comparison Block Replacement): replace a comparison block with another block from the same class.
– LBR(Logical Block Replacement): replace a logical block with another block from the same class.

These five mutation operators are related to FBD fault classes. For example, faults on constant values are related to the CVR mutation operator. The IID operator represents faults on the inverter. A small bubble connected to the output of the AND_BOOL block in Figure 1 is an inverter. The ABR, CBR, and LBR mutation operators are related to faults on arithmetic(addition, subtraction, exponential, etc.), comparison(less than or equal to, less than, equal to, etc.), and logical(and, or, exclusive-or, etc.) blocks, respectively.

To generate mutants for a subject FBD model, each of five mutation operators was applied to each block or each edge of the model whenever possible. The number of mutants in our experiments is summarized in Table 2. We manually eliminated equivalent mutants. In each execution of the given test suites, we carefully investigated the remaining mutants and confirmed that there were no test cases to be able to kill the remaining mutants.

We reflected real FBD faults described in [17] in our definition of mutation operators for FBD models. Although we borrowed similar mutation operators for program code [18], we modified them to be applicable to FBD models, and our five mutation operators can simulate the majority of real FBD faults.

5 Analysis Results

Each of 100 test suites for BC, ICC, and CCC reports a mutant score. Table 3 shows minimum, average, and maximum mutant scores for each module and coverage criterion. The rightmost column shows minimum, average, and maximum scores for the corresponding model when considering three coverage criteria in total. Figure 3 demonstrates the mutant score distributions for each model. In Figure 3, x-axis represents the FBD coverage criteria, y-axis represents mutant scores, and a box-plot displays degree of spread and skewness in the data by using five statistics: minimum, lower quartile, median, upper quartile, and maximum. For example, the MRC module has considerable spread of mutant scores and is skewed to upper-side for the three FBD coverage criteria while the TD module has uniform distributions for the three criteria.

Table 3. Summary of mutant scores of test suites achieving each of three FBD coverage criteria

FBD	Mutant score (minimum/average/maximum)			
Module	BC	ICC	CCC	Total
MRF	68.6/84.2/93.1	71.6/75.9/89.2	74.5/91.1/97.1	68.6/83.7/97.1
FR	62.7/66.0/80.4	64.7/69.1/76.5	72.5/86.0/94.1	62.7/73.7/94.1
HB	53.6/58.7/65.8	55.3/59.0/66.8	72.6/75.6/80.0	55.3/64.5/80.0
MRC	35.3/54.7/75.6	33.3/50.4/72.5	41.2/58.5/76.5	33.3/54.5/76.5
TD	16.7/16.7/16.7	33.3/33.3/33.3	33.3/33.3/33.3	16.7/27.8/33.3
Total	16.7/56.1/93.1	33.3/57.5/89.2	33.3/68.9/97.1	16.7/60.8/97.1

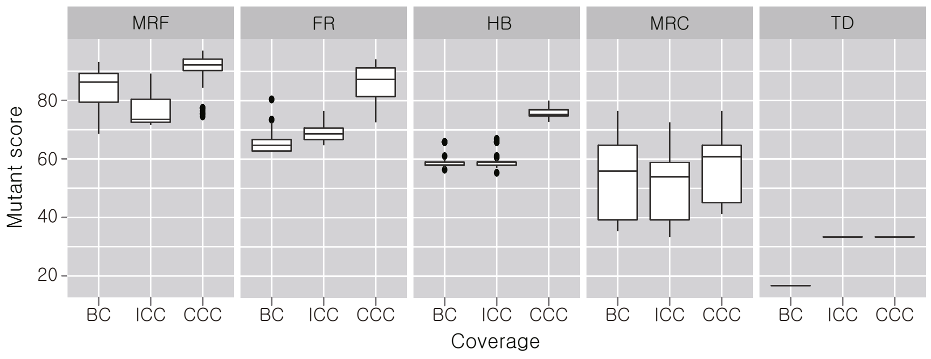


Fig. 3. Experimental results: mutant score distributions

From the experimental results, we not only confirmed expected behaviors and strengths of the FBD model-based coverage criteria, but also found weaknesses of the FBD model-based test coverage criteria in the aspect of the fault detection effectiveness. In the following subsections, we address our research questions with detailed analysis of the result.

5.1 Assessing Fault Detection Effectiveness of BC, ICC, and CCC

We investigated the fault detection effectiveness of each of the three test coverage criteria with respect to the research question Q1. Figure 4 shows the total mutant scores of four subject models[1] with respect to the three coverage criteria.

For the BC criterion, the maximum mutant score is 93.1. It means a BC-suite detects at most 93.1% of faults in an FBD model. The minimum score 35.3 indicates that at least 35.3% of faults in an FBD model are detected with the BC criterion. The lower quartile 58.9 and the upper quartile 71.6 means the spread of BC-suites in the aspect of the fault detection effectiveness. The average mutant score 64.7 indicates the average fault detection effectiveness of the BC criterion. For the ICC criterion, at most 89.2% and at least 33.3% of faults are detected by ICC-suites. Average mutant score is 68.6. The lower quartile and the upper quartile are 58.8 and 73.5, respectively. Mutant scores for the CCC criterion ranges from 41.2 to 97.1. The lower quartile and the upper quartile are 74.7 and 89.2, respectively. The average mutant score for CCC is 81.2.

In summary, at least 35.3%, 33.3%, and 41.2% of faults could be detected by BC-suites, ICC-suites, and CCC-suites, respectively. Note that a C-suite is a set of almost minimal number of test cases achieving 100% coverage level of the test coverage criterion C. With these BC-suite, ICC-suite, and CCC-suite, we can expect at most 93.1%, 89.2%, and 97.1% of fault detection, respectively.

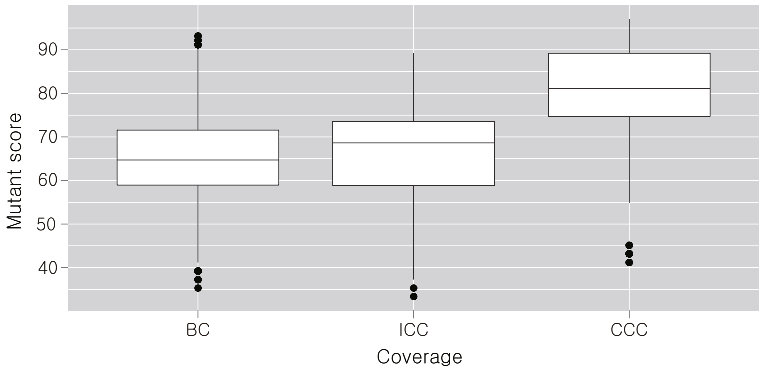


Fig. 4. Total mutant scores of subject models for each criterion

[1] Because TD reports exceptionally poor mutant scores and uniform distribution, we consider this as an outlier and discuss it in Section 5.3.

One interesting issue is that ICC-suites shows lower mutant scores than BC-suites in some cases although the ICC criterion subsumes the BC criterion. Specific FBD model structures and use of specific test case generation methods may cause this situation. We have a plan to conduct further experiments related to this issues.

5.2 Fault Detection Strength and Weakness

This section addresses research question Q2: what types of faults are likely to be found by the coverage criteria? As noted in Section 4.3, we defined five FBD mutation operators which are relevant to faults frequently occurring in FBD models. We analyzed what types of faults can or cannot be found by the test suites achieving each of criteria. We classified all the mutants into four types as follows:

- Type1: mutants neither killed by the minimum-score-suite nor the maximum-score-suite
- Type2: mutants killed by only the minimum-score-suite
- Type3: mutants killed by only the maximum-score-suite
- Type4: mutants killed by both the minimum-score-suite and the maximum-score-suite

Each type has meaningful implication. For example, type1 mutants indicate weak points of a criterion while type4 mutants indicate guaranteed fault detection effectiveness of a criterion. Mutants of type2 and type3 demonstrate possible variations in the fault detection effectiveness. In other words, mutants of type2, type3, and type4 indicate potential fault detection effectiveness.

Figure 5 represents distribution of different types of mutants killed by CCC-suites. We only show the result for CCC-suites since BC-suites and ICC-suites have similar characteristics to CCC-suites in the fault detection effectiveness for

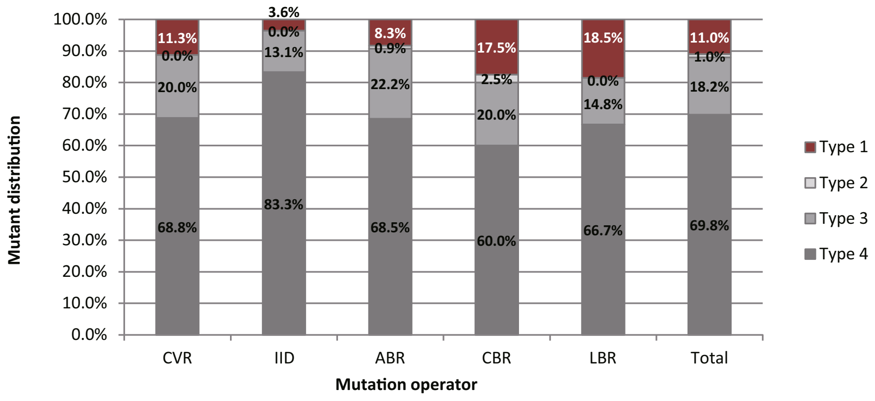


Fig. 5. Different types of killed mutants by CCC-suites

different kinds of faults. We can see that IID (negated Boolean edge) has 3.6% type1 mutants and this is the least portion among five mutant operators. It means that CCC-suites can detect 96.4% of faults like IID mutants. On the other hand, LBR (replaced logical blocks) and CBR (replaced comparison blocks) have 18.5% and 17.5% type1 mutants, respectively. These results show that CCC-suites are rather weaker in detecting LBR and CBR faults than in detecting IID faults.

The reason why IID mutants are most well detected is because the FBD coverage criteria focus on Boolean edges by definition. Note that CCC is designed to cover all Boolean edges with $true$ and $false$ values. Since FBD models are generally composed of many Boolean edges, focusing on testing the Boolean edges is worthwhile. Highly guaranteed detection of faults related to Boolean-edges is considered one of key strengths of the FBD coverage criteria.

However, LBR and CBR mutants are less covered by the FBD coverage criteria. To clearly evaluate the CCC criterion with respect to LBR and CBR mutants, we conducted an additional experiment with a unit FBD model. The unit model has one OR4 block (logical block) which has four Boolean inputs and one output edge. We denote a test case tc for an FBD model with n inputs and m outputs as $tc := \langle input_1, input_2, ..., input_n \rangle \rightarrow \langle output_1, ..., output_m \rangle$. To achieve 100% coverage level for the CCC criterion, we need just two test cases: $\langle true, true, true, true \rangle \rightarrow \langle true \rangle$ and $\langle false, false, false, false \rangle \rightarrow \langle false \rangle$. These two test cases cannot properly cover OR4 block since AND4 block shows the same behavior as the OR4 block with the two test cases. This problem occurs similarly for the comparison blocks. For example, two test cases $\langle -1, 0 \rangle \rightarrow \langle true \rangle$ and $\langle 0, 0 \rangle \rightarrow \langle false \rangle$ can be generated for the LT(Less Than) block as well as for the NE(Not Equal) block. These two test cases cannot distinguish the LT and NE blocks.

We can generalize the above problem. For a given FBD block A, let $T(A)$ be the set of all valid and executable test cases for A. Let us assume that another FBD block B is replaceable with A syntactically. If we generate a test suite for A by the elements of $T(A) \cap T(B)$, then the test suite cannot distinguish A and B, i.e., the test suite cannot detect faults such as replaced blocks. Since the replacement by a similar comparison or logical block is one of frequently occurring faults in FBD models, it would be worthwhile to generate test cases which are guaranteed to detect this kind of faults. To solve this problem, a test suite needs to have at least one test case from a set of $(T(A) \cup T(B)) - (T(A) \cap T(B))$. Based on this idea, research on defining stronger FBD model-based test coverage criteria is ongoing. Variations for non-Boolean edges as well as Boolean edges need to be considered in defining stronger FBD model-based coverage criteria to improve the fault detection effectiveness.

5.3 Discussion

Cost-Effectiveness. We investigated cost-effectiveness[2] of the three FBD model-based test coverage criteria. While CCC-suites show better mutant scores than BC-suites and ICC-suites for all the subject models, we need to consider that the

[2] We focus on the cost of test suite generation since test generation with respect to coverage criteria is the dominant time-consuming process in FBD testing.

number of test requirements (TRs) for CCC is much higher than ones for BC and ICC. We used the number of TRs as a measurement for the cost of test suite generation and the maximum mutant score as a measurement for the effectiveness of the FBD model-based coverage criteria.

Table 4 summarizes normalized cost-effectiveness values for the subject models. For example, the value 0.74 of ICC for FR means that ICC-suites kill 74% of mutants with the same cost of BC-suites. As shown in the rightmost column, in average, ICC-suites and CCC-suites can detect only 83% and 18% of faults with the same cost of BC-suite.

Table 4. Relative cost-effectiveness of subject models for each criteria

Coverage	FR	HB	MRC	MRF	TD	Average
BC	1.00	1.00	1.00	1.00	1.00	1.00
ICC	0.74	1.02	0.65	0.75	1.00	0.83
CCC	0.17	0.13	0.20	0.13	0.26	0.18

FBD Unit Size and Fault Detection Effectiveness. As shown in Table 3 and Figure 3, the TD module shows abnormally poor mutant scores. One of the reason is that the TD model is much smaller than others. In Table 1, we can find that the number of *d*-paths for TD is far fewer than other models; thus, the number of test requirements is also smaller than others. When test requirements are too simple or straightforward to satisfy, the generated test cases have lack of variations as explained in Section 5.2.

This issue is related to unit size of FBD models in testing. When a unit FBD model size is very small, it would be better to merge the small unit model with an adjacent model for testing in order to obtain better fault detection. When we conducted an additional experiment in which we merged FR and TD into a unit model FRTD, maximum mutant scores for FRTD were 74.6, 85.5 and 92.0 for BC, ICC, and CCC, respectively. Most of mutants in TD were killed when testing with FRTD.

Threats to Validity. Careful identification for threats of validity is important since there is no perfect experiment and analysis in empirical evaluations. We discuss three types of threats to validity of our experiments: *internal*, *external* and *construct* validity of our experiments.

One threat to internal validity is due to the mutation operators we use. We mentioned this issue in the last paragraph of Section 4.3, and also we have a plan to define more comprehensive mutation operators specific to FBD models. Nevertheless, the analysis result for strengths and weaknesses of the FBD coverage criteria with respect to the selected mutation operators remains still valid.

External validity is related to specific tools such as *FBDTester* and the SMT solver in test suites generation. While we generate each test case to meet the test criterion, there is different strategy to generate test cases without any knowledge of the test criterion at the first phase. In that case, the test coverage criteria is used as stopping or selecting rule after the test case generation phase. Depending on the test case generation strategy, the fault detection effectiveness of the test coverage criteria could be vary.

Construct validity is related to measurements and measured properties. The fault detection effectiveness is deduced from mutant scores of test suites. Since we purposed achieving 100% coverage level of each of coverage criteria, the fault detection effectiveness of a criterion can be inferred from a number of independent test suites satisfying the coverage criterion.

6 Conclusion

This paper reports empirical evaluations for the three FBD model-based test coverage criteria by mutation analysis in terms of the fault detection effectiveness. Our analysis results demonstrated that the FBD coverage criteria are effective to detect at most 93.1%, 89.2%, and 97.1% of faults with the BC, ICC, and CCC criterion, respectively. Especially, the FBD coverage criteria were shown to be highly effective to detect faults when the target FBD models have many Boolean edges.

We also found that the FBD coverage criteria are rather weak to discover specific types of faults such as replaced logical or relational blocks. These findings provides useful information to improve the existing test coverage criteria and testing strategies. Research on developing more sophisticated and strong test coverage criteria for FBD models is ongoing.

We defined mutation operators for FBD models and presented a mutation analysis approach to evaluate model-based test coverage criteria. Our evaluation strategy for model-based test coverage criteria can be extended to other modeling languages such as Lustre and UML.

In model-driven development environment, FBD models are transformed into executable C code automatically by a case tool. We have a plan to compare the FBD model coverage criteria with the existing code coverage criteria in our future work.

Acknowledgements. This research was partially supported by the KETEP (Korea Institute of Energy Tchology Evaluation and Planning)(KETEP-2010-T1001-01038) and the Agency for Defense Development under the grant UD100031CD.

References

1. International Electrotechnical Commission: Programmable controllers : Part 3: Programming languages. IEC, Geneva (2003)
2. Doosan Heavy Industry and Construction: KNICS-RPS-SDS231-01, Rev. 01., Software Design Specification for the Bistable Processor of the Reactor Protection System (2006)

3. USNRC: Software Unit Testing for Digital Computer Software Used in Safety Systems of Nuclear Power Plants, Regulatory Guide 1.171 (1997)
4. Jee, E., Yoo, J., Cha, S., Bae, D.: A data flow-based structural testing technique for FBD programs. Information and Software Technology 51, 1131–1139 (2009)
5. Andrews, J.H., Briand, L.C., Labiche, Y., Namin, A.S.: Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. IEEE Transactions on Software Engineering 32, 608–624 (2006)
6. King, K.N., Offutt, A.J.: A fortran language system for mutation-based software testing. Software: Practice and Experience 21, 685–718 (1991)
7. Jia, Y., Harman, M.: An Analysis and Survey of the Development of Mutation Testing. IEEE Transactions on Software Engineering 37, 649–678 (2011)
8. Hutchins, M., Foster, H., Goradia, T., Ostrand, T.: Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In: 16th International Conference on Software Engineering, pp. 191–200. IEEE Computer Society Press, Los Alamitos (1994)
9. Li, N., Praphamontripong, U., Offutt, A.J.: An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In: International Conference on Software Testing, Verification and Validation Workshops, pp. 220–229. IEEE Press, New York (2009)
10. Offutt, J., Abdurazik, A.: Generating Tests from UML Specifications. In: France, R., Rumpe, B. (eds.) UML 1999. LNCS, vol. 1723, pp. 416–429. Springer, Heidelberg (1999)
11. Andrews, A., France, R., Ghosh, S., Craig, G.: Test adequacy criteria for UML design models. Software Testing, Verification and Reliability 13, 95–127 (2003)
12. Mcquillan, J.A., Power, J.F.: A survey of UML-based coverage criteria for software testing. Department of Computer Science. NUI Maynooth, Co. Kildare, Ireland (2005)
13. Lakehal, A., Parissis, I.: Structural test coverage criteria for lustre programs. In: 10th International Workshop on Formal Methods for Industrial Critical Systems, pp. 35–43. ACM Press, New York (2005)
14. Mader, A.: A classification of PLC models and applications. In: 5th International Workshop on Discrete Event Systems –Discrete Event Systems, Analysis and Control, pp. 239–247. Springer, Heidelberg (2000)
15. Jee, E.: A Data Flow-Based Structural Testing Technique for FBD Programs. Ph.D Thesis. KAIST Press, Republic of Korea (2009)
16. Yices SMT solver, http://yices.csl.sri.com
17. Oh, Y., Yoo, J., Cha, S., Seong Son, H.: Software safety analysis of function block diagrams using fault trees. Reliability Engineering & System Safety 88, 215–228 (2005)
18. Offutt, A.J., Lee, A., Rothermel, G., Untch, R.H., Zapf, C.: An experimental determination of sufficient mutant operators. ACM Transactions on Software Engineering and Methodology 5, 99–118 (1996)

Seeing Errors: Model Driven Simulation Trace Visualization

El Arbi Aboussoror, Ileana Ober, and Iulian Ober

IRIT, Université de Toulouse, 118 Route de Narbonne
F -31062 Toulouse, France
{El-Arbi.Aboussoror,Ileana.Ober,Iulian.Ober}@irit.fr

Abstract. Powerful theoretical frameworks exist for model validation and verification, yet their use in concrete projects is limited. This is partially due to the fact that the results of model verification and simulation are difficult to exploit. This paper reports on a model driven approach that supports the user during the error diagnosis phases, by allowing customizable simulation trace visualization. Our thesis is that we can use models to significantly improve the information visualization during the diagnosis phase. This thesis is supported by Metaviz - a model-driven framework for simulation trace visualization. Metaviz uses the IFx-OMEGA model validation platform and a state-of-the-art information visualization reference model together with a well-defined development process guiding the user into building custom visualizations,essentially by defining model transformations. This approach has the potential to improve the practical usage of modeling techniques and to increase the usability and attractiveness of model validation tools.

Keywords: Software visualization, trace exploration, embedded systems, model based validation, model dynamic analysis.

1 Introduction

Important efforts were deployed by research and industry in order to develop powerful verification and validation techniques for the design models used in the early phases of development of real-time embedded systems (RTES) [7,1]. In spite of the fact that a lot of interesting results were obtained, formal verification and validation are used on a very few concrete projects. This is partially due to the fact that the results of the formal verification are difficult to exploit Our thesis is that we can use models to significantly improve the information visualization during the diagnosis phase. To support this thesis, we have built *Metaviz* - a model-driven framework for simulation trace visualization. Metaviz aims to *support the user during the error diagnosis phases*, by allowing flexible simulation trace visualization. It is built on top of IFx-OMEGA [34] a simulation and verification toolbox for UML and SysML RTES models.

The goal of the simulation step in the validation process of a System Under Diagnosis (SUD) model is either the *interactive detection of design errors*, or the *understanding of the nature of errors detected by automatic verification*. Therefore, the purpose of performing interactive simulation is *diagnosis*, which is essentially a *cognitive* task: the user has to understand the overall behaviour of the system using a scenario exploring interface, and to discover errors in the scenario.

For the type of complex RTES design models that are targeted by IFx-OMEGA, it turns out that the diagnosis generally involves examining *multiple non-contiguous steps of a scenario*, and multiple *entities in the system (blocks, ports, message queues, etc.)*. While supporting some simple forms of view customization, the traditional simulation interface cannot define visualizations computed from different steps in the simulation scenario, and thus it is hard for a user to infer the cause of an inter-process communication error from the simulation of an error scenario.

It is a commonly accepted fact that the human working memory is limited to a few items [42], while dealing with a simulation trace usually implies watching values and relationships of tens or hundreds of elements at multiple steps in the trace. Therefore, what we need is a way to boost the user perception and cognition so that it can gather the right information for finding an error pattern. This kind of problem is addressed by many works in the field of information visualization [46]. The synergy between research in information visualization and software visualization is a promising research area [23] that we exploit in our approach, by defining new visualization facilities. As Larkin and Simsons [29], we believe that using a well designed visualization framework can make the exploration of the simulation traces more effective.

This paper illustrates a new application domain of modelling techniques: the visualisation customization. We apply it to simulation trace visualisation in order to assist the error detection during model validation and verification. The work was triggered by feed-backs we received from industrial partners on the use of traditional model validation and verification frameworks.

The rest of this paper is organized as follows: section 2 overviews the simulation and diagnosis features used in validation tools and the diagnosis process using them. Section 3 presents an extension to this process that includes the creation of customized visualisations. Section 4 presents Metaviz – the model-driven implementation of our approach, that is evaluated in Section 5 using the SysML model of the Solar Generation System (SGS) of the Automated Transfer Vehicle (ATV), designed by Astrium Space Transportation. Section 6 overviews related work.

2 Model Simulation and Diagnosis: Process, Toolset, Limits

In this section we present an overview of the simulation and diagnosis features currently used in our tools, which are representative of what is available in other model simulation and validation tools. We also outline some of the limits of currently used approaches, which motivate our work.

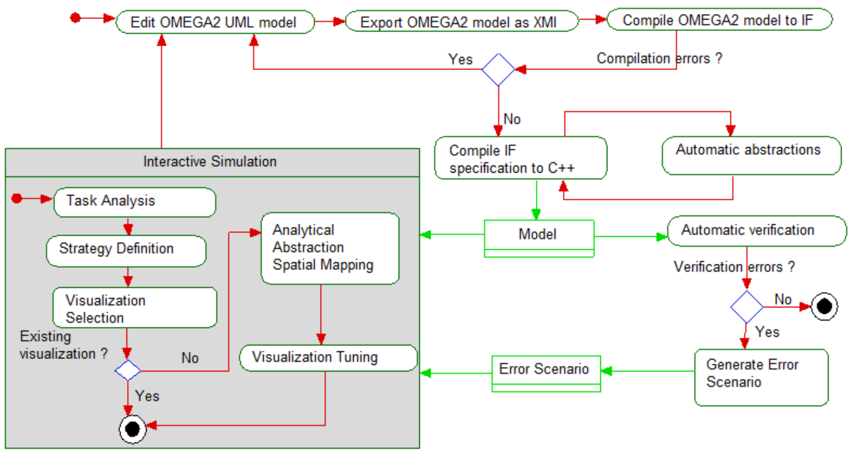


Fig. 1. The IFx validation process. The activity *Interactive Simulation* is refined in the new validation process to cover visualisation.

IFx-OMEGA[1] is a simulation and verification (*model-checking*) toolbox for UML and SysML RTES models [34]. The toolset relies on the automatic translation of models into a lower-level language (named IFx) based on asynchronous communicating extended timed automata, and on the use of the extensive toolset available for this language [12]. Figure 1 shows the validation workflow. The activity *Interactive Simulation* is empty in a classical setting, as it corresponds to our current contribution that will be detailed in Section 3.

The validation acts on a UML or SysML model, which is first translated to an IFx model, and then compiled[2] to an executable program that will be used for automatic verification and interactive simulation.

The interactive simulator offers the possibility to store simulation scenarios (as XML) and replay them later. Additional simulation scenarios are generated for each error detected by automatic verification. The simulation interface allows the user to fire any of the enabled transitions in each step of the trace and to analyse the current state of the System Under Diagnostic (SUD) (variable values, message queues, state machine configurations, etc.). The interface offers a set of customizable tree-based views of the model state and the trace steps (transitions) and a set of user controls for interacting with these views. The state views can reflect the structure of the system and its components either at the IFx level (timed automata) or in terms of the UML/SysML concepts (objects/blocks, ports, etc.).

The IFx validation approach has been applied to several industry-grade models such as Ariane-5 [36], MARS [37] and SGS [19], and has proven to be very effective in discovering design issues. The issues are generally related to the

[1] http://www.irit.fr/ifx

[2] In some cases, the model can be first simplified using automatic abstraction techniques as shown in Figure 1.

distributed, concurrent and timed nature of the systems, and very often relate to undesired message processing patterns such as an unexpected message order. Nevertheless, neither IFx, nor any other simulation tool known to us, propose any kind of advanced visualization of the error scenario being played. Effective information exploration always relies on some form of *overview* of the analysed data [32], yet no tool supports this for simulation scenarios.

Another limitation of currently used model simulators is that the visualizations are ad hoc, i.e. not based on a visualization reference model [38] Consequently, any visualization customization (data, visual structures or view customization [16]) is a challenging task. Any extension of the tool, to allow new kinds of visualization needs significant coding.

Our aim is to define an approach based on a flexible reference model that will guarantee a clear separation between the *simulation trace domain* and the *visualization concerns*. This model should enable a clear building and customization process for simulation trace visualizations. The new diagnosis platform architecture should offer extensible facilities for simulation trace visualizations. In the following section we present our approach to build a new visualization facility for simulation trace visualizations. The new tool facility should be integrated in the workflow; for this purpose, we have refined the current validation process, depicted in Figure 1, by refining the process step *Interactive Simulation*.

3 Diagnosis Process Assisted by Visualizations

The IFx-OMEGA tool set is used for validating UML design models, using the validation process illustrated in Figure 1. One of the main steps of this process is the *Interactive Simulation,* where the user extracts *error scenarios* while interacting with the *Interactive Simulation Interface*. To assist the user in this task, we have refined this step with a *diagnosis process* built around visualization concepts. The goal of this diagnosis process is to detect simulation errors and give insight into their reasons. For this purpose we provide the user with enhanced visualization of the simulation traces. In this work, we do not focus on an error taxonomy, but rather on an effective framework for building a visualization tool to support a trace exploration and analysis techniques.

The description of the different visualization stages given by Ware in [43] is the starting point for refining the interactive stimulation step. Ware describes 4 steps in designing visualizations: (i) The first step is the collection and storage of the data followed by (ii) a pre-processing step that transform the raw data into understandable data. (iii) This derived data is then displayed to (iv) enable the user to perform a perceptual tasks on it. To be effective, this high-level process needs to be refined, such it has been done by Ed Chi [18]. We believe that an efficient visualization tool should be designed based on a good understanding of the end user task. Moreover the visualization design should be primarily focused on the user task to be supported [32,11,45]. In this spirit, the visualisation design we use is composed of the following steps as shown in Figure 1 in the refinement of the *Interactive Simulation* activity.

1. User **Task Analysis**: In this context, the user task is to diagnose a certain type of errors in the IF specifications, such as message processing errors. We base our user task definitions on existing task taxonomies such as [39] and [44]. In this step one should analyse why the user task cannot be satisfied using the current means. In our setting, as we have discussed in Section 2, the visualizations cannot be customised.
2. **Strategy Definition**: The goal of this activity is to improve the user performance using external cognition. We can take into account the human perception system and study how the error scenarios that we want to diagnose should be presented to the user (see Section 5).
3. **Visualization Selection** that would amplify user cognition [8] and would support user performance improvement strategy which was defined. In this step we choose a suitable technique to support visualization, taxonomies such as [17,8] can be used. The execution of the selected technique leads to an *Analytical Abstraction.* If no existing technique is found satisfactory, new visualization techniques can be defined.
4. Define the **Analytical Abstraction Spatial Mapping** and find which variables of the Analytical Abstraction to map into spatial position in the visual structure. In fact, space is perceptually dominant [30], thus we have to identify first which data variables should be mapped to a given spatial position.
5. **Visualization Tuning** step covers all the tuning such as mapping of the variables, not considered in the previous step, to other visual coding (marks, connections, temporal encoding, etc.). It also covers *user controls* to enable interaction with the produced visualization and *attention-reactive features* to better manage user attention [45] (e.g. color highlighting). Although very important in the user data manipulation process this step is not covered by the current work.

4 MetaViz: Supporting the Simulation Process

To support the extended process introduced in the previous section, we have developed a model-based software architecture framework: Metaviz.

Metaviz assists the user from choosing a visualization technique to completing the visual mapping of the analytical abstractions. We did not find any effective formalisation of the first two steps, which are by nature highly informal. Therefore, corresponding to these steps, we assist the user solely with guidelines.

In order to build the Metaviz framework we have mapped the Data State Model [25] components to MDE concepts4. We use Metaviz *metamodels* and model to model transformations to represent *Data Stages* and *Transformation Operators* respectively.

The Data State Reference Model (DSRM). Various data-oriented visualization taxonomies exist. Some of them categorise the visualization techniques based on the visualised domain data. Others, such as the one proposed by Maletic et al. [32], are task-oriented. While these categorisations give a wide and clear

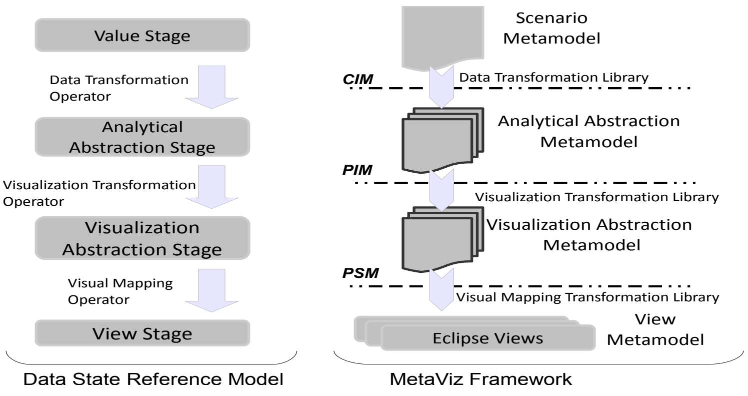

Value Stage
Data Transformation Operator
Analytical Abstraction Stage
Visualization Transformation Operator
Visualization Abstraction Stage
Visual Mapping Operator
View Stage
Data State Reference Model
Scenario Metamodel
CIM
Data Transformation Library
Analytical Abstraction Metamodel
PIM
Visualization Transformation Library
Visualization Abstraction Metamodel
PSM
Visual Mapping Transformation Library
Eclipse Views
View Metamodel
MetaViz Framework

Fig. 2. The Metaviz architecture derived from the DSRM [17]

vision of the visualization design space, they are not refined enough to promote software decoupling and reuse in the implementation process. Another taxonomy, proposed by H. Chi [17], focuses not only on the *data types* manipulated through the visualization pipeline but also on the *visual processing operators*. The Data State Reference Model proposed here is based on (i) a set of *Data Stages* that gather the data structures and (ii) visual *Transformation Operators*.

Using a data state model, the user can build the visualization pipeline in a flexible manner, as it is possible to clearly see the intermediate results of the different transformation operators and to easily plan the future stages and transformations. Moreover, in an MDA [33] spirit this separation of concern enables the reuse of stages and operators.

Metaviz components are designed following the data state model stages and operators. Figure 2 shows the Metaviz architecture and the mapping of its components to the data state model. In this mapping, we have decoupled domain and visualization assets, thus encouraging data structure and transformation reuse.

The major strength of the Metaviz architecture is the *separation of the different visualization pipeline concerns*, that is made possible by the use of the data state model. To achieve the separation of concerns, the use of a model driven approach was a perfect choice.

The *Data Stages* were implemented by Ecore metamodels [4] and the *Data Operator Transformations* were implemented using the ATL [27,2] declarative programming style to make the visualization pipeline mappings explicit in the transformation rules.

The use of Ecore and its XMI serialization facility offers an opening point towards the use of different modeling tools. The Metaviz framework is composed of several metamodels and transformations corresponding respectively to the DSRM *stages* and *operators*. The model transformations are implementing the within and non-within *stage transformation Operators*.

Value Metamodel gathers the simulation trace raw data that we want to explore by the visualisation. For this we have defined a *scenario metamodel* to *inject* [28] the XML-based simulation traces generated from the IFx automatic validation process or stored manually by the user during the interactive simulation. This metamodel gathers concepts such as fired transitions, processes, messages, etc. This metamodel can be reused or easily replaced by other descriptions [9,22] without breaking the visualization pipeline.

Value Stage transformations manipulate the scenario model elements and do not derive new data types. This transformation is implemented using OCL queries on the stage model elements. To execute those operators we have used the ATL *superimposition technique* [40] and a predicate-based query approach for filtering the relevant data.

Data Transformation are a set of ATL Transformations that transform the *Scenario* models into analytical abstractions. Any analytical technique that gives an insight into the trace data is categorized as a data transformation. For further reuse, these transformation library can be organized based on a categorisation of the existing exploration techniques, such as for instance the categories proposed by Andrienko et al. [11] : *see the whole, simplify and abstract, Look for recognisable* etc.

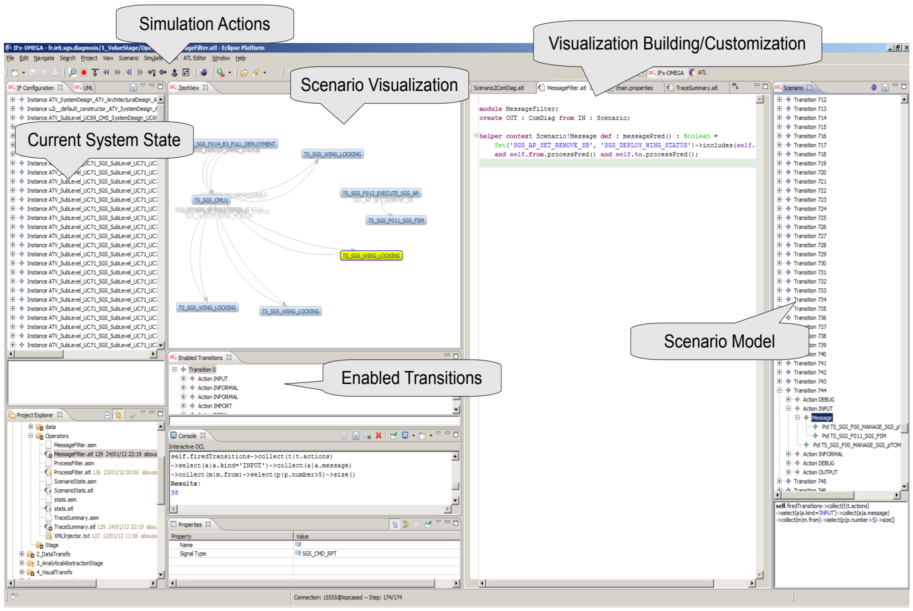
Analytical Abstraction Metamodels are defined or chosen among the existing techniques to extract meaningful information from the traces regarding relevant user task. Some widely used abstractions are communication graphs, inter-process communication patterns, event statistics. This metamodeling layer makes the capitalisation and reuse of the data analysis techniques possible. The visualization approaches often merge this layer with the visualization abstraction in the implementation phases. However we have explicitly separated this layer to enable the reuse of different analysis techniques. This layer gathers a set of ready to use trace analysis techniques. Up to now we have implemented *an inter-process communication graph* and *a trace summarizing technique.*

Visualization Transformation Operators produce visualizable content, mostly tree-based or graph-based structures, from analytical abstractions.

Visualization Abstraction Metamodels are preparing data for a set of visualization tools. It is the last step before the end-user visualization interface. A typed-node link graph is one of the mostly used abstraction in this step. A hierarchy of nodes or a more elaborated abstraction could also be used in this stage. Bull gives an overview of some widely used abstractions in [14]. The advantage of using those visual abstractions is that a set of visualization tools can share the same set of visualization abstractions.

Visual Mapping Transformation is the last transformation step to produce the visualization end product. It is usually implemented using geometric tools and layout techniques.

View Metamodels are tool specific. they gather data that is optimised for a specific visualization tool. In section 5 we will enrich this metamodeling layer with metamodels for Graphviz [6] and Zest [3]. The concept of View used in Chi's taxonomy should be refined using different models according to a tool specific criteria to enable effective implementation and reuse. Consequently we have two



```
module MessageFilter;
create OUT : ComDiag from IN : Scenario;

helper context Scenario!Message def : messagePred() : Boolean =
    Set{'SGS_AP_SET_REMOVE_SB', 'SGS_DEPLOY_WING_STATUS'}->includes(self.
    and self.from.processPred() and self.to.processPred();
```

```
self.firedTransitions->collect(t|t.actions)
->select(a|a.kind='INPUT')->collect(a|a.message)
->collect(m|m.from)->select(p|p.number>5)->size()
Results:
38
```

Property	Value
Name	
Signal Type	SGS_CMD_RPT

```
self.firedTransitions->collect(t|t.actions)
->select(a|a.kind='INPUT')->collect(a|a.message)
->collect(m|m.from)->select(p|p.number>5)->size()
```

Fig. 3. IFx-OMEGA User Interface enhanced with Metaviz features

types of metamodels in this layer: a tool-independent and a tool-specific model. A well defined visualization process and supporting framework are step forward to enable effective diagnostic of real-time and embedded system models. But they are not enough to make the users adopting the validation tool. Metaviz has to be seamlessly integrated to the IFx-OMEGA platform and the different user roles have to be clearly defined. For this purpose we have implemented the new simulation interface on top of the Eclipse platform. The figure 3 gives an overview of this new interface [3].

5 Evaluation

To illustrate our visualization framework we *build a new graph-based* visualization and we *customize* an existing one using a *trace summarizing technique*. For this, we execute the extended Diagnosis Process 3 on an industrial case study.

The Solar Generation Subsystem (SGS) [19] is a software part of the ATV (Automated Transfer Vehicle) program, a spacecraft developed by Astrium Space Transportation for ESA (European Space Agency). The ATV aims at supplying the International Space Station (ISS). The purpose of SGS is to provide functional chains to realise the solar arrays deployment and their rotation.

[3] A video demonstration is available at: http://www.irit.fr/~El-Arbi.Aboussoror/metaviz.html

5.1 Building a Visualization

The complexity of the SGS model leads to an important amount of data that needs to be analysed during the diagnostic phase. Part of the diagnostic is done by feeding back verification results at OMEGA level, as it was reported in previous work [35] still this is not enough. The kind of visualization field engineers use (e.g. message communication graph), cannot be found directly at OMEGA/UML, it has to be created by combining information from the model and from the error scenarios. A simulation scenario of the SGS model is an XML file of tens of thousands lines. The listing 1.1 give a small excerpt of this file.

```
name="u2i___default_constructor_TS_SGS_F01_EXECUTE_SGS_COMMANDS" no="0" /></by>
</IfEvent> <IfEvent kind="INFORMAL" value="--create sub-component
TS_SGS_F012_EXECUTE_SGS_AP--"> <by><pid
name="u2i___default_constructor_TS_SGS_F01_EXECUTE_SGS_COMMANDS" no="0" /></by>
</IfEvent> <IfEvent kind="IMPORT" value=""> <by><pid
name="u2i___default_constructor_TS_SGS_F01_EXECUTE_SGS_COMMANDS" no="0" /></by>
</IfEvent>
```

Listing 1.1. SGS scenario excerpt

Obviously, the user cannot answer questions about the participation of a process to a trace and the messages it exchanges with certain processes from this kind of trace. One of the information that is useful to trace during diagnostic concerns the concrete communication occurred during the execution. For this, we create a communication diagram visualization that explores the simulation scenarios of SGS.

The first step, corresponding to the *Overview* task in Shneiderman's taxonomy [39], is to *explore* the simulation traces stored by the IFx toolset user. With the classic interfaces the user can play the entire scenario step by step, but he can not see the whole scenario communication trace in a visual form. Since the human working memory restricts the amount of information one can reason about, an effective visualization should take into account this limitation and provide features that augment user cognition by external means [45]. Thus, encoding simulation trace needs to take into account the reader (OMEGA designer) and use UML-like constructs. The reader decoding tasks should not take too much effort, in order to let the user focus on understanding the system behaviour encoded in the visualization [26].

Visualizing a huge set of objects with inter-communication relationships can be performed by extracting a *communication graph* (a simplified UML communication diagram). This enables the user to grasp the process types and the exchanged messages. A communication graph is a set of *nodes* representing process types and a set of relating *edges* representing a message passing between two processes. In the SGS case study verification, the communication diagram was instrumental in reasoning about the system and in achieving the verification goals [19]. It was used to detect clusters of objects for which abstractions could be defined and used to solve the state space explosion problem. Following the diagnosis process, we have created a visualization that is precisely characterised in figure 4. This table shows the stages and operators alongside with their implementation components.

Stages	**Operators**	**Description**	**Implementation**
Value		A metamodel describing the simulation trace data	Scenario.ecore
	Value Operators	Model injection, Filtering IFx internal messages and constructors	MessageFilter.atl (superimposed to Scenario2ComDiag.atl)
	Data Transformation	Creates communication graph from the set of messages	Scenario2ComDiag.ecore
Analytical Abstraction		Inter-process communication graph	ComDiag.ecore
	Visualization Transformation	Creates a typed-node link graph from the communication graph	ComDiag2Graph.atl
Visualization Abstraction		A typed-node link graph	Graph.ecore
	Visual Mapping	Creates tool specific visualization model (e.g add layout directives)	Graph2Dot.atl, Graph2Zest.atl
View		A metamodel specific for each targeted visualization tool	Zest.ecore, Dot.ecore

Fig. 4. Characterization of the visualization technique

The visualization produced by executing this transformation chain is rendered on 2 different tools, namely Graphviz [6] and GEF/Zest [3]. An overview of the Zest view rendering is shown at the right of the figure 5

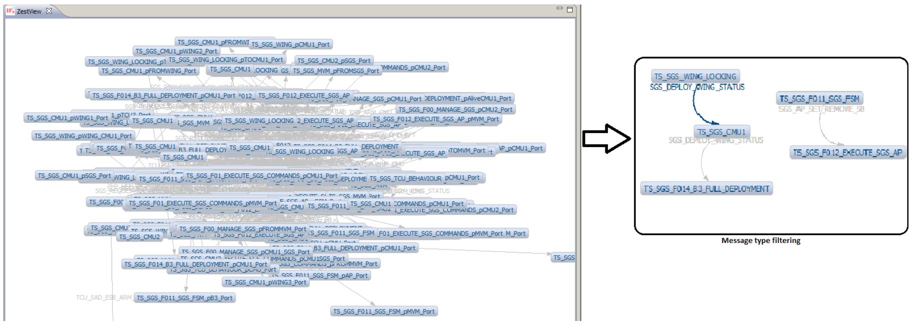


Fig. 5. Filtering a simulation trace

5.2 Customizing a Visualization

Message Type Filtering. After exploring an error scenario the user may need to understand why an error is occurring in a certain scenario. It is a *Focus* type of task. We need to construct a visualization that helps in exploring inter-process communication for a restricted set of message types. To build this new visualization we have customised the previous view by adding a *filtering* operator. This operator will filter in the scenario model the messages based on their types. Since a customization is a variation of the base behaviour we have used the *superimposition* feature of ATL to perform this filtering operations. The ATL code (a predicate on the trace messages) is given in listing 1.2. One can notice that this customization needs only few lines of code. The end product of executing the visualization pipeline is rendered on a Zest view, see figure 5 on the left. In this figure we can see the result of executing the following within-stage transformation:

```
helper context Scenario!Message def : messagePred(): Boolean=
    Set{'SGS_AP_SET_REMOVE_SB', 'SGS_DEPLOY_WING_STATUS'}
        −>includes(self.signalType)
    ;
```

Listing 1.2. ATL helper for message filtering

Trace Summarizing. To move forward in abstracting the information obtained from the error trace and illustrate the ability of Metaviz to implement more elaborated trace visualizations, we use summarising techniques such as [24]. These techniques were originally defined for the static analysing of the the call graph, but we have adapted them to build inter-process communication dynamic traces. The new customization is implemented as an ATL superimposition module. The ATL helpers in listing 1.3 were written to enable this customization.

```
helper context Scenario!Message def: messagePred(): Boolean=
    self.from.processPred() and self.to.processPred();
helper context Scenario!Pid def: processPred(): Boolean=
    self.forwardingMetric() < thisModule.threshold;
helper context Scenario!Pid def: forwardingMetric(): Real=
    let N : Integer = thisModule.allProcessSize in
      (self.fanin()/N)∗(N/(self.fanout()+1)).log()/N.log();
```

Listing 1.3. ATL helpers for trace summarizing

The *forwardingMetric* helper computes for each process *p* in the trace, a metric based on the set of processes that send messages to *p* (the *fanin* helper) and the set of processes that receive messages from *p*. For more details on the metric used in this helper the reader is referred to [24].

Using an appropriate *threshold* value, we can filter the set of processes that are performing only message forwarding. Once again, building this trace summarizing visualization, only needs a few lines of ATL code. The trace summary is rendered on a *Graphviz* view in figure 6. The main processes that collaborate in the SGS model are shown in a manner that highlights the structure, which is obviously more useful than the raw XML file 1.1. The user would confront these results to the original model and see how those processes are communicating to execute the system main functionalities.

Evaluating the Process. The use of verification and validation techniques remains marginal in the industry. Therefore, we have not been able to evaluate Metaviz by deploying it to a large set of users and questioning them on its

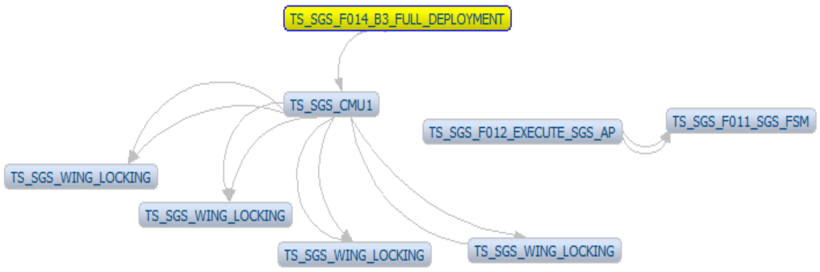


Fig. 6. Trace Summarizing

effectiveness and user-friendliness. The feed-backs we had from the industrial partners we worked with in originally validating SGS were very positive and encouraging, still we feel that we need a more objective evaluation basis.

To evaluate the Metaviz creation and customization approach we have compared it to an ad-hoc visualization implementation, as used in the validation of SGS, and that proved very useful for handling complexity. Several taxonomies [32,44] can successfully be used to evaluate the two approaches. Bull [14] offers a more complete and yet practical evaluation method. It consists of functional requirements and a set of design recommendations. Before performing the comparison it is important to notice the importance of defining the target audience [32] for the visualizations. We are tageting the IFx-OMEGA platform users. They are expected to be familiar with MDE techniques especially UML modelling with the OMEGA profile.

The figure 7 refers to the use of Metaviz versus the use of a command line ad-hoc implementation of the communication graph visualization. The comparison uses the evaluation approach above-mentioned.

One of the big drawbacks of the ad-hoc implementation is the use of UNIX command line utilities (e.g. sed) that target users are rarely familiar with. That makes the visualization building process difficult to understand or to customize. Adding automatically user controls to the generated visualization is not feasible, unless the viewer (Graphviz) is changed, but then the code is not more working.

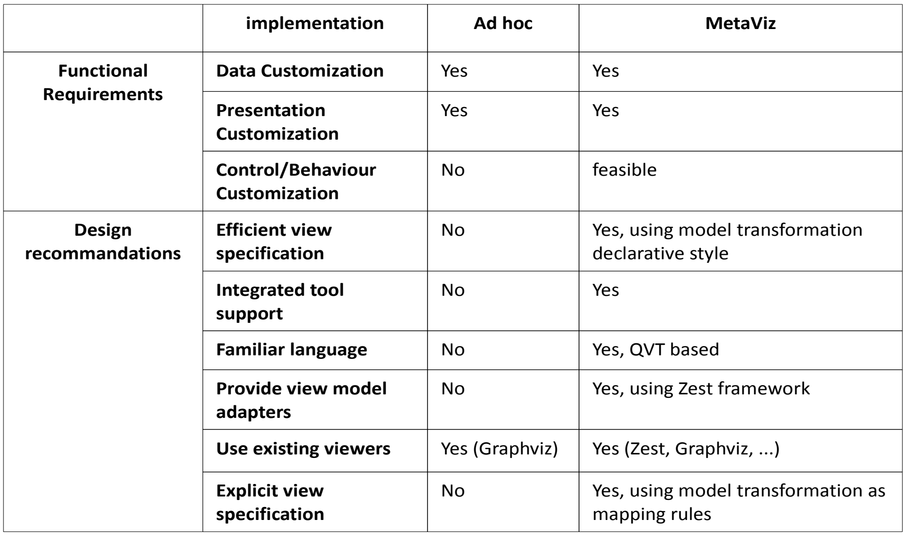
The code necessary for this ad hoc visualization is given in listing1.4

```
cat o.aut | grep −v '""' | grep −v u2i___default | grep −v u2i___init | sed 's/^.∗<<//g'
| sed 's/!.∗}{/{/g' | sed 's/>>.∗//g' | sort −u | sed 's/}//g' | awk '
BEGIN { print "digraph LTS {"; print " node [shape = circle];"; }
END { print "}"; }
/des.∗/ { next; } // { split($1,a1,"_"); split($2,a2,"_");
print "\"" a1[length(a1)] "\"" " −> " "\"" a2[length(a2)] "\"" ";" ; }
' | dot −Tepdf > net.pdf
```

Listing 1.4. Ad-hoc visualization builder

In contrast, Metaviz uses ATL, a largely used model transformation engine, and targets a model driven viewer [3]. It is also seamlessly integrated to the new

IFx-OMEGA Eclipse-based interface. The specification of the visualization pipeline is explicit thanks to the declarative style used to write the transformations.



	implementation	Ad hoc	MetaViz
Functional Requirements	Data Customization	Yes	Yes
	Presentation Customization	Yes	Yes
	Control/Behaviour Customization	No	feasible
Design recommandations	Efficient view specification	No	Yes, using model transformation declarative style
	Integrated tool support	No	Yes
	Familiar language	No	Yes, QVT based
	Provide view model adapters	No	Yes, using Zest framework
	Use existing viewers	Yes (Graphviz)	Yes (Zest, Graphviz, ...)
	Explicit view specification	No	Yes, using model transformation as mapping rules

Fig. 7. Ad hoc and Metaviz implementation comparison

6 Related Work

In this section we overview related approaches on execution trace visualisation. A lot of effort is invested in program comprehension through the dynamic analysis of *execution trace visualization*. A tremendous number of approaches focus on the visualization techniques, making the tool implementation an ad-hoc exercise. Few rely on a well defined reference model. [20,25,41,21,31].

In the Eclipse implementation of De Pauw's tool, TPTP [5], only some components are model-based (e.g. the metamodel of the trace) but the tool relies on ad-hoc architecture which make the reuse or customization of the visualization difficult. Chi has proposed the DSRM and has implemented his tool around this model but the implementation was tailored to the spreadsheet visualizations and was not intended to be extended easily by new *stages*. Walker et al. [41] use models for *stages* and a mapping language for describing explicitly some *operators*, but the mapping language does not cover the whole visualization process.Thus, most of the contribution implementations make the comprehension and the reuse of the tool challenging.

Only three approaches offer a flexible Model Driven approach for customizing the visualizations [14,31,13]. Bull in [14] has taken similar approach to build his tool around a Visualization Reference Model (VRM) [15]. This Model Driven Visualization approach is suitable for visualization that do not involve complex analytical abstractions. To implement complex transformations like execution trace summarising techniques [24], this approach needs to be refined to take into account a categorization of the operators, since the VRM is a high level data-oriented model and does not cover the whole visualization pipeline. Another similar approach is the Portolan Framework [31], but like MDV, it uses a generative approach for rendering the views, consequently a latency is introduced in the visualization prototyping loop. In contrast Metaviz promotes an interpretive approach to render the models at runtime and produce the visualizations. Finally, Buckl's approach [13] is tidily coupled to the Enterprise Architecture field and does not enable reusing the framework in visualizing simulation traces.

7 Conclusion

Previous work on verification and validation [36,37,19], performed in the context of the real-time systems specification and validation tool set IFx-OMEGA [34], as well as interactions with practitioners, convinced us about the need for meaningful, flexible and effective visualisations. In this paper we present Metaviz: our approach to support the user during the validation phase in performing model diagnosis.

Based on the Data State Reference Model [17], we extend *the diagnosis process for IFx-OMEGA*. The new diagnosis process includes the definition of *a set of simulation trace visualizations* that effectively help the user during the interactive simulation process .

Metaviz relies heavily on a model-driven implementation of the Data State Reference Model, in terms of a chain of re-usable model transformations and meta-models libraries, leading to customized visualisations. By defining new visualizations (e.g. message graph), the engineers would use these newly created visualizations instead of having to dig into large models (system level, class level, state machine level) and error scenarios etc...

We illustrate the effectiveness of Metaviz by applying it the validation of the industrial case study SGS [19]. It is impossible to perform a large scale evaluation of an approach that makes model based validation and verification more accessible, since the validation and verification are themselves marginally used by the industry. Therefore, our evaluation is done on a case study that was subject to verification and validation at Astrium, and we show how our visualisations helps the verification.

This work opens the way to several future research directions such as using Metaviz to implement new visualizations and an automatic mechanism for inter-process communication error pattern recognition using techniques such as [10]. Moreover, we intend to enhance the tool support, for instance by managing the visualization modelling artefacts (e.g. dedicated explorer for the *stages* and the *operators*) or by adding user controls to the visualizations. On another direction, we intend to enrich our work by coupling it with the goal-oriented verification engine existing in the framework.

References

1. Atego Web Site, http://www.atego.com/
2. ATL Transformation Language, http://www.eclipse.org/atl
3. Eclipse GEF Zest Framework, http://www.eclipse.org/gef/zest
4. Eclipse Modeling Framework, http://www.eclipse.org/modeling/emf
5. Eclipse Test & Performance Tools Platform, http://www.eclipse.org/tptp/
6. Graphviz, http://www.graphviz.org/
7. IFx-OMEGA, http://www.irit.fr/ifx
8. Information Visualization. In: The Human Computer Interaction Handbook: Fundamentals, Evolving Technologies and Emerging Applications, p. 509. Lawrence Erlbaum Associates (2008)
9. Alawneh, L., Hamou-Lhadj, A.: MTF: A scalable exchange format for traces of high performance computing systems. In: ICPC, pp. 181–184 (2011)
10. Alawneh, L., Hamou-Lhadj, A.: Pattern recognition techniques applied to the abstraction of traces of inter-process communication. In: CSMR, pp. 211–220 (2011)
11. Andrienko, N., Andrienko, G.: Exploratory analysis of spatial and temporal data: a systematic approach. Springer (2006)
12. Bozga, M., Graf, S., Ober, I., Ober, I., Sifakis, J.: The IF Toolset. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 237–267. Springer, Heidelberg (2004)
13. Buckl, S., Ernst, A.M., Lankes, J., Schweda, C.M., Wittenburg, A.: Generating visualizations of enterprise architectures using model transformations. In: EMISA, pp. 33–46 (2007)
14. Ian Bull, R.: Model Driven Visualization: Towards A Model Driven Engineering Approach For Information Visualization. PhD thesis, University of Victoria, BC, Canada (2008)
15. Card, S.K., Mackinlay, J.: The structure of the information visualization design space. In: Proceedings of the 1997 IEEE Symposium on Information Visualization (InfoVis 1997). IEEE Computer Society, Washington, DC (1997)
16. Card, S.K., Mackinlay, J.D., Shneiderman, B. (eds.): Readings in information visualization: using vision to think. Morgan Kaufmann Publishers Inc., San Francisco (1999)
17. Chi, E.H.: A taxonomy of visualization techniques using the data state reference model. In: Proceedings of the IEEE Symposium on Information Vizualization 2000, INFOVIS 2000, IEEE Computer Society, Washington, DC (2000)
18. Chi, E.H.H., Riedl, J.T.: An operator interaction framework for visualization systems. In: Proceedings of IEEE Symposium on Information Visualization, pp. 63–70 (October 1998)
19. Conquet, E., Dormoy, F.-X., Dragomir, I., Graf, S., Lesens, D., Nienaltowski, P., Ober, I.: Formal Model Driven Engineering for Space Onboard Software (regular paper). In: International Conference on Embedded Real Time Software and Systems (ERTS2). Society of Automobile Engineers (SAE), Janvier (2012)
20. De Pauw, W., Helm, R., Kimelman, D., Vlissides, J.: Visualizing the behavior of object-oriented systems. In: Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, OOPSLA 1993, pp. 326–337. ACM, New York (1993)
21. Favre, J.-M.: Gsee: a generic software exploration environment. In: Proceedings of the 9th International Workshop on Program Comprehension, IWPC 2001, pp. 233–244 (2001)

22. Garces, K., Deantoni, J., Mallet, F.: A model-based approach for reconciliation of polychronous execution traces. In: 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA), pp. 259–266 (2011)
23. Gotel, O., Marchese, F.T., Morris, S.J.: The potential for synergy between information visualization and software engineering visualization. In: Proceedings of the 2008 12th International Conference Information Visualisation, pp. 547–552 (2008)
24. Hamou-Lhadj, A., Lethbridge, T.: Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system. In: 14th IEEE International Conference on Program Comprehension, ICPC 2006, pp. 181–190 (2006)
25. Chi, E.H.H.: A Framework for Information Visualization Spreadsheets. PhD thesis, The University of Minnesota, USA (1999)
26. Iliinsky, N., Steele, J. (eds.): Designing Data Visualizations. O'Reilly Media, Inc. (2011)
27. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Sci. Comput. Program. 72(1-2), 31–39 (2008)
28. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a dsl for the specification of textual concrete syntaxes in model engineering. In: Proceedings of the 5th International Conference on Generative Programming and Component Engineering, GPCE 2006, pp. 249–254. ACM, New York (2006)
29. Larkin, J., Simon, H.: Why a diagram is (sometimes) worth ten thousand words. Cognitive Science (1987)
30. MacEachren, A.M.: How Maps Work - Representation, Visualization, and Design, ch. 8, p. 368. Guilford Press (2004)
31. Mahe, V., Perez, S.M., Doux, G., Brunelière, H., Cabot, J.: PORTOLAN: a Model-Driven Cartography Framework. Rapport de recherche RR-7542, INRIA (February 2011)
32. Maletic, J.I., Marcus, A., Collard, M.L.: A task oriented view of software visualization. In: Proc. 1st Int. Workshop on Visualizing Software for Understanding and Analysis (Vissoft), pp. 32–40. IEEE (2002)
33. Miller, J., Mukerji, J.: MDA guide version 1.0.1. omg/2003-06-01. Technical report, OMG (2003)
34. Ober, I., Dragomir, I.: OMEGA2: A new version of the profile and the tools. In: 15th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS), pp. 373–378 (March 2010)
35. Ober, I., Ober, I., Dragomir, I., Aboussoror, E.A.: UML/SysML semantic tunings. Innovations in Systems and Software Engineering 7(4), 257–264 (2011)
36. Ober, I., Graf, S., Lesens, D.: Modeling and Validation of a Software Architecture for the Ariane-5 Launcher. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 48–62. Springer, Heidelberg (2006)
37. Ober, I., Graf, S., Yushtein, Y., Ober, I.: Timing analysis and validation with UML: the case of the embedded mars bus manager. Journal on Innovations in Systems and Software Engineering 4(3), 301–308 (2008)
38. Robertson, P., De Ferrari, L.: Systematic Approaches to Visualization: Is a Reference Model Needed? In: Scientific Visualization: Advances and Challenges. Academic (1994)
39. Shneiderman, B.: The eyes have it: a task by data type taxonomy for information visualizations. In: Proceedings of IEEE Symposium on Visual Languages, pp. 336–343 (September 1996)

40. Wagelaar, D., Van Der Straeten, R., Deridder, D.: Module superimposition: a composition technique for rule-based model transformation languages. Software and Systems Modeling 9, 285–309 (2010)
41. Walker, R.J., Murphy, G.C., Freeman-Benson, B., Wright, D., Swanson, D., Isaak, J.: Visualizing dynamic software system information through high-level models. SIGPLAN Not. 33, 271–283 (1998)
42. Ware, C.: Information visualization: perception for design. Morgan Kaufmann Publishers Inc., San Francisco (2000)
43. Ware, C.: Information visualization: perception for design, vol. 1. Morgan Kaufmann Publishers Inc., San Francisco (2000)
44. Wiss, U., Carr, D., Jonsson, H.: Evaluating three-dimensional information visualization designs: a case study of three designs. In: Proceedings IEEE Conference on Information Visualization, pp. 137–144 (July 1998)
45. Wood, S., Cox, R., Cheng, P.: Attention design: Eight issues to consider. Computers in Human Behavior 22, 588–602 (2006)
46. Zhang, J.: The nature of external representations in problem solving. Cognitive Science 21(2), 179–217 (1997)

A Modeling Approach to Support the Similarity-Based Reuse of Configuration Data

Razieh Behjati[1,2], Tao Yue[1], and Lionel Briand[1,3]

[1] Certus Software V&V Center, Simula Research Laboratory, Norway
[2] University of Oslo, Oslo, Norway
[3] SnT Centre, University of Luxembourg, Luxembourg
`{raziehb,tao,briand}@simula.no`

Abstract. Product configuration in families of Integrated Control Systems (ICSs) involves resolving thousands of configurable parameters and is, therefore, time-consuming and error-prone. Typically, these systems consist of highly similar components that need to be configured similarly. For large-scale systems, a considerable portion of the configuration data can be reused, based on such similarities, during the configuration of each individual product. In this paper, we propose a model-based approach to automate the reuse of configuration data based on the similarities within an ICS product. Our approach enables configuration engineers to manipulate the reuse of configuration data, and ensures the consistency of the reused data. Evaluation of the approach, using a number of configured products from an industry partner, shows that more than 60% of configuration data can be automatically reused using our similarity-based approach, thereby reducing configuration effort.

Keywords: Product configuration, Internal similarities, Model-based software engineering, UML/OCL, Feature Modeling.

1 Introduction

Modern society is increasingly dependent on embedded software systems such as Integrated Control Systems (ICSs). Examples of ICSs include industrial robots, process plants, and oil and gas production platforms. Many ICS producers apply product-line engineering to develop the software embedded in their systems. They typically build a generic software, specifying a large number of interdependent configurable parameters, that need to be configured for each product according to the product's hardware architecture [6]. To configure the generic software, engineers manually assign values to tens of thousands of configurable parameters, while accounting for the constraints and dependencies between them. This makes software configuration time-consuming, error-prone, and challenging.

In the literature, the area of product configuration is still rather immature [22] and largely concentrates only on resolving high-level variabilities in feature models [19] and their extensions [10,11]. Feature models, however, are not easily amenable to capturing complex architectural variabilities and dependencies in embedded systems. Consequently, existing configuration approaches do not focus on configuration challenges in highly-configurable embedded systems, where large numbers of configurable components need to be configured and cloned.

In a previous study [6], we identified characteristics of ICS families, and their configuration challenges. Our studies show that ICSs, like many other embedded systems, bear a high degree of structural similarity within their hardware architectures to fulfill several product requirements, related for example to the environment, safety, and cost efficiency. Structural similarities in hardware affect software design and configuration, i.e., similar patterns of configuration are repeated throughout the software configuration.

In this paper, we devise a model-based approach to automatically infer configuration decisions based on the internal structural similarities of a product and previously made decisions. Our solution (1) includes a similarity modeling approach for capturing structural similarities in terms of architectural elements in an ICS family model, (2) applies feature models in practice to provide user-level representations of structural similarities so as to enable controlling the required amount of configuration reuse through feature selection, and (3) enables reducing configuration effort in large-scale, highly-configurable ICSs based on structural similarities. We build on our previous work, where we proposed a modeling methodology [5,6], called SimPL, for modeling families of ICSs, and a model-based configuration approach [4] that uses finite domains constraint solving to automate and interactively guide consistent configuration of such systems.

We motivate the work and formulate the problem in Section 2, by explaining the current practice in configuration reuse. We analyze the related work in Section 3. An overview of our model-based solution is given in Section 4. An example ICS family illustrating the main aspects of the SimPL methodology is presented in Section 5. We explain our similarity modeling approach in Section 6. The use of feature selection to control configuration reuse, and constraint propagation to automate configuration reuse are presented in Sections 7 and 8. We evaluate the effectiveness of our approach in Section 9. Finally, we conclude the paper in Section 10.

2 Configuration Reuse: Practice and Problem Definition

Figure 1 shows a simplified model of a fragment of a subsea production system produced by our industry partner. As shown in the figure, products are composed of mechanical, electrical, and software components. Our industry partner, similar to most companies producing ICSs, has a generic product that is configured to meet the needs of different customers. For example, different customers may require products with different numbers of subsea Xmas trees. A Xmas tree in a subsea production system provides mechanical, electrical, and software components for controlling and monitoring a subsea well.

Configuration in the ICSs domain is typically performed in a top-down manner where the configuration engineer starts from the higher-level components and determines the type and the number of their constituent (sub)components. Some components are invariant across different products, and some have parameters (i.e., *configurable parameters*) whose values differ from one product to another. The latter group, known as *configurable components*, may need to be further decomposed and configured. In the rest of this paper, whenever clear from the context, we use *configuration* to refer either to the configuration process or to the description of a configured artifact.

Subsea production systems, and in general ICSs, are typically large-scale systems with thousands of components and tens of thousands of configurable parameters. Usually, in these systems, a high degree of similarity is required among different configurable components to fulfill certain product requirements such as environmental, safety, or cost efficiency. For example, to reduce the costs of design and production, it may be required that all the Xmas trees in a product contain the same number and types of devices, thus requiring all the controller software units (SemApplications) to be configured similarly.

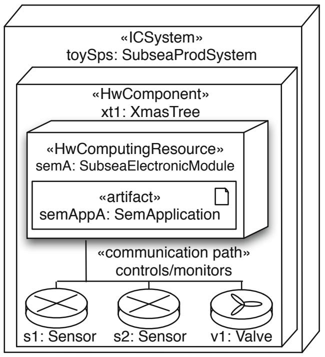


Fig. 1. Fragment of a simplified subsea production system

Similarity, in this context, is defined as a relationship between two or more configurable components. Two configurable components are similar if a subset of their configurable parameters have identical values. Such configurable components are not themselves identical, as some of their configurable parameters may have different values. The similarity that exists in such systems enables the reuse of configuration data: instead of configuring every configurable parameter separately, configurable parameters with identical values can be configured all at once. The large number of configurable parameters and the high degree of similarity lead to the potential for a high degree of configuration reuse. This can considerably reduce the *configuration effort*, which we define to be proportional to the number of manual configuration decisions.

Configuration is currently done in our industry partner using an in-house tool with primitive support for configuration reuse through a copy and paste mechanism. The existing support for the reuse of configuration data at our industry partner has the following limitations: (1) It does not provide the user with sufficient control over the configuration reuse. The user can only select one subcomponent and duplicate its whole configuration. As a result, it is sometimes necessary to modify the values of some configurable parameters in the duplicated subcomponents. (2) It does not automatically enforce the reuse of configuration data. The configuration engineer has to derive, based on her own knowledge and experience, a *configuration reuse plan* that specifies what data should be reused and how. The configuration tool cannot help following the configuration reuse plan. (3) Changes in the configuration data are not automatically propagated to the copies, therefore resulting in inconsistencies.

In our previous work [5,4], we proposed a model-based configuration approach that ensures the consistency of a, possibly partial, product during the configuration process. In this paper, we build on our previous work to propose an approach for modeling structural similarities in ICSs to automatically reuse configuration data while preventing all the above-mentioned limitations.

3 Related Work

Feature models [19,10] have been most commonly studied in the literature (e.g., [20,16,9]) for specification and model-based analysis of product families. However, few industrial applications (i.e., [13,15,23,25]) of feature models have been reported according to the findings of a preliminary review presented in [18]. Another group of approaches, which address architecture-level variability modeling (e.g., [24,27,17,21]), are studied and evaluated in our previous work [5,6]. Structural similarities within individual products, and modeling solutions to capture them are, however, missing from these approaches and applications.

Practical challenges in the configuration of highly-configurable systems have been studied, and large numbers of configurable parameters and their implicit interdependencies have been categorized as one major source of configuration errors [12]. Moreover, results from a systematic literature review [22] confirm that automation is one of the most important requirements for configuration and product derivation support. Related work on automated verification and guidance during configuration is presented in our previous work [4]. To the best of our knowledge, however, there is no work in the literature focusing on the automated reuse of configuration data, or on the similarity-based approaches to improve or automate configuration. In this paper, we address this gap by proposing a model-based approach to the automated reuse of configuration data based on structural similarities in large-scale, highly-configurable embedded systems.

4 Overview of Our Approach

Figure 2 shows an overview of our *reuse-oriented* configuration approach, which is a model-based approach to the automated reuse of configuration data based on the similarities that exist within a particular product. This approach is an extension to our previous work (the upper part in Figure 2) on automated, model-based configuration, which has two major steps. In the first step, we build a configurable and generic model for an ICS family (the Product-family modeling step). In the second step, the Guided configuration step, we interactively guide users to generate specifications of particular products complying with the generic model built in the first step.

As shown in the lower part of Figure 2, in our reuse-oriented configuration approach, we have extended both the modeling step and the configuration step of the original configuration approach. Therefore, the reuse-oriented configuration approach has four major steps. In the first step, the Product-family modeling step, a configurable and generic model of an ICS family is created by following the SimPL methodology [5,6]. In the second step, the Similarity modeling step, possible structural similarities that may exist in some particular products are modeled and organized in a *similarity model*. In the third step, the Similarity

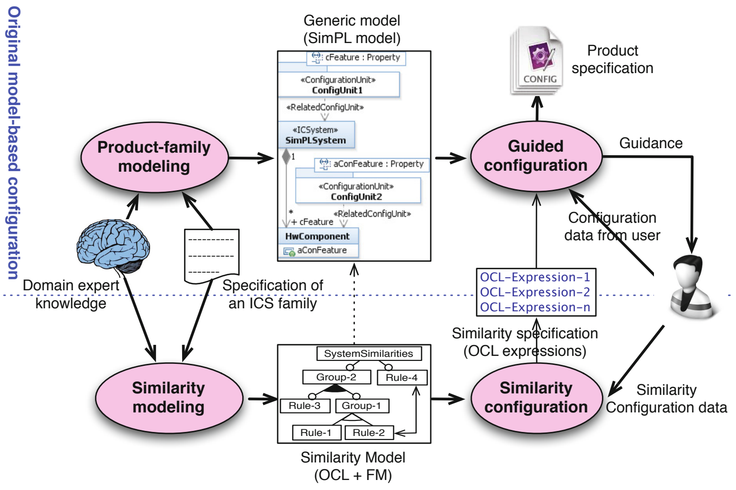


Fig. 2. An overview of our reuse-oriented configuration approach

configuration step, the similarity model is used to generate *similarity specifications* of particular products. Finally, in the Guided configuration step, we use our existing automated configuration approach [4] to interactively guide users to generate specifications of particular products that comply both with the generic SimPL model of the product family and with the similarity specifications of the products generated in the previous step.

Step 1: Product-Family Modeling

During the product-family modeling step, we provide domain experts with a modeling methodology, called SimPL [5,6], to manually create a product-family model describing an ICS family. The SimPL methodology enables the domain experts to create, from textual specifications and tacit domain knowledge, architecture models of ICS families that encompass, among other things, information about variabilities and consistency rules. We briefly describe and illustrate the SimPL methodology in Section 5. Note that our reuse-oriented extension has no impact on the product-family modeling step. This step is performed exactly as it is done in our original configuration approach.

Step 2: Similarity Modeling

During the similarity modeling step, domain experts follow the similarity modeling approach presented in this paper to manually create similarity models from textual specifications and their own domain knowledge. A similarity model expresses the structural similarities in two levels of abstraction. In the lower level of abstraction, OCL is used to express the similarity in terms of the model elements in the SimPL model of the product family. Each OCL constraint in this level specifies one *similarity rule*. In the higher level of abstraction, a feature model [19] is used to provide a user-level representation of the similarity rules. This feature model captures the variability that exists among individual products with respect to the applicability of the similarity rules. We describe and illustrate our approach to similarity modeling in Section 6.

Step 3: Similarity Configuration
During the similarity configuration step, configuration engineers use the feature models created in the previous step to select, for each product, the applicable similarity rules according to the needs of that particular product. The result of this step is a similarity specification, which is a collection of OCL constraints each representing one applicable similarity rule. Using feature models as the user-level representation of similarity rules, configuration engineers can generate similarity specifications without requiring to know OCL or the SimPL methodology. In addition, by organizing the similarity rules (that can result in the reuse of configuration data) and their variabilities in a feature model, we provide configuration engineers with a suitable mechanism to gain control over the reuse of configuration data. This way, we address the first limitation of the existing support for configuration reuse as discussed in Section 2. Similarity configuration is illustrated in Section 7.

Step 4: Guided Configuration
During the guided configuration step, configuration engineers create full or partial product specifications by resolving variabilities in a product-family model. Inputs to the guided configuration step are the generic model of the product family and the similarity specification of the product. We use these two inputs to ensure the consistency of the product specification during the entire configuration process. For this purpose, we use a finite domains constraint solver to validate each user decision, and to identify the impacts of each decision. As an impact of a user decision, the constraint solver may infer the values of one or more configurable parameters. We refer to this as the reuse of configuration data.

The main idea in this work is to use the similarity rules in the similarity specifications to trigger the inference capability of the constraint solver to automatically enforce the reuse of configuration data. Moreover, to keep the product specification consistent with respect to the similarity rules, whenever the value of a configurable parameter is changed the new value is automatically propagated to replace the related inferred values. Therefore, using our extended configuration approach, we address the second and third limitations discussed in Section 2. Note that, in this work, we have extended our original guided configuration step only by adding to it one extra input, which is the similarity specification. However, this simple extension automatically results in the automated similarity-based reuse of configuration data. This is described in details together with a brief description of our original guided configuration step in Section 8. Our original guided configuration step is described in details in [4].

5 A Subsea Product-Family Model

The SimPL methodology organizes a product-family model into two views: a *system design view*, and a *variability view*. The system design view presents both hardware and software entities of the system and their relationships using UML classes [1]. The variability view, on the other hand, captures the set of system variabilities using a collection of *configuration units*. Each configuration unit is related to exactly one class in the system design view and defines a number of *configurable features*. Each configurable feature describes a variability in the value, type, or cardinality of a property in the corresponding class. In addition to the two views described above, each SimPL model has a repository of OCL expressions [2]. These OCL expressions specify constraints among the values, types, or cardinalities of different properties of different classes. We call these OCL constraints *universal consistency rules*, as they are part of the product-family commonalities and must hold for all the products in the family.

Figure 3 shows a fragment of the SimPL model for a simplified subsea production system[1], SubseaProdSystem. In a subsea production system, the main computation resources are the Subsea Electronic Modules (SEMs), which provide electronics, execution platforms, and the software required for controlling subsea devices. SEMs and Devices are contained by XmasTrees. Devices controlled by each SEM are connected to the electronic boards of that SEM. Software deployed on a SEM, referred to as SemAPP, is responsible for controlling and monitoring the devices connected to that SEM. SemAPP is composed of a number of DeviceControllers, which is a software class responsible for communicating with, and controlling or monitoring a particular device. The system design view in Figure 3 represents the elements and the relationships discussed above.

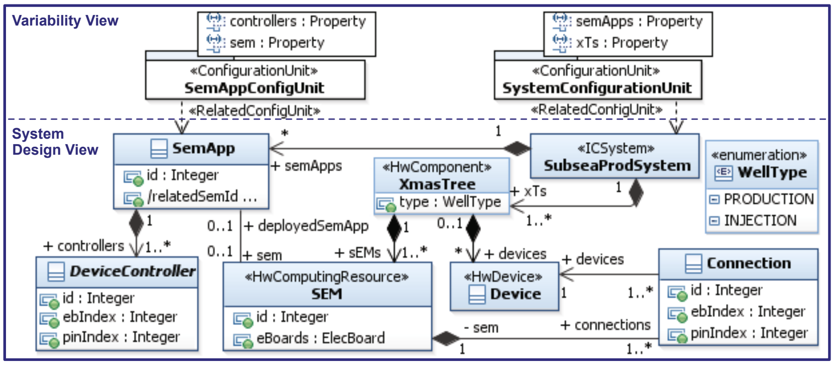


Fig. 3. A fragment of the SimPL model for the subsea production system

The variability view in the SimPL methodology is a collection of template packages, each representing one configuration unit. The upper part in Figure 3 shows a fragment of the variability view for the subsea production system. Due to the lack of space we have shown only two template packages in the figure. As shown in the figure, the package SystemConfigurationUnit represents the configuration unit related to the class SubseaProdSystem in the system design view.

[1] This example is a sanitized fragment of a subsea production case study [6].

Template parameters of this package specify the configurable features of the subsea production system, which are: the number of XmasTrees, and SEM applications (semApps).

A number of universal consistency rules are defined for the subsea production system in Figure 3. Below are OCL expressions for two of these consistency rules.

```
context Connection inv PinRange
self.pinIndex >= 0 and self.sem.eBoards->asSequence()->
     at(self.ebIndex+1).numOfPins > self.pinIndex
context Connection inv BoardIndRange
self.ebIndex >= 0 and self.ebIndex < self.sem.eBoards->size()
```

The first constraint states that the value of the pinIndex of each device-to-SEM connection must be valid, i.e., the pinIndex of a connection between a device and a SEM cannot exceed the number of pins of the electronic board through which the device is connected to its SEM. The second constraint specifies the valid range for the ebIndex of each device-to-SEM connection, i.e., the ebIndex of a connection between a device and a SEM cannot exceed the number of the electronic boards on its SEM.

Product specifications are created from family models by instantiating the classes associated to configuration units, and assigning values to the configurable parameters (i.e., instances of configurable features) of those instances.

6 Similarity Modeling

As mentioned in Section 4, in the similarity modeling step, we create similarity models that specify the similarity rules in two levels of abstraction. In this section, we first define and exemplify[2] the similarity rules. Then we explain how OCL can be used to model similarity rules in terms of the model elements in the SimPL model of the product family. Then we explain how feature models are used to provide a user-level representation of similarity rules and their variabilities. Finally, we explain the refactoring of similarity models.

6.1 Similarity Rules

A similarity rule specifies a relationship between two or more configuration unit instances within a particular product. Two configuration unit instances are similar if a subset of their configurable parameters have equal or identical values. For example, a similarity rule named XtTypeSimilarity specifies that all the Xmas trees (Figure 3) in a subsea product must be of the same type. Here, Xmas trees

[2] Examples in this section focus on describing hardware similarities, as the SimPL model in Figure 3 mostly contains hardware classes. However, in practice, similarity rules are mainly defined in terms of software classes, as they are intended to be used for reusing software configuration decisions. Note that, software similarities in a product family are, in general, very similar to its hardware similarities.

are the configuration units that are required to be similar. Types of the Xmas trees, which can either be production or injection, are the configurable parameters that are required to be identical for the similarity rule to hold.

Every similarity rule has two parts: a *scope*, and a *similarity relation*. The scope of a similarity rule determines the configuration unit instances that must be similar. For example, the scope of the similarity rule XtTypeSimilarity is the set of all Xmas trees in the product. The similarity relation in a similarity rule specifies how the similarity is achieved. It is normally composed of one or more equality relationships. Each relationship relates the values of different instances of a particular configurable feature, each belonging to a configuration unit instance in the scope of the similarity rule. For example, in XtTypeSimilarity, the similarity relation is composed of a single equality relationship that relates the values of the configurable parameter type of all the Xmas trees in the product.

It is possible to have several similarity rules with the same scope, but expressing different aspects of similarity. For example, in addition to XtTypeSimilarity, we can have another similarity rule among all the Xmas trees in the product, named XtSemNumSimilarity, expressing that all of the Xmas trees must have the same number of SEMs.

6.2 Architecture Level Modeling of Similarity Rules Using OCL

Configuration in our automated, model-based approach is performed by resolving variabilities through assigning values to configurable parameters [4]. To enable the reuse of such configuration decisions based on the similarities within a product, we express the similarity rules in terms of the configurable features and other model elements in the SimPL model of a product family. For this purpose, we use OCL, as it is the standard language for expressing constraints on the elements in UML class diagrams.

Each OCL expression is written in the context of an instance of a specific type [2]. In an OCL expression representing a similarity rule, the context must be the instance that contains all the configuration unit instances that form the scope of the similarity rule. For example, to model the similarity rule XtTypeSimilarity, we use an OCL invariant written in the context of the class SubseaProdSystem. This class is the topmost class in the SimPL model (Figure 3), and contains all the instances of XmasTree[3]. Each equality relationship in the similarity relation of a similarity rule becomes a boolean subexpression in the corresponding OCL invariant. The following is the OCL invariant expressing XtTypeSimilarity.

```
context SubseaProdSystem inv XtTypeSimilarityInv
self.xTs->forAll(x | x.type = WellType::PRODUCTION) or
self.xTs->forAll(x | x.type = WellType::INJECTION )
```

The scope of a similarity rule does not always contain all the instances of a configuration unit. In general, for modeling the scope of a similarity rule more

[3] In the SimPL methodology, each product contains only one instance of the topmost class [5,6]. In a product specification created from the SimPL model in Figure 3, the only instance of the class SubseaProdSystem contains all the XmasTree instances.

expressive OCL constructs such as *implication*- or *selection*-statements are required. The following is an example. This similarity rule specifies that all the production Xmas trees must have two SEM instances. Here, the scope of the similarity rule is the set of all Xmas trees that are of type production (specified using the selection-statement), and the number of SEMs is the configurable feature that must have the same value for all such Xmas trees.

```
context SubseaProdSystem inv ProductionXtTwoSemSimilarityInv
self.xTs->select(x | x.type = WellType::PRODUCTION)
          ->forAll(x | x.sEMs->size() = 2)
```

We use OCL *and*-statements to specify similarity relations that are composed of two or more equality relationships. SemDesignSimilarityInv is an example.

```
context SubseaProdSystem inv SemDesignSimilarityInv
SEM.allInstances()->forAll(s, t | s.eBoards->size() = t.eBoards->size())
and
SEM.allInstances()->forAll(s, t |
      s.eBoards->forAll(e1 | t.eBoards->exists(e2 | e2 = e1)))
```

6.3 User-Level Modeling of Similarity Rules Using Feature Models

As mentioned in Section 4, we use feature models [19] to provide a user-level representation of the similarity rules. We call these feature models *similarity feature models*. A similarity feature model captures the variabilities that exist among individual products with respect to the applicability of the similarity rules. A similarity feature model is part of a product-family specification, and is created only once for that product family.

Figure 4 shows a fragment of the similarity feature model for the product family shown in Figure 3. To create a similarity feature model, we follow the existing feature modeling methodologies [3] and organize features into a tree. Each leaf feature in the tree represents a similarity rule and is associated with an OCL expression. For example, XtTypeSimilarity is a leaf feature associated with the OCL invariant XtTypeSimilarityInv. Non-leaf features (e.g., XtSimilarity) are used to group related similarity rules, or other non-leaf features. In Figure 4, XtSimilarity is a non-leaf or-feature that groups two leaf features XtTypeSimilarity and XtSemNumSimilarity. An or-feature specifies that one or more of its subfeatures can be selected. Both XtTypeSimilarity and XtSemNum-Similarity are optional features and therefore introduce variabilities that should be resolved during similarity configuration.

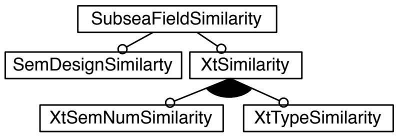


Fig. 4. A fragment of the similarity feature model for the subsea production systems family

Different types of dependencies, such as *imply* and *exclude*, may exist among similarity rules. Using feature models to organize similarity rules allows modeling these dependencies among the features representing the similarity rules. This makes OCL constraints simpler and independent from each other, thus easier to maintain. In general, all similarity rules must be consistent with the universal consistency rules in the SimPL model (This consistency can be checked, for example, using the approaches in [8] and [14]). Similarity rules are, in fact, complementary to the universal consistency rules, but must not be contradictory to them. However, similarity rules can be contradictory to each other. If two similarity rules are contradictory, an exclude or alternative relationship is necessary between the features representing them to avoid any inconsistency in the products. Figure 5 shows an example. The similarity feature model in this figure is achieved by refactoring (Section 6.4) the similarity feature model in Figure 4. AllInjection (AllProduction) is a similarity rule that specifies that all Xmas trees must be of type injection (production). The OCL constraints associated with AllInjection and AllProduction are contradictory and cannot be true simultaneously. To ensure that these two similarity rules are never selected simultaneously, the features representing them are grouped in an alternative-feature (RefactoredXtTypeSimilarity). In addition, the similarity feature model in Figure 5 shows an exclude relationship between the features AllInjection and ProductionXtTwoSemSimilarity, as selecting AllInjection makes ProductionXtTwoSemSimilarity void.

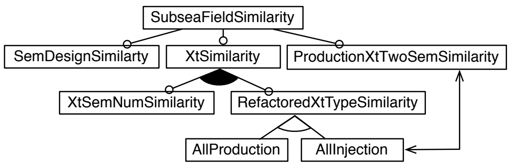


Fig. 5. Dependencies between similarity rules are modeled as dependencies between features

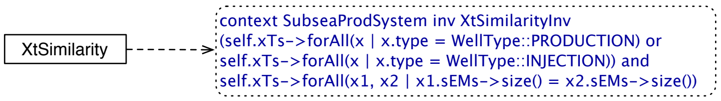
6.4 Refactoring Similarity Models

Creating similarity models is an incremental process, which may involve refactoring course-grained similarity rules into more fine-grained ones. Refactoring a similarity rule is done in both the architecture (i.e., OCL expressions) and the feature levels. Refactoring similarity models is, in particular, useful when product families evolve [7,26] and new requirements are introduced.

Consider the OCL invariant XtSimilarity in Figure 6-(a). XtSimilarity represents a similarity rule that requires all the Xmas trees in the susbea field to be of the same type (i.e., all production or all injection), and that all the Xmas trees have the same number of SEMs. This rule is associated with a single feature in the similarity feature model.

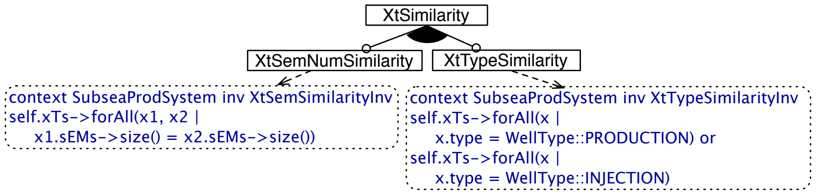
Figure 6-(b) shows the similarity feature model and OCL constraints resulting from refactoring XtSimilarity. This refactoring is done to fulfill the needs of a new product that requires all the Xmas trees in the field to have the same number of SEMs, but does not require all the Xmas trees to be of the same type. The refactoring shown in Figure 6 has decomposed XtSimilarity into two finer-grained similarity rules that can be selected independently during similarity configuration. To fulfill the needs of the new product, one must select the features XtSimilarity and XtSemNumSimilarity and leave XtTypeSimilarity unselected.

In general, if the OCL constraint expressing a similarity rule is a conjunction of subexpressions each expressing an equality relation on a different configurable feature, then it is a good modeling practice to refactor the similarity model by decomposing that similarity rule so that each subexpression becomes an independent similarity rule. To reflect this refactoring step in the similarity feature model, we make the feature corresponding to the original similarity rule a non-leaf or-feature and add to that a number of optional subfeatures each associated with one of the OCL subexpressions. In Figure 6-(b), the two OCL expressions associated with features XtTypeSimilarity and XtSemNumSimilarity are in fact the two subexpressions of the OCL constraint in Figure 6-(a).



```
context SubseaProdSystem inv XtSimilarityInv
(self.xTs->forAll(x | x.type = WellType::PRODUCTION) or
self.xTs->forAll(x | x.type = WellType::INJECTION)) and
self.xTs->forAll(x1, x2 | x1.sEMs->size() = x2.sEMs->size())
```

(a) Coarse-grained similarity rule.



```
context SubseaProdSystem inv XtSemSimilarityInv
self.xTs->forAll(x1, x2 |
    x1.sEMs->size() = x2.sEMs->size())
```

```
context SubseaProdSystem inv XtTypeSimilarityInv
self.xTs->forAll(x |
    x.type = WellType::PRODUCTION) or
self.xTs->forAll(x |
    x.type = WellType::INJECTION)
```

(b) Refactored finer-grained similarity rules.

Fig. 6. Refactoring of a similarity rule

As shown in Figure 5, XtTypeSimilarity can be refactored by decomposing its associated OCL constraint into two finer-grained OCL constraints, one (i.e., AllProduction) expressing that all the Xmas trees must be of type production, the other (i.e., AllInjection) expressing that all Xmas trees must be of type injection. This refactoring allows configuration engineers to identify the type of the Xmas trees during the similarity configuration; while, without this refactoring, configuration engineers must make this choice during the guided configuration step. Note that in both cases the total number of configuration decisions to be made are equal. Whether refactoring XtTypeSimilarity or not depends on the requirements of the product family (e.g., presence of ProductionXtTwoSemSimilarity).

7 Similarity Configuration

Optional features in the similarity feature model represent variability points that should be resolved during the similarity configuration step to generate similarity specifications. Configuration engineers resolve these variabilities by selecting

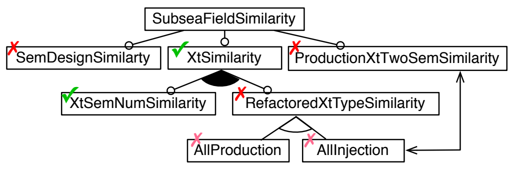


Fig. 7. Similarity feature model configured for a particular product

features in the similarity feature model according to the needs of a particular product. For example, Figure 7 shows the similarity feature model in Figure 5 configured for a product that requires all the Xmas trees to have the same number of SEMs.

Features that are selected during similarity configuration represent the similarity rules that must hold within the product under configuration. OCL constraints associated to the selected features are used to automatically generate the similarity specification of the product. For example, the similarity specification for the product mentioned above, will contain one OCL constraint, which is XtSemNumSimilarityInv that is the OCL constraint associated with XtSemNumSimilarity as shown in Figure 6.

8 Configuration Reuse through Constraint Propagation

Our original model-based configuration approach, presented in details in [4], gets as input a SimPL model, which is composed of a set of UML class diagrams and a set of OCL constraints. From these inputs, it creates a *constraints system* and uses a finite domains constraint solver to validate user decisions, to ensure the consistency of the configured product, and to automatically infer values.

Originally, OCL constraints that are fed to the configuration engine specify universal consistency rules. As mentioned in Section 4, we extend our original approach by adding to it one more input: the similarity specification of a product. In the reuse-oriented configuration approach, OCL constraints in the similarity specification are merged with the OCL constraints of the universal consistency rules, and are used by the configuration engine to create the constraints system.

Bringing the similarity rules – which express equality relationships among configurable parameters – in the constraints system forces the configuration engine to infer new values whenever a value is assigned to a configurable parameter involved in a similarity rule. For example, as a result of selecting XtTypeSimilarity, when the configuration engineer sets the type of one Xmas tree to production, the type of all other Xmas trees will be automatically set to production.

In general, OCL constraints representing similarity rules are expected to result in high numbers of inferences and a high ratio of reuse of configuration data. Using the similarity feature model and by configuring it (through selecting features), configuration engineers can control the degree of configuration reuse for each product. Note that some of the universal consistency rules may, as well, result in the reuse of configuration data. Table 1 compares universal consistency rules and similarity rules.

Table 1. A comparison between universal consistency rules and similarity rules

	Applies to	**Modeled in**	**Specifies**	**Impact on reuse**
Universal consistency rule	All products	OCL	All types of relationships	May result in reuse
Similarity rule	A subset of products	OCL	Equality relationships	Results in reuse if selected

In addition to inferring values and reusing configuration decisions, using similarity rules, value changes will be automatically propagated into similar parts of the configuration. This allows keeping the configuration consistent after changing the value of a configurable parameter and without requiring extra effort. For example, as a result of selecting XtSemNumSimilarity, whenever the configuration engineer adds a new SEM to one of the Xmas trees (i.e., changes the number of SEMs in the Xmas tree) the inference engine automatically adds a new SEM to all other Xmas trees in the field.

9 Evaluation

To empirically evaluate our approach, we investigated two complete subsea products of our industry partner. These products, detailed in Table 2, are representative considering their size, types of components, and similarity specifications.

Table 2. An overview of the two investigated products

*	# XmasTrees	# SEMs	# Devices	# Configurable parameters **
Product_1	9	18 (9 twin SEMs)	2360	29796
Product_2	14	28 (14 twin SEMs)	5072	56124

* The two products are very dissimilar with respect to their internal similarities and each represent one of two main types of subsea fields (scattered and clustered subsea fields).
** Total number of configurable parameters that need to be configured to create the software specification for the product.

Similarity Modeling. Generic software of the product family investigated in this case study contains 36 configuration units, which in total have 264 configurable features. To create a similarity feature model, we thoroughly studied both products and identified the similarities within each product. The resulting similarity feature model is a tree of depth four, with a total of 200 features, including 81 leaf features representing the similarity rules. These similarity rules have, in total, 423 equality relations that are defined in terms of classes and configurable features in the generic software model.

Similarity-Based Reuse. To create software products, we started by selecting the required similarity rules using the similarity feature model. The total number of selected similarity rules, and equality relations are reported, for each product, in Table 3. Among these similarity rules 12 are common between the two products, resulting in 110 equality relations in common. This relatively low number of common similarity rules reflects the fact that the chosen products are very dissimilar with respect to their internal similarities.

Table 3. Summary of similarity rules, and automated reuse in the two products

	# Similarity rules	# Eq. Relation	# Auto. decisions	Reuse rate
Product_1	52	263	19289	0.647
Product_2	41	270	46801	0.834

To identify the effectiveness of our approach, we introduce a measure called *reuse rate*, which provides an insight into the percentage of the decisions that can be automatically inferred based on the applied similarity rules and the previously provided configuration decisions. The fourth column in Table 3 gives, for each product, the number of such decisions. Reuse rate, for each product, is calculated by dividing the number of automated decisions by the number of configurable parameters (last column in Table 2). As shown in the fifth column in Table 3, reuse rates for product_1 and product_2 are 0.647 and 0.834, respectively. It means that, for example in product_2, 83.4% of configuration decisions can be automatically made by the configuration tool using the similarity rules, and the user has to manually configure only 16.6% of the parameters. Given the very large number of configurable parameters, this result is of practical significance. In particular, assuming automated configuration decisions have similar complexity to manual ones, our results show that such an automation can save more than 60% of the configuration effort in large-scale systems. Note that the 60% gain is calculated with respect to cases where no support for reuse is provided, not compared to the current situation at our industry partner where primitive support for reuse is provided through copy-and-paste mechanism.

Discussion. Modeling, in general, is manual and time consuming. This applies to our similarity modeling approach too. However, the effort that is put into creating similarity models is paid back because, (1) only one similarity model is created for each product family and is used during the configuration of all products, and (2) as our evaluation shows, a great portion of the configuration data can be automatically derived using similarity models, reducing the configuration effort. When the number of configurable parameters is very large–often in the thousands, as in many ICSs, the benefit of such similarity models can be substantial. This has shown to be clearly the case in our industrial case studies.

Hardware similarities that are the basis for automated reuse in our approach are present in many embedded software systems as well as distributed networked systems. Therefore, we expect our results to generalize to those domains, as well as to other ICSs with highly-symmetric hardware architectures.

10 Conclusion

This paper focuses on the automated similarity-based reuse of configuration data in families of integrated control systems (ICS). Individual ICS products, like many other embedded software systems, usually bear a high degree of similarity within their hardware structures, which results in internal similarities within their software configurations. In this paper, we propose an approach to model such internal similarities. As opposed to the commonalities in a product family that capture similarities among different products, internal similarities capture similarities among different parts of an individual product. In our similarity modeling approach, to enable automated reuse, we model internal similarities in terms of the elements in the generic model of the product family as a set of similarity rules using OCL. We use feature models to provide a user-level representation of similarity rules and the variabilities they introduce. We evaluated the effectiveness of our approach using two product configurations from our industry partner. Our results show that an automated similarity-based approach to configuration reuse can save more than 60% of configuration decisions, and consequently, can reduce configuration effort. In future, we will conduct experiments with human subjects, to further evaluate the applicability of our approach.

Acknowledgments. This work was mostly supported by a grant from Det Norske Veritas (DNV). L. Briand was supported by a FNR PEARL grant. We are grateful to FMC Technologies Inc. for their help on performing the industrial case study.

References

1. UML Superstructure Specification, v2.3 (May 2010)
2. Object Constraint Language (2012), http://www.omg.org/spec/OCL/2.2/
3. Batory, D.S.: Feature Models, Grammars, and Propositional Formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
4. Behjati, R., Nejati, S., Yue, T., Gotlieb, A., Briand, L.: Model-Based Automated and Guided Configuration of Embedded Software Systems. In: Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., Kolovos, D. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 226–243. Springer, Heidelberg (2012)
5. Behjati, R., Yue, T., Briand, L., Selic, B.: SimPL: a product-line modeling methodology for families of integrated control systems. Submitted to Information and Software Technology Journal (2011)
6. Behjati, R., Yue, T., Briand, L., Selic, B.: SimPL: a product-line modeling methodology for families of integrated control systems, Tech. Report 2011-14, Simula Research Lab (2011), http://simula.no/publications/Simula.simula.746
7. Bosch, J.: Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization. In: Chastek, G.J. (ed.) SPLC 2002. LNCS, vol. 2379, pp. 257–271. Springer, Heidelberg (2002)
8. Cabot, J., Clarisó, R., Riera, D.: Verification of uml/ocl class diagrams using constraint programming, pp. 73–80. IEEE Computer Society, Washington, DC (2008)
9. Czarnecki, K., Antkiewicz, M.: Mapping Features to Models: A Template Approach Based on Superimposed Variants. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 422–437. Springer, Heidelberg (2005)
10. Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. In: Software Process: Improvement and Practice (2005)
11. Czarnecki, K., Helsen, S., Eisenecker, U.W.: Staged configuration through specialization and multilevel configuration of feature models. In: Software Process: Improvement and Practice (2005)
12. Deelstra, S., Sinnema, M., Bosch, J.: Product derivation in software product families: a case study 74, 173–194 (January 2005)
13. Dordowsky, F., Hipp, W.: Adopting software product line principles to manage software variants in a complex avionics system. In: Proceedings of the 13th International Software Product Line Conference, SPLC 2009, pp. 265–274. Carnegie Mellon University, Pittsburgh (2009)

14. Egyed, A.: Instant consistency checking for the uml. In: Proceedings of the 28th International Conference on Software Engineering, ICSE 2006, pp. 381–390. ACM, New York (2006)
15. Gillan, C., Kilpatrick, P., Spence, I.T.A., Brown, T.J., Bashroush, R., Gawley, R.: Challenges in the application of feature modelling in fixed line telecommunications. In: VaMoS, pp. 141–148 (2007)
16. Gomaa, H., Shin, M.E.: Automated software product line engineering and product derivation. In: Proceedings of the 40th Annual Hawaii International Conference on System Sciences, HICSS 2007, p. 285a. IEEE Computer Society, Washington, DC (2007)
17. Haugen, Ø., Møller-Pedersen, B., Oldevik, J., Olsen, G.K., Svendsen, A.: Adding standardized variability to domain specific languages. In: SPLC, pp. 139–148 (2008)
18. Hubaux, A., Classen, A., Mendonça, M., Heymans, P.: A preliminary review on the application of feature diagrams in practice. In: VaMoS, pp. 53–59 (2010)
19. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21 (1990)
20. Kang, K.C., Kim, S., Lee, J., Kim, K., Kim, G.J., Shin, E.: Form: A feature-oriented reuse method with domain-specific reference architectures. Annals of Software Engineering 5, 143–168 (1998)
21. Morin, B., Perrouin, G., Lahire, P., Barais, O., Vanwormhoudt, G., Jézéquel, J.-M.: Weaving Variability into Domain Metamodels. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 690–705. Springer, Heidelberg (2009)
22. Rabiser, R., Grünbacher, P., Dhungana, D.: Requirements for product derivation support: Results from a systematic literature review and an expert survey. Information & Software Technology 52(3), 324–346 (2010)
23. Reiser, M.-O., Kolagari, R.T., Weber, M.: Unified feature modeling as a basis for managing complex system families. In: VaMoS, vol. 01. Lero Technical Report, pp. 79–86 (2007)
24. Santos, A.L., Koskimies, K., Lopes, A.: A model-driven approach to variability management in product-line engineering. Nordic J. of Computing 13, 196–213 (2006)
25. Steger, M., Tischer, C., Boss, B., Müller, A., Pertler, O., Stolz, W., Ferber, S.: Introducing PLA at Bosch Gasoline Systems: Experiences and Practices. In: Nord, R.L. (ed.) SPLC 2004. LNCS, vol. 3154, pp. 34–50. Springer, Heidelberg (2004)
26. Svahnberg, M., Bosch, J.: Evolution in software product lines: Two cases. Journal of Software Maintenance 11(6), 391–422 (1999)
27. Ziadi, T., Jézéquel, J.-M.: Software product line engineering with the UML: Deriving products, pp. 557–588 (2006)

Model Driven Configuration of Fault Tolerance Solutions for Component-Based Software System

Yihan Wu[1,2], Gang Huang[1,2,*], Hui Song[1,3], and Ying Zhang[1,2]

[1] Key Lab of High Confidence Software Technologies (Ministry of Education)
[2] School of Electronic Engineering & Computer Science, Peking University, China
[3] Lero: The Irish Software Engineering Research Center, Trinity College Dublin, Ireland
{wuyh10,zhangying06}@sei.pku.edu.cn, hg@pku.edu.cn,
hui.song@scss.tcd.ie

Abstract. Fault tolerance is very important for complex component-based software systems, but its configuration is complicated and challenging. In this paper, we propose a model driven approach to semi-automatic configuration of fault tolerance solutions. At design time, a set of reusable fault tolerance solutions are modeled as architecture styles, with the key properties verified by model checking. At runtime, the runtime software architecture of the target system is automatically constructed by the code generated from the given architectural meta-model. Then, the impact of each component on the system reliability is automatically analyzed to recommend which components should be considered in the fault tolerance configuration. Finally, after which components are guaranteed by what fault tolerance solution is decided by the system administration, the architecture model is automatically changed by merging with the selected fault tolerance styles and finally, these changes are automatically propagated to the target system. This approach is evaluated on Java enterprise systems.

Keywords: fault tolerance, component-based system, dynamic configuration, mode driven approach, software architecture.

1 Introduction

Fault tolerance is well studied and practiced in the past decades. For different types of systems or different sources of faults, we need different fault tolerance solutions [15]. For example, if the fault is caused by temporary race between the current re-quests, only re-issuing the requests will significantly decrease the rate of fault response. Alternatively, if the fault is caused by an accumulated reason, such as the memory leak, rebooting the system or a part of it is usually necessary. Such fault tolerance solutions consist of different mechanisms for detecting the faults, buffering the requests, rebooting the components, recovering the responses, etc.

In today's popular component-based systems, fault tolerance solutions themselves also become more componentized, that is, fault tolerance mechanisms are

[*] Corresponding author.

implemented as a set of reusable components by the component framework and can be configured to guarantee or ignore the given system components.

However, it is not easy to properly reuse the fault-tolerance solutions in complex component-based systems. The challenge is twofold. One is how to specify the reusable fault tolerance solutions on a specific platform. From the structural aspect, the specification should make clear the types of components required by the solution, the property of each component, and the relation between these components. The difficulty here is how to ensure the automated deployment, and in the meantime make the specification easy to understand. From the behavioral aspect, the specification should make clear the proper context for each solution, i.e., what type of faults the solution fits for, and the effect after deploying the solution. The difficulty here is how to classify the faults and how to verify the effect before really deploying the solution. Having the proper specification of fault-tolerant solutions, the second challenge is how to deploy them automatically. The first problem here is how to assist the system administrators choosing the part of the system to deploy the solution, and the proper solution to deploy. After choosing the solution, the remaining problem is how to automatically deploy and configure the reusable fault tolerance mechanism according to the solution.

In this paper, we present a model driven approach to specification and semi-automatic configuration of fault tolerance solutions for component-based systems, on the software architecture level. Based on our initial idea of supporting fault tolerance at software architecture level with the help of middleware [6], and an existing framework named SM@RT [2] [24] to support runtime model, we provide a systematic and automated framework with the help of runtime models, called SM@RT. The whole approach is divided into two phases. In the specification phase, the experts of the given system or platform define the fault tolerance mechanisms implemented by the system, in the form of a specific kind of components named fault tolerance facilities. Based on the components, the experts specify the reusable fault tolerance solutions as partial architectures composed by some of the existing facilities. The experts also list the fault tolerance properties satisfied by each of the solutions, as a reference indicating what kind of faults is proper to be fixed by this solution. In this phase, our framework provides the code generation support to wrap low-level fault tolerance mechanisms as reusable facilities, the meta-model to construct the partial architecture, and the model checking support to verify if the solution satisfies the declared fault tolerance properties. In the configuration phase, our framework helps the system administrators to semi-automatically deploy the proper fault tolerance solutions on the system. Specifically, our framework first reflects the system as runtime software architecture, and then uses this runtime architecture to calculate the key component that has the maximal influence to the global system reliability. With these two pieces of information as references, the administrator evaluates the type of faults, and chooses the proper solution. Finally, our framework automatically deploys the solution to the system, by merging the current architecture with the partial architecture specified by the solution, and then calculating and executing the required changes between the original and the result architecture.

The main contributions of this paper can be summarized as follows. Firstly, we analyze the component's impact on system reliability, and recommend key component(s). Secondly, we realize the model merging of runtime software architecture with fault tolerance style automatically. Thirdly, a systematic and semi-automated configuration framework is proposed, which is used to configure the fault tolerance solutions into component-based systems.

The rest of this paper is organized as follows: section 2 gives an overview of our approach and a motivating example of fault tolerance for an EJB component. Section 3 describes the concept of FTS (fault tolerance style) and the verification of FTS by model checking. Section 4 describes the details of analyzing the key component(s), selecting FTS, configuring RSA (runtime software architecture) with FTS, and propagating RSA changes to the target system. Section 5 shows how to use the approach to solve the problem in the motivating example. Section 6 shows the related work, and section 7 shows the discussion and future work on our approach.

2 Approach Overview

In this section, we first illustrate the fault-tolerance solutions on a real component-based system. Based on this example, we give a brief overview of the complete approach for modeling and configuring the fault tolerance solutions.

2.1 Illustrative Example

ECperf [12] is an EJB benchmark application, which simulates the process of manufacturing, supply chain management, and order/inventory in business problems. Create-a-New-Order is a typical scenario in ECperf, i.e. a customer lists all products, adds some to a shopping cart, and creates a new order. We use a Software Architecture (SA) model to depict the relations among EJBs in this scenario in Fig.1 (NFTUnit is none-fault-tolerant component). We assume these EJBs are black boxes. The structural model of ECperf comes from runtime information analysis, with the monitoring support provided by SM@RT.

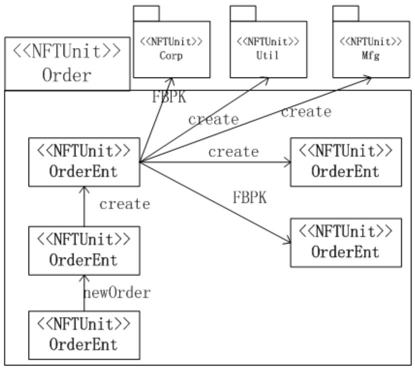


Fig. 1. The SA of ECperf in the scenario of Create-a-New-Order

ECperf cannot tolerate any faults originally, but it needs to be fault tolerable, especially for *ItemBmpEJB*, which is a frequently used bean-managed persistent EJB in the Create-a-New-Order scenario. The availability of *ItemBmpEJB* may be imperiled by database faults. These faults are permanent – they do not disappear unless the database or the connections are recovered, unlike transient faults, which may disappear in a nondeterministic manner even when no recovery operation is performed. In addition, these faults are activated only under certain circumstances like heavy-load or heavy communication traffic. So the first fault-tolerance requirement is to make *ItemBmpEJB* capable of tolerating environment-dependent and non-transient (EDNT) faults.

2.2 Approach Overview

For the above example, the Ecperf is without any fault tolerance function. Our approach for fault tolerance configuration has four steps as follows:

(1) Locating the key component(s). Component is the basic unit of component based system, and the key component is the one whose reliability matters the most to the reliability of the whole system. In this paper, we use a scenario-based reliability analysis approach to analyze the reliability of the system and locate the key component.
(2) Selecting suitable FTS. To alleviate the difficulty in the selection of the fault-tolerance mechanisms, these mechanisms are abstracted as FTS at first. Then the required fault-tolerance capabilities are specified as fault-tolerance properties, and the satisfactions of the required properties for candidate FTSs are verified by model checking [6]. The system administrator just needs to input the fault tolerance capabilities which need to be satisfied.
(3) Configuring RSA with FTS. In this step, we perform fault tolerance by model merging at the architecture level. The two models which are merged are RSA and FTS. The components in RSA which need to be configured are analyzed in step 1. And the suitable FTS is chosen in step 2.

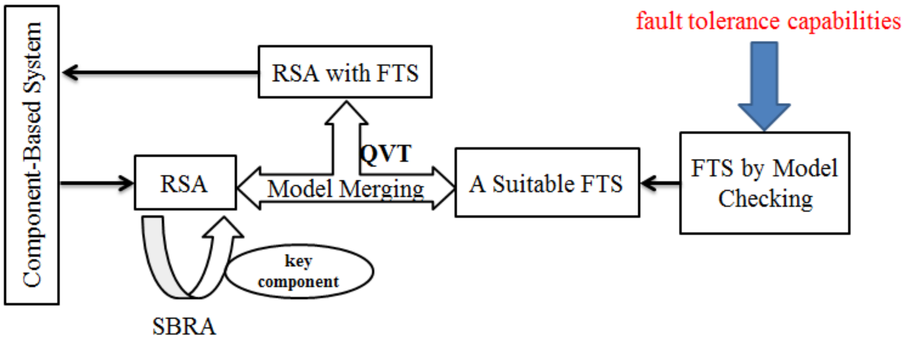


Fig. 2. Model driven configuration of fault tolerance solution

(4) Propagating the RSA changes to the target system. After step 3, we have performed fault tolerance at architecture level. For the sake of realizing fault tolerance, we use SM@RT [2][24] to propagate the change to the target system. SM@RT provides a domain-specific modeling language and a code generator to support model-based runtime system management. Figure 2 shows the whole process of our approach.

3 Solution Modeling

3.1 The Concept of Fault-Tolerance Styles

The primary activities of different fault-tolerance mechanisms are similar. They control the messages passed in, and monitor or control a component's states. An architectural style is a set of constraints on coordinated architectural elements and relationships among them. The constraints restrict the role and the feature of architectural elements and the allowed relationships among those elements in a SA that conforms to that style [21]. Entities in a fault-tolerance mechanism are modeled as components, interactions among the entities are modeled as connectors, and constraints in a mechanism are modeled as an FTS, from the point of view of architectural style. The architecture of a fault-tolerant application is a Fault-Tolerance Software Architecture (FTSA), which conforms to an FTS and tolerates a kind of fault.

3.2 Modeling Solutions as Fault-Tolerant Styles

We use a UML profile for both SA and FT [20, 25] and made necessary extensions to specify FTSs and FTSAs. There are three kinds of components in this UML profile: «NFTUnit», «FTUnit» and «FTFaci» components. «NFTUnit» components are business components without fault-tolerant capability. «FTUnit» components are business components with fault-tolerant capability either by its internal design or by applying a set of «FTFaci» components to an «NFTUnit» component. We define a stereotype «FTFaci» for well-designed and reliable components, which provide FT services for «NFTUnit» components. An «NFTUnit» component and its attached «FTFaci» components, which interact with each other in a specific manner, form a composite «FTUnit» component. There are two kinds of connectors: «FTInfo» and «FTCmd» connectors. «FTInfo» connectors are responsible for conveying a component's states to another. «FTCmd» connectors are responsible for changing an «NFTUnit» component's states.

Based on the profile, we model fault-tolerant mechanisms as FTSs. Each mechanism's structure is modeled in UML2.0 component diagram. Micro-reboot mechanism [7] is an illustrative mechanism to be modeled as FTS. A Micro-reboot style consists of four «FTFaci» components (*ExceptionCatcher*, *Reissuer*, *FTMgr*, and *BufferRedirector*) and an «FTCmd» Reboot connector for an «NFTUnit» component (Fig. 3). The *ExceptionCatcher* catches all unexpected exceptions in the «NFTUnit» component. After the caught exceptions are analyzed and the failed component is identified, the failed component is rebooted. Meanwhile, the *BufferRedirector* blocks incoming requests for the component during recovery. When the failed component is successfully recovered, the *BufferRedirector* re-issues the blocked requests and the normal process is resumed.

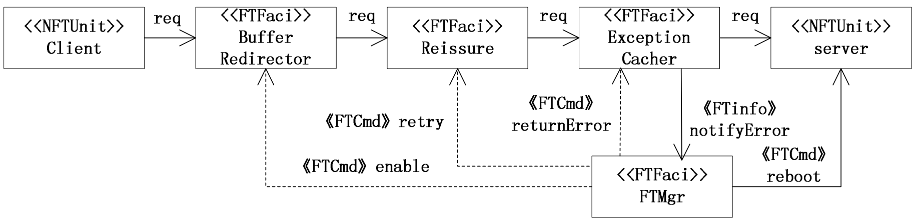


Fig. 3. The component diagram of Micro-reboot style

3.3 Validation of Solutions

In this section, we abstract both fault-tolerant capability requirements and fault assumptions on components as fault-tolerant properties. And then we translate a FTS's behavioral models in the UML sequence diagram, the properties, and the constraints into verification models, and use model checking to verify the FTS's satisfaction of the properties and the constraints.

Table 1. Fault assumption and generic fault-tolerant capabilities

Type	Property Name & Description
Fault Assumption	**Transient fault assumption (P1)**: When a component is providing services and a transient fault is activated in it then, its states will be resumed if a fault-tolerant mechanism was applied. Transient faults are nondeterministic and are also called "Heisenbugs".
	Environment-dependent and non-transient (EDNT) fault assumption (P2): When a component is providing services and an EDNT fault is activated in it then, its states will be resumed if a fault-tolerant mechanism was applied. EDNT faults are deterministic and activated only on a specific environment.
	Environment-independent and non-transient fault assumption (EINT) (P3): When a component is providing services and an EINT fault is activated in it then, its states will be resumed if a fault-tolerant mechanism was applied. EINT faults are deterministic and are independent of specific environment.
Generic Fault Tolerant Capabilities	**Fault containment (P4)**: If an error is detected in a component, other components would not be aware of the situation.
	Fault isolation (P5): When a failed component is being recovered, no new incoming requests can invoke the component.
	Fault propagation (P6): If an un-maskable fault is activated in a component, and it cannot be recovered successfully, the client, who issues the request and activates the fault, would receive an error response.
	Coordinated error recovery (P7): If a global error, which affects more than one component, happened, the error can be recovered.

Fault-Tolerant Properties

Fault assumption assumes the characters of faults in a component or an application. Only when an FTS can deal with a certain kind of fault, it is meaningful to discuss the FTS's other capabilities. Properties P1 to P3 shown in Table 1 denote three fault assumptions. These three properties form a dimension of selecting FTSs. Then fault containment, fault isolation, fault propagation, and recovery coordination are four generic fault-tolerant capabilities. They are shown in Table 1 as P4 to P7 and form another dimension of the selection.

For fear of error propagation, P4 stipulates that the source of a failure should be masked. P5 stipulates that new incoming requests cannot arrive at a failed component. Because not all errors can be masked, property P6 states that if a failed component cannot be recovered, the error should be allowed to propagate to others to trigger a global recovery process. This is important for some faults that can be tolerated by coordinated recovery among several dependent components. P7 means that both of the failed component and the components which depend on it need to be recovered.

It should be noted that the above fault-tolerant properties only cover some important and typical ones, and they are distilled from a study of FT. Other properties, such as those presented in Yuan et al.'s study [22], can also be appended to the table.

Verification of FTS
We verify the FTS by translating the intuitive behavior description into the formal specification in Promela, and then evaluate the formal on by the model checker SPIN.

We predefine a set of templates to automatically translate the extended UML2.0 sequence diagrams into Promela. The automatic translation of standard elements in UML2.0 sequence diagram has been addressed in related literature [23]. Interaction elements in UML2.0 sequence diagram, such as timeline and message dispatch, are mapped to basic block or elements in Promela, such as process (proctype) and channel (chan element). Structured control operators in UML2.0 sequence diagrams, such as conditional execution and loop execution, are mapped to control-flow constructs in Promela, such as the selection statement (if…fi) and loop statement (do…od). And the details are illustrated in our previous work [6].

3.4 Solutions Provided by Java Application Server

Using the specification mechanism, we summarized four FT solutions on a concrete system type, the Java Application Server. These solutions are widely used in the JEE systems, and the reusable facilities constituting them are able to implement based on the JEE techniques. An experiment implementation of all these facilities on the JBoss application server can be found in our previous work [9]. And the four solutions are listed as follows:

- Simple retry style: send the failed request again.
- N-copy programming style: send a request to several instances of a component, avoiding the failure of a few instances.
- Micro reboot style: initialize the failed component and recover it to original state.
- Retry block style: send the request to a component instance. If the return result is an error, modify the request, restore the environment state and resend the request.

4 Solution Configuration

4.1 Construct Runtime Software Architecture

As our fault-tolerant approach is performed at the architecture level, we need to do the following steps to get the runtime system information:

1. Define the meta architectural model of the target system.
2. Define the access model of the target system.
3. Generate the code and instantiate the RSA.

The meta-model of the target system defines the structure of RSA, including property, class, and association between classes. The access model defines the methods which used to get the runtime system information. The method *get()* is used to get the system information and *set()* is used to modify the system properties. In this paper, we use SM@RT to define the access model, generate the synchronization code, and instantiate the RSA.

4.2 Analyze Component's Reliability Impact

In this section, a scenario based reliability analysis approach is described, and then we introduce a SBRA-based algorithm to find out the key components which have the crucial influence to the system.

Scenario-Based Reliability Analysis Approach
SBRA is a reliability analysis technique for component-based software, which was proposed by Sherif Yacoub et al. in [4]. Using scenarios of component interactions, they construct a probabilistic model named Component-Dependency Graph (CDG). Based on CDG, a reliability analysis algorithm is developed to analyze the reliability of the system as a function of reliabilities of its architectural constituents.

A CDG is defined as follows:

CDG=<N,E,s,t>

$\boldsymbol{N}$={$\boldsymbol{n}$},which is a set of nodes in the graph; $\boldsymbol{E}$={$\boldsymbol{e}$},which is a set of directed edges in the graph; $\boldsymbol{s}$ and $\boldsymbol{t}$ are the start and termination nodes.
$\boldsymbol{n=<NC_i, RC_i, EC_i>}$ $n \in N$, models a component $\boldsymbol{C_i}$, $\boldsymbol{NC_i}$ is the name of component $\boldsymbol{C_i}$, $\boldsymbol{RC_i}$ is the reliability of component $\boldsymbol{C_i}$, and $\boldsymbol{EC_i}$ is the average execution time of the component $\boldsymbol{C_i}$.
$\boldsymbol{e=<T_{ij}, RT_{ij}, PT_{ij}>}$ $e \in E$, models the control flow transfer from one component to another. $\boldsymbol{T_{ij}}$ is the transition name from node $\boldsymbol{n_i}$ to $\boldsymbol{n_j}$, denoted $< \boldsymbol{n_i}, \boldsymbol{n_j} >$, $\boldsymbol{RT_{ij}}$ is the transition's reliability, and $\boldsymbol{PT_{ij}}$ is the transition's execution probability. Fig. 4 shows the CDG of a system consisting of four components.

A CDG is an input parameter of SBRA, and the other input is AE_{appl}, which is the average execution time of the application. SBRA is shown in Fig.5. The output is R_{appl}, the reliability of the application.

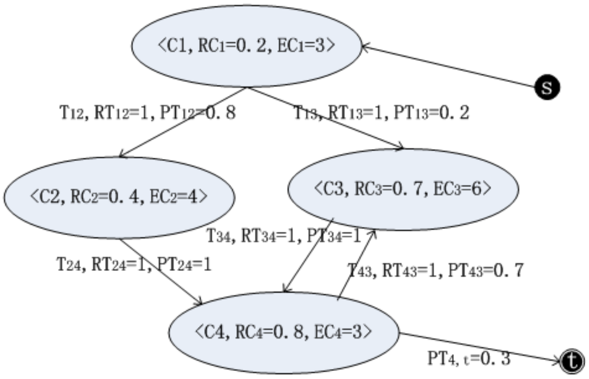


Fig. 4. A sample CDG

```
Parameters
Consumes CDG, $AE_{appl}$
Produces    $R_{appl}$
Initialization:
    $R_{appl}$=0;Time=0;$R_{temp}$=1;
Algorithm
Push tuple<$C_1$, $RC_1$, $EC_1$>, Time, $R_{temp}$
While Stack not EMPTY do
    Pop <$C_i$, $RC_i$, $EC_i$>, Time, $R_{temp}$
    if   Time>$AE_{appl}$ or $C_i$=t;
            $R_{appl}$+=$R_{temp}$;
    else
        <$C_j$,$RC_j$,$EC_j$> $\in$children($C_i$)
        push(<$C_j$,$RC_j$,$EC_j$>,Time+=$EC_i$,
        $R_{temp}$=$R_{temp}$*$RC_i$*$RT_{ij}$*$PT_{ij}$)
    end
end while
```

Fig. 5. SBRA

```
Parameters
Consumes CDG, $AE_{appl}$
Produces
   componentList <componentName, $R_{appl}$>
Initialization:
     $R_{appl}$=0;
     for each components $RC_i$=0.8;
Algorithm:
for each component <$C_i$,$RC_i$,$EC_i$>
       $RC_i$=RCi+0.2;
       $R_{appl}$=SBRA();
       componentList.add(<Ci,$R_{appl}$>);
  $RC_i$=0.8;
for <$C_i$,$R_{appl}$> in   componentList
      sorted by $R_{appl}$ decending;
```

Fig. 6. Select Key Components

SBRA-based Algorithm for Selecting Key Components

The reliability of an application is affected by several attributes in SBRA, such as the reliability and use frequency of each component, which means some components have a stronger influence on the reliability of the whole system than others. The frequency of each component depends on the scenarios. Several techniques have been proposed to estimate the reliability of software components, such as fault injection, testing, and retrospective analysis. In order to discover the key components, we value the reliability of components statically and run the SBRA-based algorithm in Fig.6.

In the above algorithm, *SBRA()* returns the reliability of the whole system, which is calculated from the CDG. We assume the reliability of each component is 0.8 at first.

Before invoking SBRA each time, the current component's reliability is improved to 1. So the value of R_{appl} is the reliability of the system after the current component's reliability improved. That means the component with the maximum value of R_{appl} is the key component.

4.3 Select Proper Solutions

Given a specific application, a set of requirements on fault-tolerant capabilities, and a set of candidate FTSs, it is critical to select the most suitable one for concerned components in the application to meet the requirements. We assist the system administrators in selecting the proper solutions by providing them the following to guidance: fault assumptions and fault-tolerant capabilities.

(a) Simple retry style

	P4	P5	P6	P7
P1	√	√	√	×
P2	×	×	×	×
P3	×	×	×	×

(b) N-copy programming style

	P4	P5	P6	P7
P1	√	×	√	×
P2	√	√	√	√
P3	×	×	×	×

(c) Micro-reboot style

	P4	P5	P6	P7
P1	√	√	√	√
P2	√	√	√	√
P3	×	×	×	×

(d) Retry blocks style

	P4	P5	P6	P7
P1	√	√	√	√
P2	×	×	×	×
P3	×	×	×	×

Fig. 7. The satisfaction of properties for Simple retry style, N-copy programming style, Micro-reboot style, and Retry blocks style. (√: preserve; ×: do not preserve). Fault assumptions form a dimension; other fault-tolerant properties form another dimension.

In model checking process, Spin simulates a FTS's behavior and traverses all its states combinations. A component's states are defined and stored in variables. These states are initialized at the beginning, and re-assigned by fault simulation function and state transit rules. When Spin control flow arrives at an assertion, it checks the truth or not of the assertion. It either confirms that the properties hold or reports that they are violated. A false assertion means the style does not preserve the property represented by the assertion, and a counter-example is provided. Otherwise, the above verification process continues. When all assertions are true, it means the FTS satisfies all the concerned properties. The result in Fig.7 shows four fault-tolerant styles' satisfaction to fault-tolerant properties.

4.4 Configure RSA with Fault-Tolerant Styles

The fault-tolerant styles define the topological structure and behavior restriction of fault-tolerant components and the business components. For the sake of implementation of fault tolerance at the architecture level, we just need to merge the RSA with a suitable FTS. And the change can be propagated to the system by the modification on middleware. This process is accomplished automatically, which avoids configuration errors and reduces the burden of system architects. The inputs of this kind of configuration are components which need to be configured, a selected FTS and the RSA of the application. The output is a fault-tolerant runtime software architecture.

The key of the configuration process is the automatic composition of RSA with FTS. In this paper, it is achieved based on Model Merging or Model Composition [18] [19]. Model Merging is a special kind of model transformation, and the function of Model Merging is merging two models MA and MB, conforming to meta-model MMA and MMB, respectively, and the result is MC, conforming to meta-model MMC [19]. MA is called Receiving Model, MB is called Merged Model, and the merging process is to merge the elements in MB into MA, and produce a Resulting Model MC. In general, there are two phases in Model Merging: comparison and merging. In the first phase, it needs to determine the match relationship automatically between the elements in Receiving Model and the elements in Merged Model. The second step, merging, adds the elements in Merged Model to the Receiving Model automatically in light of the match relation.

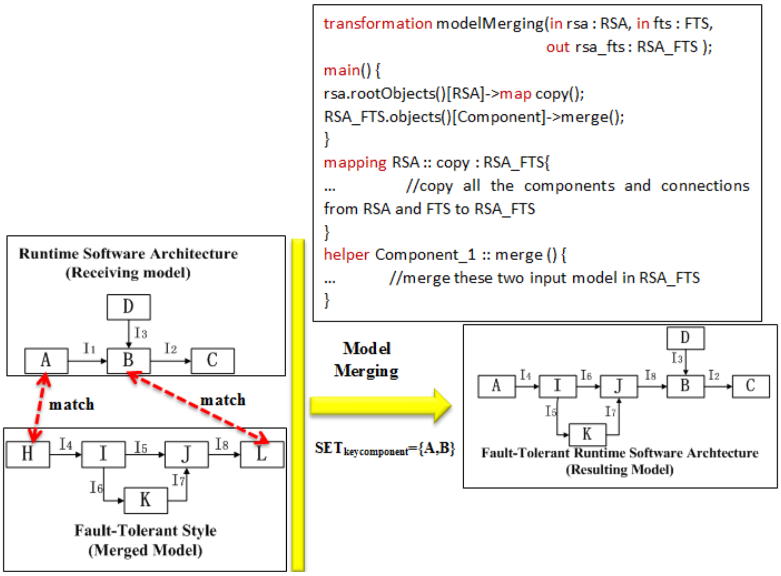

```
transformation modelMerging(in rsa : RSA, in fts : FTS,
                                        out rsa_fts : RSA_FTS );
main() {
rsa.rootObjects()[RSA]->map copy();
RSA_FTS.objects()[Component]->merge();
}
mapping RSA :: copy : RSA_FTS{
...           //copy all the components and connections
from RSA and FTS to RSA_FTS
}
helper Component_1 :: merge () {
...          //merge these two input model in RSA_FTS
}
```

Fig. 8. The merging of RSA with FTS and the QVT implementation

In this paper, the Merged Model is FTS, and the Receiving Model is RSA. The Resulting Model is fault-tolerant runtime software architecture. The process is illustrated in Fig.8, and we use QVT (Query/View/Transformation) to implement the merging, which is a standard set of languages for model transformation defined by the Object Management Group.

4.5 Propagate RSA Changes to the Target System

For the sake of getting the real system with fault tolerant, we realized the following steps: firstly, we provide fault-tolerant sandboxes for application components. And then, encapsulate the operations which are used to add (remove) a component or a connection between two components. Thirdly, we use QVT to realize model comparison [11], which is used to compare the original RSA with FTSA. And the comparison result will guide the modification of the target system.

5 Evaluations

In this section, we use the approach to configure ECperf with fault-tolerant solutions.

5.1 Select Key Components on ECperf

We obtain the CDG of ECperf via runtime information analysis, with the monitoring support provided by a reflective JEE Application Server (AS), which is shown in Fig.9. After executing the SBRA, the result is shown in Fig.10, the component name as the abscissa, and the ordinate is the reliability of the system after the reliability of the corresponding component improved.

Figure 10 shows that there are 3 key components: ItemEnt, OrderSes, and OrderEnt. The reliability of the application will be maximized, if the reliability of these three components is improved. The component ItemEnt is invoked 403 times in the process of creating a new order, while others are invoked no more than 10 times. And ItemEnt is invoked by OrderEnt, which is invoked by OrderSes. There is a strong dependence between them. So it is easy to understand the three components are key components. The QVT code of SBRA can be downloaded from our google code project[3].

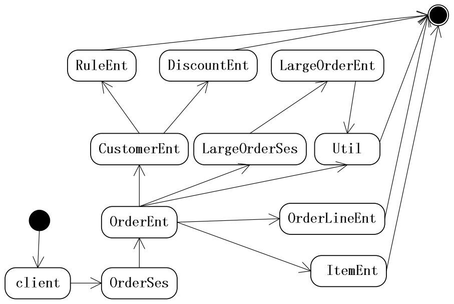


Fig. 9. The CDG of ECperf

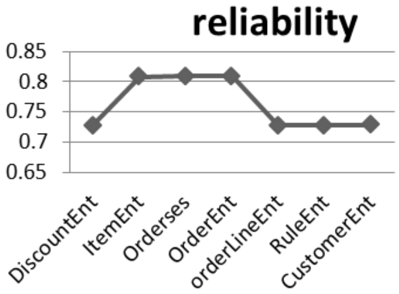


Fig. 10. The reliability of system

5.2 Select FTS for the Key Component

ECperf runs on a sequential execution environment (JEE AS), so N-Copy Programming style cannot be used because they require concurrent execution support. Then the remaining candidates include Retry Blocks style, Simple Retry style, and Microreboot style. There are no more ECperf-specific characters help to select or exclude one of the above candidates. To select a proper FTS from existing ones, we select the most suitable FTS by applying the procedure presented in 4.3.

The fault assumption of the three components is EDNT. And we need the FTS to satisfy the property P4-P7. The result is shown in Fig.7. And Micro-reboot style is the winner because it fits the EDNT fault assumption and supports all the properties, but the other three styles cannot.

5.3 Merge Ecperf with FTS

In section 5.2, the set of key components is acquired, and $SET_{key\text{-}comp}$={ItemEnt, OrderSes, OrderEnt }. In section 5.3, we find out that the micro-reboot style is suitable. Each element of the $SET_{key\text{-}comp}$ corresponds to the component "server" in FTS. As there are three components that need to be configured, the merging process has three steps. And the match relationship is shown in Fig.11.

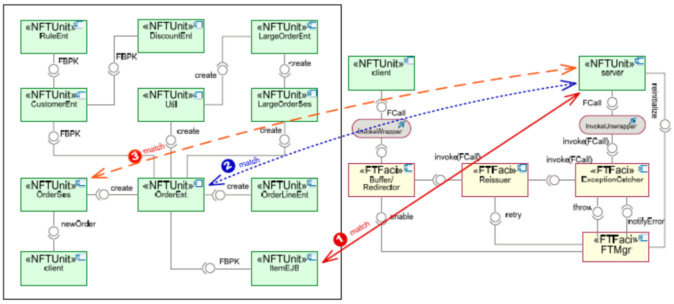


Fig. 11. The match relationship of RSA and FTS

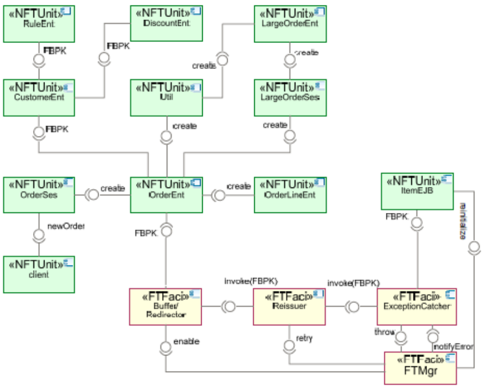


Fig. 12. The software architecture after first merging

The first step is to configure ItemEnt with micro-reboot style. In this process, "ItemEnt" corresponds to the "server", and "OrderEnt" corresponds to the "client". And the invocation between OrderEnt and ItemEnt disappeared. The result is shown in fig 12. The next two steps are similar. And we implement the model merging by QVT, the source code can be downloaded in our google code project [3];

We create three different versions of fault-tolerant ECperf by modifying its original SA. Each version conforms to one of the above three FTS. We also perform a set of comparative experiments to validate the practical correctness of the selection. In the experiments, micro-reboot mechanisms and simple retry mechanism are attached to the components as external utility mechanisms, with the supports of SM@RT. We periodically inject Java exceptions into *ItemBmpEJB* to simulate EDNT faults. As a result, the rates of successful submitted orders using Micro-reboot and Simple Retry are 87.3% and 50.7%, respectively, compare to 45.4% with no FT (Fig.13). It is clear that Micro-reboot style works better than Simple Retry style. The experimental result is consistent with the model checking result. It should also be minded that the fault tolerance induces performance penalty. When no exceptions are injected into the experiments, the response time is 19.12s with no FT, and 21.71s with micro-reboot (Fig.13), which increases 13.49% of response time on average.

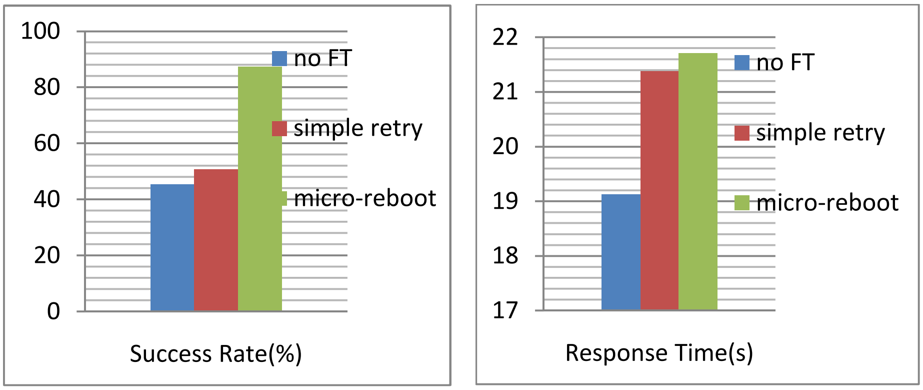


Fig. 13. The comparison success rate and response time

5.4 Discussion

From the evaluations of ECperf, we can see that the whole configuration process is more automatic than our previous work [6, 9], system advocates just need to specify the fault-tolerant properties that the target system needs to satisfy. In this section, we have a discussion after the experiment.

FTS Specification. We have described four fault tolerance solutions in this paper, and abstracted them into FTS: micro-reboot style, simple retry style, N-copy programming style, and retry block style. There are still some other solutions (such as Recovery Blocks style, N-Version Programming style, etc.), and we can leave them as future work.

Key Component Recommendation. In this paper, we use SBRA to estimate the reliability of the whole system, and recommend key components. And the analysis result is tallied with the actual situation. We choose SBRA mainly because it is a typical method for component-based system with a "CDG" model, which is more in favor of the analysis at architecture level. Some other component-based algorithms, such as the K. Goseva-Popstojanova's approach [27], can be integrated into our framework, by specifying the process in QVT.

Configuration Framework. For a complex component-based software system, the fault tolerance configuration process is complicated and challenging as follows: the components which need configuration are indeterminate; the fault-tolerant solutions are undefined; and the configuration process is without a guide. In this paper, we successfully handle these problems at architecture level: we abstract the target system into SA, which helps to identify the key component; we abstract the fault tolerance solution into FTS, which helps to check the fault tolerance properties and the selection of solutions; and the configuration process is under the model merging's guidance. In common cases, users only need to choose the FTS and target component, based on our recommendation. No future configuration or coding work is required. If users want to define their own FTS, integrate other analysis algorithms, etc., they just use the MOF standard languages to define their extension work at the model level.

6 Related Work

In the area of Architecting Fault-Tolerant Systems [1], components (computing entities), connectors (communication entities), and configuration (topology of components and connectors) have been used to model fault-tolerant software as FTSA. Previous work in the area mainly focuses on how to model a specific fault-tolerant mechanism [10, 13, 14, 25, 26], for example, exception handling-based mechanism [14, 26]. A few studies consider the reasoning or analysis on an FTS. Yuan et al. [26] specify a Generic Fault-Tolerant Software Architecture (GFTSA), which obeys idealized Fault-Tolerant Component style, in formal language Object-Z, and performs manual formal proofs to demonstrate fault-tolerant properties the GFTSA preserves. The authors also present a template to automate the customization process when using the style. Sözer et al. [25] specify the structure of a local recovery style in an UML profile, and perform performance overhead and availability analysis. In contrast, we uniformly model and analysis various mechanisms that can be used for third-party components as fault-tolerant styles.

The study of Architecting Fault-Tolerant Systems aims to achieve better fault-tolerant software by including FT in earlier development phase to bridge the gap between the requirement to build dependable software systems and the implementation to deal with failures in the software. As one of the important fault-tolerance mechanisms, exception handling is widely used in the study of architecting fault-tolerant software systems. A notable study is the CORRECT project [8] in Luxembourg, which introduces the Coordinated Atomic Actions (CAAs) mechanism in SA specification phase. The resulting SA specification with fault-tolerance notations is transformed into CAAs model automatically and further, transformed to an implementation framework. The output of such approach is a skeleton code that satisfies the functional and fault-tolerant requirements. It is based on model-driven techniques, and separates the function design from the fault-tolerant design of the system in the model layer. However, it only supports a specific fault-tolerant technology without any others. The approach in this paper abstracts the fault-tolerant technologies as FTS, and identifies the components which need to be configured automatically, which means the assemblers just need to input the properties which need to be satisfied.

7 Conclusion and Future Work

This paper proposes a model driven configuration of fault tolerance solution for component-based system. It needs to be polished in the future. At one side, the approach just provides an enablement to satisfy the fault-tolerant capabilities in the situation of a fault assumption. What capabilities should be satisfied and the type of fault assumption are given by developers. Therefore, how to facilitate the use of the enablement is critical for practice. For example, a powerful exception analysis support can alleviate the burden of developers in identifying which type of fault assumption the fault belongs to. The fault detection mechanisms help to decide what kind of fault-tolerant capabilities should be satisfied. On the other side, the fault-tolerant configuration may be more general and efficient. However, since the number of popular middleware in a period is relatively few, we argue that making a concrete middleware more powerful on exception handling is more important. This study is being carried out now, with the help of our Runtime Software Architecture [5][24].

Acknowledgments. This work is supported by the National Basic Research Program of China (973) under Grant No. 2009CB320703; the National Natural Science Foundation of China under Grant No. 61121063, 60933003; the European Commission Seventh Framework Programme under grant no. 231167; the Science Foundation Ireland grant 10/CE/I1855 to Lero; and the NCET.

References

1. Workshop on Architecting Dependable Systems, `http://www.cs.kent.ac.uk/wads/`
2. SM@RT: Supporing Models at Run-Time, `http://code.google.com/p/smatrt/`
3. SM@RT fault tolerance configuration, `http://code.google.com/p/smatrt-ftc/downloads/list`
4. Sherif, Y., Bojan, C., Ammar, H.H.: A Scenario-Based Reliability Analysis Approach for Component-Based Software. IEEE Transactions on Reliability 53 (2004)
5. Huang, G., Mei, H., Yang, F.-Q.: Runtime Recovery and Manipulation of Software Architecture of Component-based Systems. International Journal of Automated Software Engineering 13(2), 251–278 (2006)
6. Li, J., Chen, X., Huang, G., Mei, H., Chauvel, F.: Selecting Fault Tolerant Styles for Third-Party Components with Model Checking Support. In: Lewis, G.A., Poernomo, I., Hofmeister, C. (eds.) CBSE 2009. LNCS, vol. 5582, pp. 69–86. Springer, Heidelberg (2009)
7. Candea, G., et al.: JAGR: an autonomous self-recovering application server. In: Proc. of the 5th Int'l Workshop on Active Middleware Services, Seattle, USA, pp. 168–177 (2003)
8. Capozucca, A., Guelfi, N., Pelliccione, P., Muccini, H.: An Architecture-driven Methodology for Developing Fault-Tolerant Systems, Software Engineering Competence Center Technical Report nr. TR-SE2C-05-10, SE2C, Luxembourg (2005)
9. Huang, G., Wu, Y.: Towards Architecture-Level Middleware-Enabled Exception Handling of Component-based Systems. In: Component-based Software Engineering (2011)

10. Garlan, D., Chung, S., Schmerl, B.: Increasing System Dependability Through Architecturebased Self-Repair. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems. LNCS, vol. 2677. Springer, Heidelberg (2003)
11. Song, H., Huang, G., Chauvel, F., Zhang, W., Sun, Y., Shao, W., Mei, H.: Instant and Incremental QVT Transformation for Runtime Models. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 273–288. Springer, Heidelberg (2011)
12. ECperf webpage, `http://java.sun.com/developer/earlyAccess/j2ee/ecperf/download.html`
13. Issarny, V., Banatre, J.: Architecture-Based Exception Handling. In: Proc. of the 34th Annual Hawaii International Conference on System Sciences, vol. 9, p. 9058 (2001)
14. Lan, L., Huang, G., Wang, W., Mei, H.: A Middleware-based Approach to Model Refactoring at Runtime. In: Proceedings of the 14th Asia-Pacific Software Engineering Conference, APSEC 2007 (2007)
15. Avizienis, A., Lapri, J.-C., Randell, B., Landwehr, C.: Basic Conceptsand Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing 1(1), 11–33 (2004)
16. Avižieni, A., Kelly, J.P.J.: Fault Tolerance by Design Diversity: Concepts and Experiments. IEEE Computer 17(8), 67–80 (1984)
17. Ammann, P.E., Knight, J.C.: Data Diversity: An Approach to Software Fault Tolerance. In: Proceedings of the 17th International Symposium on Fault-Tolerant Computing Systems (FTCS-17), Pittsburgh, PA, pp. 122–126 (1987)
18. Pottinger, R.A., Bernstein, P.A.: Merging models based on given correspondences. In: Proc. 29th International Conference on Very Large Data Bases, VLDB 2003 (September 2003)
19. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Merging Models with the Epsilon Merging Language (EML). In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 215–229. Springer, Heidelberg (2006)
20. Object Management Group, UML$^{(TM)}$ Profile for Modeling Quality of Service and Fault Tolerance Characteristics and Mechanisms, `http://www.omg.org/docs/ptc/04-09-01.pdf`
21. Perry, D.E., Wolf, A.L.: Foundations for the study of software architecture. SIGSOFT Software Engineering Notes 17(4), 40–52 (1992)
22. Issarny, V., Banatre, J.: Architecture-Based Exception Handling. In: Proc. of the 34th Annual Hawaii International Conference on System Sciences, vol. 9, p. 9058 (2001)
23. Bose, P.: Automated Translation of UML Models of Architectures for Verification and Simulation Using SPIN. In: Proceedings of the 14th IEEE Int'l Conference on Automated Software Engineering, pp. 102–109. IEEE Computer Society Press, Los Alamitos (1999)
24. Song, H., Huang, G., Chauvel, F., Xiong, Y., Hu, Z., Sun, Y., Mei, H.: Supporting Runtime Software Architecture: A Bidirectional-Transformation-Based Approach. Journal of Systems and Software, doi:10.1016/j.jss.2010.12.009
25. Sözer, H., Tekinerdogan, B.: Introducing Recovery Style for Modeling and Analyzing System Recovery. In: Proc. of 7th IEEE/IFIP Working Conference on Software Architecture, Vancouver, Canada, pp. 167–176 (2008)
26. Yuan, L., Dong, J.S., Sun, J., Basit, H.A.: Generic Fault Tolerant Software Architecture Reasoning and Customization. IEEE Trans. on Reliability 55(3), 421–435 (2006)
27. Goseva-Popstojanova, K., Trivedi, K.: Architecture-based approach to reliability assessment of software systems. An International Journal on Performance Evaluation 45, 179–204 (2001)

Applying a Consistency Checking Framework for Heterogeneous Models and Artifacts in Industrial Product Lines

Michael Vierhauser[1], Paul Grünbacher[2],
Wolfgang Heider[3], Gerald Holl[3], and Daniela Lettner[3]

[1] Siemens VAI Metals Technologies, Linz, Austria
michael.vierhauser.ext@siemens.com
[2] Systems Engineering and Automation,
Johannes Kepler University Linz, Austria
paul.gruenbacher@jku.at
[3] Christian Doppler Laboratory for Automated Software Engineering,
Johannes Kepler University Linz, Austria
{heider,holl,lettner}@ase.jku.at

Abstract. Product line engineering relies on heterogeneous models and artifacts to define and implement the product line's reusable assets. The complexity and heterogeneity of product line artifacts as well as their interdependencies make it hard to maintain consistency during development and evolution, regardless of the modeling approaches used. Engineers thus need support for detecting and resolving inconsistencies within and between the various artifacts. In this paper we present a framework for checking and maintaining consistency of arbitrary product line artifacts. Our approach is flexible and extensible regarding the supported artifact types and the definition of constraints. We discuss tool support developed for the DOPLER product line tool suite. We report the results of applying the approach to sales support applications of industrial product lines.

Keywords: Model-based product lines, consistency checking, sales support.

1 Introduction and Motivation

Software product line engineering (SPLE) [14,17] is based on reusing heterogeneous artifacts such as software components, documents [19], or test cases [17] to increase the productivity of development and to improve product quality. Variability models are used in SPLE to define the commonalities and variability of the different products that can be derived from the product line (PL) [4]. Regardless of the variability modeling approach used, the size of the PL models as well as the heterogeneity of the involved artifacts represent major challenges in real-world PLs. Consistency needs to be ensured between heterogeneous artifacts such as source code, components, documents, or business calculations [13].

Our case studies with several industrial partners [6] showed that engineers in practice encounter significant challenges when maintaining the consistency of PL models and their corresponding artifacts. We thus developed an approach for maintaining consistency between variability models and parts of the underlying code base of a PL [23]. The approach is based on the transformation of source code into a model-based representation. An incremental checker [7] is then used to perform consistency checks between the code artifacts and the PL model. However, our approach was limited to variability models and their underlying code base. This is insufficient for industrial development environments which require multiple heterogeneous models and artifacts such as business calculation models [13] or documents [19] to define the PL and to maintain already derived products [11]. Additional challenges stem from the fact that these artifacts are typically managed and maintained by different people in distinct lifecycle stages.

This paper describes a framework for checking consistency of multiple, heterogeneous models and artifacts in SPLE. The generic framework addresses the limitations of our formerly developed approach described in an experience paper [23]. We report results of applying the approach in the area of sales support systems for product configuration in the domain of industrial automation. Our work has been performed using the DOPLER (**D**ecision-**O**riented **P**roduct **L**ine **E**ngineering for effective **R**euse) tool suite [6].

The remainder of this paper is structured as follows: Section 2 discusses related work. Section 3 describes the architecture of the implemented consistency checking framework and the steps necessary for applying the framework. Section 4 describes the application of the approach for maintaining and evolving sales support applications. Section 5 discusses key results and lessons learned. Section 6 concludes the paper with a summary and an outlook on future work.

2 Related Work

We discuss existing generic approaches for consistency checking; consistency checking approaches for specific artifact types; as well as consistency checking in PLs.

Generic consistency checking. These approaches address general consistency issues which are independent of the actual domain of the models. General purpose constraints ensure consistency between different artifacts in terms of containing or referencing elements. Nentwich *et al.* [16] present a consistency checking approach for arbitrary distributed software engineering documents encoded in XML. Their xlinkit approach uses XPath for checking constraints between artifacts. Constraints are defined in a rule language using XML syntax. The approach relies on an incremental checking strategy. Changes to XML documents are discovered by a diffing algorithm. Egyed [7, 10] presents an incremental approach for evaluating consistency rules after changes to arbitrary models. The approach is based on observing the behavior of consistency rules during their evaluation to identify the model elements that need to be checked after a change. The approach is evaluated using different types of UML diagrams. More recently, this approach has been implemented in the Rational Software Modeler tool [20]. Blanc *et al.* [2] focus on structural inconsistencies between different models in large-scale industrial software systems. Similar to Egyed *et al.* they use an event-driven approach which enables incremental evaluation of constraints. Consistency constraints are defined in Prolog and translated into Prolog queries for evaluation. Numerous constraint languages have been developed to ease the definition of consistency checks and their evaluation. Examples include OCL[1], the Xpand Check language[2] or Xtend 2[3].

Consistency checking of specific artifacts types. Numerous approaches exist for dealing with inconsistencies in specific modeling situations. For example, Tsiolakis and Ehrig [22] present an approach for checking consistency between class and sequence diagrams based on a common graph structure. Van der Straeten *et al.* [21] use a description logic to detect inconsistencies between sequence and state chart diagrams. Campbell *et al.* [3] use a model checker to evaluate inconsistencies within and across UML diagrams. Zisman and Kozlenkov [25] use a knowledge base and express consistency rules using patterns and axioms.

Consistency checking of variability models. Consistency checking is also receiving a lot of attention in the area of product line engineering. Approaches for PL evolution try to avoid the deviation from PL models up to the point where key properties no longer hold [12]. Several papers address the issue of consistency between models and code in PLs. For instance, Murta *et al.* [15] present an approach for ensuring consistency of architectural models and the corresponding implementation during evolution. The approach supports arbitrary evolution policies and is based on recording changes in a configuration management system. Several approaches exist to address consistency of PL models. Czarnecki *et al.* [5] present a feature-based approach using model templates. Constraints are defined in OCL. The approach is limited to UML models and feature models and does not provide extensibility for arbitrary artifacts which may be also part of a SPL. Consistency in SPLE also needs to address product derivation. For this purpose, Elsner *et al.* [8] present an incremental approach for checking consistency during derivation in multi product line environments.

3 A Generic Consistency Checking Framework

Maintaining and evolving PL models and their related artifacts is an extensive and error prone task. Therefore, the framework aims at continuously checking the consistency of models and artifacts that are stored in a PL workspace. The framework tracks changes to the workspace and triggers arbitrary user-defined consistency checks. The modeler is informed instantly and receives feedback about emerging constraint violations. Our incremental framework is based on

[1] Object Management Group OMG. Object Constraint Language, Version 2.2. formal/2010-02-01, February 2010.

[2] http://www.eclipse.org/modeling/m2t/?project=xpand

[3] http://www.eclipse.org/Xtext/#xtend2

existing work by Egyed [7]. We already implemented this approach in the context of product line engineering in earlier work [23]. However, the initial implementation was limited regarding the types and number of models and artifacts.

3.1 Framework Architecture

The framework depicted in Figure 1 can deal with heterogeneous models and arbitrary artifacts and is based on the Eclipse[4] platform.

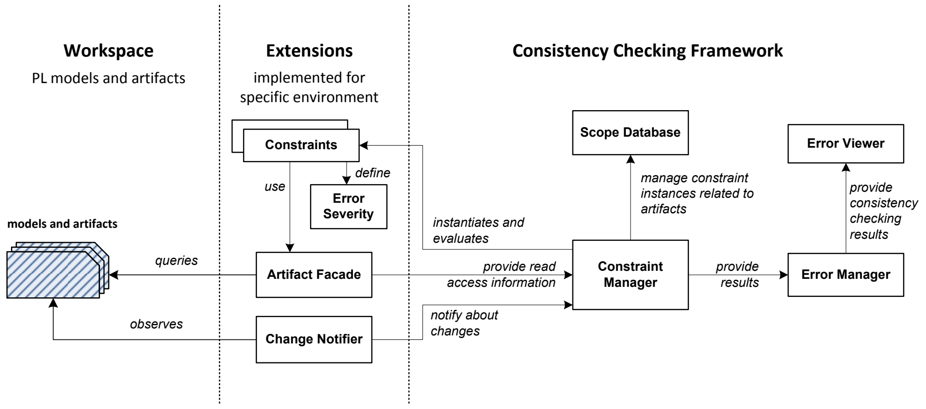


Fig. 1. Generic Consistency Checking framework architecture

The **Workspace** contains arbitrary models and artifacts required for a PL. For instance, a PL workspace might contain a variability model defining configuration options, source code, documents, or test cases.

Extensions are used to adapt the framework to specific artifacts and constraints. An *Artifact Facade* needs to be implemented for each artifact type. The use of the facade design pattern [9] allows a single access point and unified access to models and artifacts for querying specific elements or attributes which are relevant when evaluating constraints. The approach relies on profiling and change tracking of artifacts and models [7]. Read access on elements and attributes through the facades is tracked by the framework to enable incremental consistency checking. Our model profiler tracks all read-access operations performed when evaluating the constraints. This enables incremental consistency checking as only the affected constraints (=those which accessed the changed elements) need to be evaluated after changes to models or documents. A *Change Notifier* informs the framework about changes on artifacts to trigger constraint evaluations. Such a notification mechanism needs to be provided for all models and artifacts to be included in the consistency checking framework. Existing observer implementations on models or artifacts may be reused to enable the tracking of model element changes. The DOPLER tool suite provides a generic

[4] The Eclipse Foundation. http://www.eclipse.org

change tracking mechanism for variability models which is also used to determine change histories, to notify other tools, and to track evolution. Arbitrary *Constraints* can be defined that implement consistency checks. A constraint returns evaluation results regarding consistency of specific artifacts when executed by the framework. Concrete constraint definitions extend a base constraint provided by the framework which defines the data types of validation results and error messages. Despite the variety of different constraint languages available, our framework approach relies on the definition of constraints in Java. We intentionally decided not to use constraint programming languages or solvers for constraint evaluation. This allows a more flexible constraint definition and allows both modelers and domain experts to implement constraints.

The **Consistency Framework** is based on the following key components: The *Constraint Manager* controls the instantiation of available constraints which are provided through the Eclipse extension mechanism. Constraints are typically instantiated multiple times, more precisely, constraints are instantiated for every artifact instance they refer to. During initial evaluation of the constraint, the framework tracks all elements which are accessed by the constraint implementation. This information is preserved in the *Scope Database* to allow instant and incremental re-evaluation after changing an artifact. Each time the change notifier informs about an artifact change (e.g., after an engineer changes a variability model) the constraint manager triggers re-evaluation. The framework queries the scope database to retrieve the constraint instances that need to be re-evaluated. The evaluation results are provided to an *Error Manager* which feeds consistency change information into the *Error Viewer*. Depending on the type of inconsistencies and the implemented constraints, different severity levels can be defined (e.g., compilation errors or warnings). The framework allows mapping the severity levels to constraint classes or single evaluation results. Furthermore, the framework enables developers to define specific error messages and manual fixing instructions.

3.2 Applying the Framework

Four activities are necessary to tailor the framework to a specific PL development environment:

Identification of artifacts and dependencies. All artifacts relevant for consistency checking need to be identified. For instance, apart from variability models, artifacts such as product specifications, calculations, and configuration files may be part of a PL. These artifacts typically comprise various dependencies to the variability model or have dependencies to other artifacts.

Instrumentation of artifacts. The framework requires the implementation of extensions for the required artifact types. A specific artifact facade needs to be developed for each artifact type which converts the elements contained in an artifact into a unique and artifact independent representation. The *ArtifactIdentifier* enables the internal handling and assignment of artifacts and contained elements to constraints. Furthermore, read access on the identified artifacts is delegated to the facade to ensure that read notifications are provided to the consistency framework. A change notifier needs to be implemented to inform the framework about changes on artifacts and contained elements. Existing model or workspace observers may be reused or extended.

Development of consistency constraints. Possible inconsistencies between the involved artifacts and models can be identified based on the identified dependencies. Each inconsistency is implemented as a dedicated and self-contained constraint class. Each of these constraint implementations can be evaluated independently thus enabling incremental checking after the constraint has been instantiated.

Definition of error reporting categories and error messages. The resulting error messages as well as evaluation results of each constraint are customizable to support flexible and adaptable constraint definitions. This enables grouping of evaluation results by severity or by different artifact types (e.g., artifact specific error types or general PL validation types).

4 Application Examples: Sales Support Solutions for Industrial Product Lines

Siemens VAI – the world's leading company in engineering and plant-building for the iron, steel, and aluminium industries – has been developing PL models for a series of industrial PLs to assist sales staff with sales support applications providing business related information such as break-even or return on investment calculations during product configuration. The applications also allow creating product-specific artifacts such as sales documents or specifications [13,19]. The sales support applications and their underlying models were developed using the DOPLER tool suite [6,24] for variability modeling [6] and product derivation [18].

Table 1 provides an overview of the different PLs, the involved artifact types and the number of elements contained in the different artifacts. We will use the ECS PL – a fully automatic end-to-end solution for electrode control in electric furnaces – as a running example to discuss the numbers provided in Table 1. We focus on the description of extensions and constraints required for the different artifacts involved in these sales applications. We do not evaluate performance and memory consumption which was already part of earlier work [23].

4.1 Models and Artifacts

The following models, artifacts and dependencies are of interest for the sales support applications:

Variability models in the DOPLER approach contain decisions and assets. Decisions represent configuration choices that need to be made when deriving a customer-specific product from a PL. Decisions represent the differences between PL members. They are defined by a question that is asked to the user during

Table 1. Sales support product lines with number of formulae (F), configuration decisions (D), variation points in documents (VP) and references between the different artifacts (R).

Product Line	Description	#D	#F	#VP	#R
1: ChatterReduction	Integrated mechatronic solution for eliminating 3rd octave chatter in tandem mills.	71	7	32	47
2: BeamBlank	Continuous casting of near-net-shape beam blanks.	39	17	–	29
3: Zinc coating application	Control system guaranteeing quality of zinc coating thickness.	22	3	16	22
4: Roller	Roller configuration and upgrading in continuous casting machines.	43	11	–	21
5: LVL	Technology package for automation, motors and drives for minimizing shape defects through controlled deformation of products.	44	23	–	43
6: MLP	A fully integrated system for accelerated cooling and direct quench in plate production.	45	23	50	94
7: MPD	Market Price Derivation calculation tool.	20	53	–	21
8: PROFLAT	A technology package for optimized product profile, flatness and mill productivity.	39	15	28	55
9: ECS	Electrode control system for electric arc furnaces.	114	15	69	95
10: Oscillation	Automated system for adjustment of oscillation parameters for improved surface quality in steel production.	47	16	46	74
11: SoftReduction	Automated system for optimized internal cast product quality with dynamic soft reduction.	24	11	23	36
12: WidthChange	Automated system for dynamic and remote mold width adjustment via hydraulic drives for higher plant productivity.	21	7	19	26
13: Welding	Automated welding system for continuous casting lines.	28	8	27	42
14: RollLubrication	Technology package for roll friction and rolling for reduction in rolling mill stands.	42	16	34	58
15: PlantQuality-Attributes	Customer Service tool regarding plant quality attributes.	118	121	–	108
Median		43	15	23	43

product derivation. Answering a question sets the value of a decision. The ECS variability model, for instance, contains 114 configuration decisions (D).

Assets represent the reusable core artifacts of the PL like system components or documents. Assets can depend on each other functionally (e.g., one component requires another component) or structurally (e.g., a component is part of a subsystem). DOPLER allows modeling assets at arbitrary granularity and with user-defined attributes and dependencies. Users can create domain-specific metamodels to define the types of assets, attributes, and dependencies.

Calculation models comprise an arbitrary number of formulae and define constants, formula groups and diverse additional attributes. Furthermore, a calculation model contains mappings to the variability model for accessing decision values or asset attribute values. Formulae are specified in a DSL based on the Java language [13]. This allows the definition of calculations containing basic arithmetic operations and enables the extension of the defined DSL by implementing custom functions. In the ECS sales process different business values are needed. The ECS calculation model stores calculations regarding the saved CO_2 emission, total price of included spare parts and break-even calculations. In total the ECS calculation model comprises 15 different formulae (F).

Document templates provide the basis for creating product specific documents. The templates contain markups to indicate variable parts. Generators provided for a specific PL use these document markups for either substituting document parts or inserting decision values and calculation results into the document based on a configured product [19]. The templates necessary for generating ECS documents including price lists, calculation results and product information contain 69 variation points (VP) representing the variable parts of the document template.

The discussed ECS PL artifacts contain 95 cross-references (R), e.g., document templates referring to formulae stored in the calculation model or formulae using configuration decisions stored in the variability model.

4.2 Applying the Framework

Identification of artifacts and dependencies. Figure 2 provides an overview of the involved artifacts and their dependencies. All 15 PLs contain variability models and separate calculation models with numerous dependencies. Document templates for generating product specific documents are used in 10 PLs. The comprised markups refer to variability model elements and calculation model elements.

Instrumentation of artifacts. Existing DOPLER variability models and calculation models provide a standardized API for accessing the contained model elements. Moreover, change notifications are provided via model observers. To enable the handling of document templates within the framework each document is transformed into a model-based representation upon initialization, to extract and analyze the contained markups. Furthermore, file monitors are needed to observe and propagate document changes. The code example shown in Listing 1.1

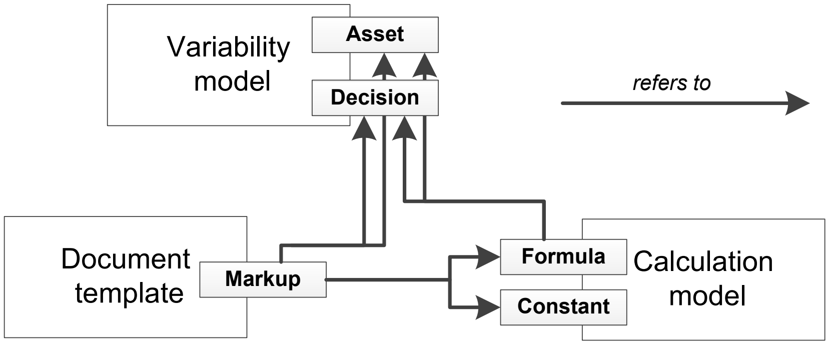


Fig. 2. Models and artifacts and their dependencies

provides an overview of a facade implementation for a calculation model. The facade implementation provides methods to convert calculation model elements into artifact independent representations, to initially evaluate the constraint, and to access the calculation model elements (e.g., formulae). Furthermore, facades track and forward the accessed model elements to the consistency checking framework for profiling.

Listing 1.1. Example program code of a facade implementation for a calculation model.

```
public class CalculationModelFacade
extends AbstractProfilerFacade implements IFacade<CalculationModel>
//This method transforms a given calculation model object into the
//unified representation of an artifact identifier
   public ArtifactIdentifier toArtifactIdentifier(Object o){
      if(Object instanceof IFormula){
        return new ArtifactIdentifier(((IFormula)o).getModel().getUniqueId(),
        ((IFormula)o).getUniqueId())
      }
      ...
   }
//This method converts a given artifact identifier into the real object
//contained in a calculation model (e.g., a formula or a constant)
   public Object fromArtifactIdentifier(ArtifactIdentifier ai){
     ICalculationModel m = ModelProvider.get(a1.getArtifactId());
      return m.getElement(ai.getElementId());
   }
//This method handles the initialization of a calculation model by
//passing all relevant elements to the consistency checker
   public boolean processArtifact(CalculationModel artifact, IFile file) {
      ArtifactStore.addArtifact(artifact,file);
      //Process constants and formulae of the model
      for (IFormula f : artifact.getFormulas()) {
        Checker.addElement(toArtifactIdentifier(f), IFormula.class);
      }
      for (IConstant c : artifact.getConstants()) {
        Checker.addElement(toArtifactIdentifier(c), IConstant.class);
      }
   }
//This method wraps the access to a certain formula
//and notifies the consistency checker
   public IFormula getFormula(String formulaName, ArtifactIdentifier
        baseArtifact, IConstraint caller) {
      ICalculationModel model = ModelProvider.get(baseArtifact.getId());
      notifyAccess(baseArtifact,formulaName, caller);
      return model.getFormula(formulaName);
   }
...
}
```

Development of consistency constraints. Models can either have intra-inconsistencies (e.g., a referenced constant is not present in the calculation model) or inter-inconsistencies (e.g., a referenced decision in the calculation model is not present in the related variability model). Document markups contain references to decisions or assets defined in a variability model, as well as dependencies to formulae or constants specified in a calculation model. The code example shown in Listing 1.2 depicts a sample implementation of a single constraint which ensures the validity of a formula term defined in a calculation model. The constraint is instantiated multiple times, once for each formula contained in a calculation model. The example ensures the consistency of formula terms by verifying that the term only contains references to decisions or asset attributes in a related variability model, defined constants, double or integer values, or other formula results. By using the provided facade implementation and the available calculation model API, a new constraint can be added to the development environment by writing less than 50 lines of Java code in most cases.

Listing 1.2. Example program code of a constraint implementation.

```java
public class FormulaConsistencyConstraint extends AbstractConstraint
//This constraint ensures that a term of a formula is valid
   public ValidationResult evaluate(){
      IFormula toCheck = Facade.fromArtifactIdentifier(this.id);
     //check every element of the formula
      for(Element e: toCheck.getElements()){
         switch(e.getType()){
            case DECISION_REFERENCE: checkDecisionReference(e);break;
            case ASSET_REFERENCE: checkAssetReference(e);break;
            case CONSTANT: checkConstant(e);break;
            case PRIMITIVE: checkValidPrimitive(e);break;
            case FORMULA: checkValidFormula(e);break;
      }
   }
  }

 //check if the Decision exists in the related Variability model
   private void checkDecisionReference(Element e) {
      IDecision target = Facade.getElement(e);
      if(target!=null){//The decision exists
         setValidationResult(ValidationResult.CONSISTENT);
      }else{ //The decision does not exists
         addError(ErrorSeverity.ERROR, "The formula is inconsistent! " + e + "
              could not be resolved!");
         setValidationResult(ValidationResult.INCONSISTENT);
     }
   }
...
}
```

Definition of error reporting categories and error messages. Each artifact can provide its own error category to ease error tracking. Furthermore, depending on the constraint, a severity level can be defined, providing a fine-grained distinction between fatal errors, simple warnings or modeling guideline violations. Error categories and severity levels are defined declaratively in XML and can be applied to an artifact type or to a specific single constraint. Listing 1.3 provides an example of a simple error category mapping to the constraint described in Listing 1.2. In this example, the class *FormulaConsistencyConstraint* is mapped to the artifact type *Formula* meaning that the constraint is applied to every element of this type provided to the consistency checker. Furthermore, *severity* defines the level of inconsistency the evaluation result of this constraint can produce. The framework offers three levels of severity (info, warning and error) but constraints may define their own severity levels. The *marker id* is primary used for visualization purpose to ease grouping of related inconsistencies.

Listing 1.3. Declarative binding of constraint classes to specific artifact types and error types.

```
<constraint class= "FormulaConsistencyConstraint">
  <artifactType artifactType= "Formula">
  </artifactType>
  <marker id= "CALCULATIONMODEL.ERROR">
  <severity= "ERROR">
  </marker>
</constraint>
```

4.3 Consistency Checking in the DOPLER IDE

A major goal when developing our incremental checker was to increase usability by providing immediate feedback to modelers about the detected inconsistencies as part of the DOPLER development environment. The framework is therefore seamlessly integrated in the Eclipse-based DOPLER tool suite. Figure 3 provides an overview of the DOPLER tool suite showing the *Workspace* with PL artifacts, the *Variability Model Editor* and the *Calculation Model Editor*. Identified inconsistencies are presented in the *Error Viewer*. The *Document Template* represents a product specific document and contains markups.

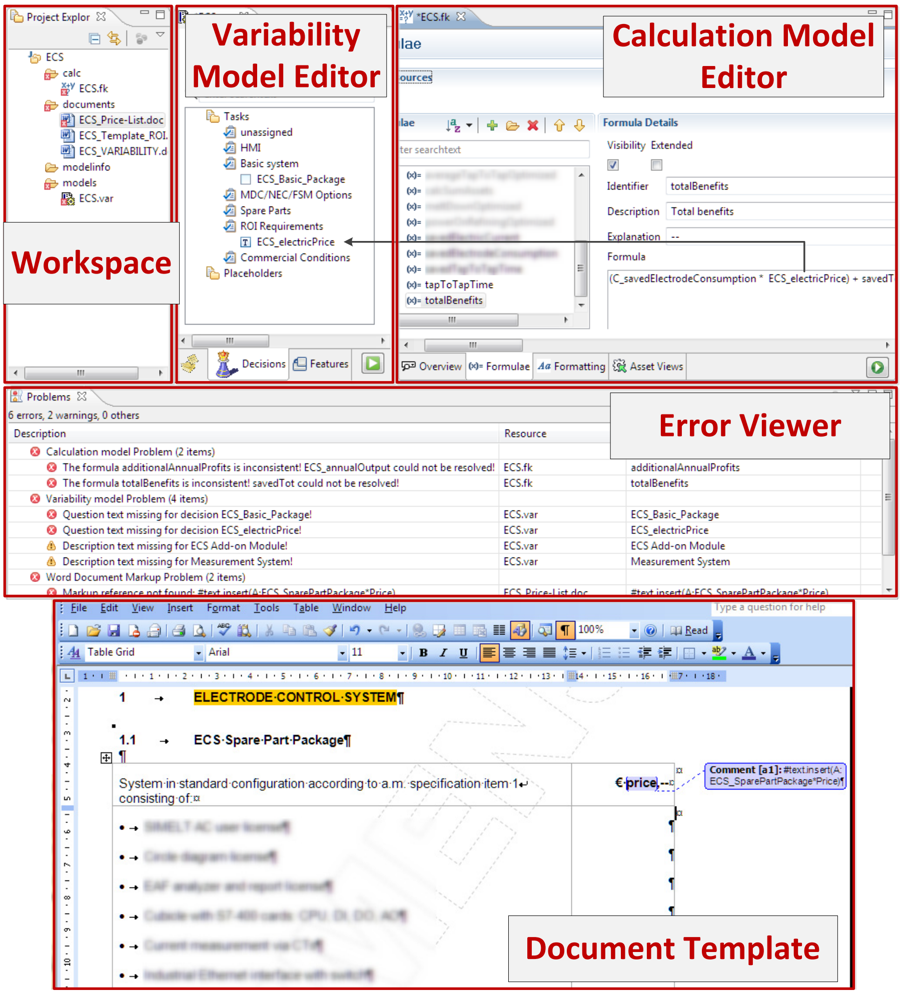
A modeler can define decisions and their attributes in the *Variability Model Editor* and formulae or constants in the *Calculation Model Editor*. Markups are directly defined within a *Document Template* which is supported by a Visual Basic application. In the *Calculation Model Editor*, the modeler can define calculations and simply add or remove relations to decisions or asset attributes via drag and drop. Manipulating components in the editors has an immediate effect. For instance, after removing or renaming a decision which is referenced in the corresponding calculation model, all involved constraints are re-evaluated automatically. Feedback about detected inconsistencies is provided immediately in the *Error Viewer*. The error messages are grouped into artifact-related error categories.

5 Lessons Learned

The identification of potential inconsistencies is crucial for successful maintenance and evolution of a PL. We summarize lessons learned of applying the generic consistency checking framework to 15 different PLs in the domain of industrial automation.

Ensure flexibility with respect to different artifact types. Product lines evolve over time and it is important that a consistency checking framework ensures flexibility regarding new artifact types and new constraints. Our solution is easily extensible with new artifact types by implementing a facade class for accessing specific artifact elements. Arbitrary constraints can be defined using facade implementations for different artifact types. No changes to the artifact implementations are necessary if proper APIs for reading artifact elements and observing changes are provided.

Provide immediate feedback to ease modeling. Modelers should be instantly aware of erroneous changes to artifacts. Inconsistencies can be caused by different types



Problems

6 errors, 2 warnings, 0 others

Description	Resource	
Calculation model Problem (2 items)		
The formula additionalAnnualProfits is inconsistent! ECS_annualOutput could not be resolved!	ECS.fk	additionalAnnualProfits
The formula totalBenefits is inconsistent! savedTot could not be resolved!	ECS.fk	totalBenefits
Variability model Problem (4 items)		
Question text missing for decision ECS_Basic_Package!	ECS.var	ECS_Basic_Package
Question text missing for decision ECS_electricPrice!	ECS.var	ECS_electricPrice
Description text missing for ECS Add-on Module!	ECS.var	ECS Add-on Module
Description text missing for Measurement System!	ECS.var	Measurement System
Word Document Markup Problem (2 items)		
Markup reference not found: #text.insert(A:ECS_SparePartPackage*Price)	ECS_Price-List.doc	#text.insert(A:ECS_SparePartPackage*Price)

Fig. 3. The generic consistency checking framework assists PL engineers modeling the sales support applications in the DOPLER IDE. The integrated Consistency Checker identifies inconsistencies and presents them in the *Error Viewer*.

of artifacts which might be maintained by different engineers. Instant feedback about inconsistencies to artifacts maintained by other engineers increases awareness regarding the impact of changes and eases fixing inconsistencies in multi-user workspaces.

Tolerate inconsistencies. Inconsistencies between artifacts can be temporary and may be the result of intermediate modeling steps that will be resolved by subsequent actions. Our framework tolerates inconsistencies and does not prevent users from entering inconsistent input [1]. Inconsistencies are displayed in a separate view and do not interfere with the actual modeling tasks. They may be ignored or resolved one by one. Checking may also be turned off at any time if desired. Especially inconsistencies between different artifact types maintained by different engineers require support for sequential maintenance. Our consistency checking framework is able to evaluate each constraint instance independently and thus work can continue even if inconsistencies are not fixed immediately. For example, an inconsistency between a calculation model and a variability model may be introduced by a change to the variability model reflecting PL asset changes. After the modeler finishes variability model adaptations, the changes including inconsistencies to the calculation model are committed to the repository. Another engineer in charge of the calculation model is informed about the newly introduced inconsistencies and can fix them.

Analyze artifact dependencies as starting point for defining constraints. Potential inconsistencies can be identified by thoroughly analyzing the dependencies among artifacts. Defining consistency constraints requires domain knowledge and high coverage can only be achieved by involving experts familiar with the models and artifacts. We carried out several workshops with domain experts and engineers to identify inconsistencies and to subsequently define constraints.

Choose right level of granularity to ensure high performance. Artifact instrumentation at different levels of granularity has huge impact on the performance of consistency checking. For example, if changes or read access can only be observed at model level, all constraint instances related to all model elements have to be re-evaluated each time the model changes, no matter what constraints are affected by a change. Observing fine-grained changes (e.g., attribute values of model elements) enables incremental checking and re-evaluation of affected constraints only. The performance of the consistency checking framework may be drastically reduced when proprietary file formats and artifacts needed to be checked for which no proper API and fine-grained change notifications are available.

Carefully design error messages. Error messages need to be clearly understandable by domain experts and PL engineers. As inconsistencies arise between different artifact types a clear separation must be possible, e.g., according to a modelers responsibility. Our framework allows a structured presentation of inconsistencies depending on the error category and importance. This allows modelers to fix emerging inconsistencies instantly if desired.

6 Conclusions and Future Work

We presented a generic framework to check consistency between multiple heterogeneous artifacts in SPLE. Our approach is based on an incremental strategy [7] and uses artifact-specific facades to provide support for various artifact types. The framework is easily extensible with new artifact types and additional constraints. A set of predefined extensions is designed to be implemented for each desired artifact type. We reported results and experiences of applying the framework when developing Siemens VAI sales support solutions for 15 different industrial PLs. Our approach supports modelers to ensure consistency between variability models, calculation models, and document templates required in these PLs.

Our emphasis so far was to support modelers during *domain engineering* by ensuring immediate feedback about detected inconsistencies while defining variability and maintaining the PL. However, ensuring consistency during *application engineering* remains a big challenge. In future work we will augment the presented consistency checking framework with runtime consistency checks. We will extend the framework to provide feedback on global constraints affecting also the derived products. Furthermore, we will refine our solution regarding repair suggestions for automated fixing of inconsistencies.

References

1. Balzer, R.: Tolerating inconsistency. In: Proceedings of the 13th International Conference on Software Engineering, pp. 158–165. IEEE Computer Society Press (1991)
2. Blanc, X., Mounier, I., Mougenot, A., Mens, T.: Detecting model inconsistency through operation-based model construction. In: Proceedings of the 30th International Conference on Software Engineering, pp. 511–520. ACM (2008)
3. Campbell, L.A., Cheng, B.H.C., McUmber, W.E., Stirewalt, K.: Automatically detecting and visualising errors in UML diagrams. Requirements Engineering 7(4), 264–287 (2002)
4. Czarnecki, K., Grünbacher, P., Rabiser, R., Schmid, K., Wasowski, A.: Cool features and tough decisions: A comparison of variability modeling approaches. In: International Workshop on Variability Modelling of Software-Intensive Systems, pp. 173–182. ACM (2012)
5. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness OCL constraints. In: Proceedings of the 5th International Conference on Generative Programming and Component Engineering, pp. 211–220. ACM (2006)
6. Dhungana, D., Grünbacher, P., Rabiser, R.: The DOPLER meta-tool for decision-oriented variability modeling: a multiple case study. Automated Software Engineering 18(1), 77–114 (2011)
7. Egyed, A.: Instant consistency checking for the UML. In: Proceedings of the 28th International Conference on Software Engineering, pp. 381–390. ACM (2006)
8. Elsner, C., Lohmann, D., Schröder-Preikschat, W.: Fixing configuration inconsistencies across file type boundaries. In: Euromicro Conference on Software Engineering and Advanced Applications, pp. 116–123 (2011)
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley, Boston (1995)
10. Groher, I., Reder, A., Egyed, A.: Incremental Consistency Checking of Dynamic Constraints. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 203–217. Springer, Heidelberg (2010)
11. Heider, W., Rabiser, R., Grünbacher, P.: Facilitating the evolution of products in product line engineering by capturing and replaying configuration decisions. International Journal on Software Tools for Technology Transfer (2012)
12. Johnson, S., Bosch, J.: Quantifying software product line ageing. In: Proceedings of the Workshop on Software Product Lines at ICSE 2000, pp. 27–30. ACM (2000)

13. Lettner, D., Vierhauser, M., Rabiser, R., Grňubacher, P.: Supporting end users with business calculations in product configuration. In: Proceedings of the of the 16th International Software Product Line Conference, Salvador, Brazil (2012)
14. van der Linden, F.J., Schmid, K., Rommes, E.: Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering. Springer (2007)
15. Murta, L.G.P., van der Hoek, A., Werner, C.M.L.: ArchTrace: Policy-based support for managing evolving architecture-to-implementation traceability links. In: Proceedings of the International Conference on Automated Software Engineering, pp. 135–144 (2006)
16. Nentwich, C., Emmerich, W., Finkelstein, A., Ellmer, E.: Flexible consistency checking. ACM Transactions on Software Engineering Methodology 12(1), 28–63 (2003)
17. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus (2005)
18. Rabiser, R., Grünbacher, P., Dhungana, D.: Supporting product derivation by adapting and augmenting variability models. In: Proceedings of the 11th International Software Product Lines Conference, pp. 141–150. IEEE Computer Society (2007)
19. Rabiser, R., Heider, W., Elsner, C., Lehofer, M., Grünbacher, P., Schwanninger, C.: A Flexible Approach for Generating Product-Specific Documents in Product Lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 47–61. Springer, Heidelberg (2010)
20. Reder, A., Egyed, A.: Model/Analyzer: a tool for detecting, visualizing and fixing design errors in UML. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, pp. 347–348. ACM, New York (2010)
21. Straeten, R.V.D., Mens, T., Simmonds, J., Jonckers, V.: Using description logic to maintain consistency between UML models. In: Proceedings of the 6th International UML Conference, pp. 326–340 (2003)
22. Tsiolakis, A., Ehrig, H.: Consistency analysis of UML class and sequence diagrams using attributed graph grammars. In: Proceedings of Graph Transformation and Graph Grammars, Berlin, Germany, pp. 77–86 (2000)
23. Vierhauser, M., Grünbacher, P., Egyed, A., Rabiser, R., Heider, W.: Flexible and scalable consistency checking on product line variability models. In: Pecheur, C., Andrews, J., Nitto, E.D. (eds.) Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering, pp. 63–72. ACM (2010)
24. Vierhauser, M., Holl, G., Rabiser, R., Grünbacher, P., Lehofer, M., Stürmer, U.: A deployment infrastructure for product line models and tools. In: Proceedings of the 15th International Software Product Line Conference, pp. 287–294. IEEE Computer Society (2011)
25. Zisman, A., Kozlenkov, A.: Knowledge base approach to consistency management of UML specification. In: Proceedings of the International Conference on Automated Software Engineering, pp. 359–363 (2001)

Generation of Operational Transformation Rules from Examples of Model Transformations

Hajer Saada[1], Xavier Dolques[2], Marianne Huchard[1], Clémentine Nebut[1], and Houari Sahraoui[3]

[1] LIRMM, Université de Montpellier 2 et CNRS, Montpellier, France
first.last@lirmm.fr
[2] INRIA, Centre Inria Rennes - Bretagne Atlantique, Campus universitaire de Beaulieu, 35042 Rennes, France
xavier.dolques@inria.fr
[3] DIRO, Université de Montréal, Canada
sahraouh@iro.umontreal.ca

Abstract. Model transformation by example (MTBE) aims at defining a model transformation according to a set of examples of this transformation. Examples are given in the form of pairs, each having an input model and its corresponding output transformed model, with the transformation traces. The transformation rules are then automatically extracted from the examples. In this paper, we propose a two-step approach to generate the transformation rules. In a first step, transformation patterns are learned from the examples through a classification of the model elements of the examples, and a classification of the transformation links using Formal Concept Analysis. In a second step, those transformation patterns are analyzed in order to select the more pertinent ones and to transform them into operational transformation rules written for the Jess rule engine. The generated rules are then executed on examples to evaluate their relevance through classical precision/recall measures.

1 Introduction

Model Transformation is a key component of Model Driven Engineering (MDE). In model-driven development, the involved models are processed by programs as a matter of priority (rather than by hand). To ease the development of such programs handling models, several languages were introduced, e.g. graph transformation languages such as VIATRA [4], declarative or semi-declarative languages like ATL [3], or object-oriented and imperative languages such as Kermeta [25].

Implementing a model transformation requires two distinct skills: model-driven engineering skills (in particular, metamodeling and model-transformation environments), and domain-specific skills, *i.e.*, good knowledge about the specification of the transformation: the input domain, the output domain, and the transformation rules by themselves. While the first skills are possessed by model-driven engineering experts, the second ones are specific to domain experts. Experience shows that domain experts more easily give transformation examples than complete and consistent transformation rules [16]. In this context, Model Transformation By Example (MTBE) [28] has emerged as a convenient way to let domain experts design transformations by giving an initial set of examples. An example consists of an input model, the corresponding transformed model, and fine-grained mappings between the constructs of both models. From those examples, an MTBE approach learns transformation rules. When those rules are operational, *i.e.*, they are written in a rule language disposing of a rule engine, they form the model transformation.

In this context, we present a Model Transformation By Example approach that goes from examples down to operational transformation rules. The learning mechanism used is based on Relational Concept Analysis (RCA) [12], a variant of Formal Concept Analysis [10]. It results in a hierarchy of non-operational rules called transformation patterns. Such transformation patterns are analyzed and filtered to derive the more relevant ones. The selected transformation patterns are then transformed into concrete and operational transformation rules that can be processed by the Jess rule engine [5]. The learning of the transformation patterns is a previous work from the authors [8], in this paper we introduce the filtering of the obtained transformation patterns, and we explain how to obtain operational rules from the transformation patterns. Finally, since the obtained rules are operational, experiments have been carried out on a case study in order to measure the relevance of the generated rules.

The remainder of this paper is structured as follows. We start by introducing the problem and describing our two-step approach in Section 2. Then, in Section 3, we briefly explain how RCA is used to extract information from examples and to generate transformation patterns. In this section, details are also given on how the obtained transformation patterns are filtered and refined. Section 4 describes the mapping of the transformation patterns into Jess rules. We present an evaluation of the approach and a discussion about the obtained results in Section 5. Section 6 presents the related work. Section 7 concludes the paper and describes future work.

2 Overview of the Rules Generation and Execution

Model-Transformation By Example (MTBE) consists in learning transformation programs/rules from examples. Usually, an example is composed of a source model, the corresponding transformed model, and transformation links between those two models. To illustrate MTBE, let us consider the well-known case of transforming UML class diagrams into relational schemas, used, among others, in [19]. For this transformation, examples are given in the form of: an input UML model (such as the one given in Figure 1), the corresponding transformed relational model (such as the one given in Figure 2), and transformation links making explicit from which elements of the UML model, the elements of the relational model stem from. For instance, a transformation link is given to specify that class `Client` is mapped into table `Client`. A transformation link is equivalent to a link of an execution trace of the expected transformation, *i.e* two

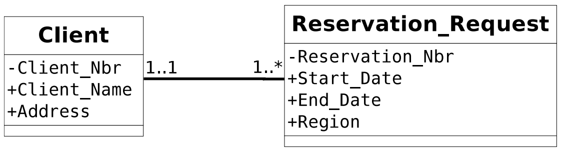


Fig. 1. Example for the UML2R transformation: input model

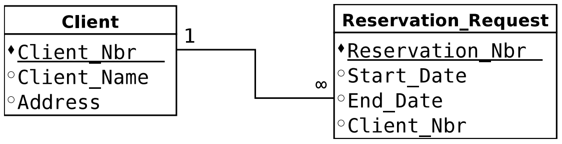


Fig. 2. Example for the UML2R transformation: transformed model

elements are related by a transformation link if the information contained in the first element is necessary to build the second one.

An MTBE process analyzes the examples and learns from them transformation rules such as *a class is transformed into a table*, or *a UML property linked to a class (i.e., an attribute and not a role) is transformed into a column of a table.* This process should produce operational rules, *i.e.*, rules that can be directly executed by a rule engine to transform any source model into a target model.

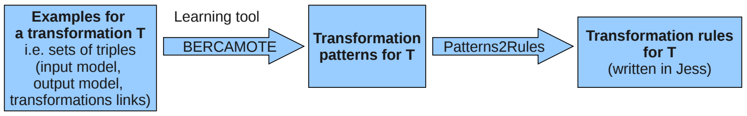

Examples for a transformation T i.e. sets of triples (input model, output model, transformations links)
Learning tool
BERCAMOTE
Transformation patterns for T
Patterns2Rules
Transformation rules for T (written in Jess)

Fig. 3. A two-step approach for MTBE

We propose to generate the operational rules in a two-step approach, as illustrated in Figure 3. The first step is the analysis of examples, that learns transformation patterns using Relational Concept Analysis. This step is supported by the Bercamote tool, that has been introduced in [8]. Each obtained transformation pattern describes a premise in the form of an input model pattern (based on the input metamodel), and a conclusion, in the form of the output model pattern (based on the output metamodel) that should be obtained after the execution of the transformation. The transformation patterns are ordered in a hierarchy. This hierarchy is analyzed to select the more relevant patterns, and sometimes to select in a transformation pattern the more pertinent fragment. We here target model-to-model transformations in which both models represent the same data but in different languages or using different structural constraints *e.g.* a transformation applying design patterns to enforce good structural modeling practices in a language. On the contrary, our MTBE approach is not well-suited to learn transformations in which new values are computed *e.g.* we cannot learn a renaming policy that forces to use lowercase for attributes names. Widening the scope of the transformations that can be learned is possible but would impact on the complexity of the results and the efficiency of the approach.

The main contribution on this paper deals with the second step, that makes the patterns operational. This is done by transforming them into rules that can be executed by a rule engine. To make the transformation patterns operational, we have transformed them into Jess rules and executed them using the Jess Rule engine. This step is detailed in Section 4.

3 A By-Example Approach to Obtain Transformation Patterns

As stated in Section 2, a key step in our MTBE approach consists in generating transformation patterns. Such patterns describe how a source model element is transformed into a target model element, within a given source context and a given target context. This step has been presented in [8], and is summarized in the beginning of this section, whereas the end of this section is dedicated to the filtering of the obtained transformation patterns.

3.1 Obtaining the Transformation Patterns

To derive patterns from examples, a data analysis method is used, namely Formal Concept Analysis (FCA) [10] and its extension to relational data, the Relational Concept Analysis (RCA) [12]. Both Formal and Relational Concept Analysis, also used for data mining problems, group entities described by characteristics into concepts, ordered in a lattice structure. While FCA produces a single classification, RCA computes several connected classifications.

Source and target model elements are classified using their metaclasses and relations. The transformation link classification relies on model element classifications and groups links that have similarities in their source and target ends: similar elements in similar contexts. From the transformation link classification, we derive a transformation pattern hierarchy, *i.e.*, a lattice of patterns, where patterns are organized by inclusion. Fig. 4 shows an excerpt of the obtained pattern hierarchy for the transformation of UML class diagrams into relational models. It contains two transformation patterns (in the two inner boxes). The transformation pattern in the bottom box is more specific than the one in the top box, which is indicated by the inclusion edge between the two boxes. The patterns are automatically named by our tool, they have a prefix beginning by *TPatt* for *transformation pattern*, then we find the number of the pattern, and finally the number of the concept representing the pattern, as generated by our RCA/FCA algorithms.

In each concept representing a transformation pattern, we have two types in two ellipses connected by a bold edge. The source ellipse of the bold edge represents the type T_s of the element to transform by the pattern. It can be seen as the main type of the premise. For instance, in Concept `TPatt_2-Concept_57`, we see that the pattern aims at transforming *properties*. This main type of the premise is linked, with non-bold edges, to the environment that an element of type T_s must have in order to be transformed by the pattern. Those edges are named according to the relation-role names between the type T_s and its environment in the metamodel. Those edges also have a cardinality defining the cardinality of the environment. Such an environment corresponds to the rest of the premise. For instance, in Concept `TPatt_2-Concept_57`, `Property` is linked to a `Class` with an edge named `property` and with a cardinality [1..*]. This means that the premise corresponds to a property, and that this property is linked to a class. The target ellipse of the bold edge represents the main type T_t of the conclusion of the pattern, *i.e.*, a T_s will be transformed into a T_t (with a specific environment). For example, in the transformation pattern `TPatt_2-Concept_57`, the conclusion corresponds to a column, linked to a table, linked, in turn, to a primary key.

The transformation pattern `TPatt_2-Concept_57` has been deduced from a set of transformation links that were grouped together because they link a property (connected to a class) to a column (connected to a table, itself connected to a primary key). This pattern is included in the pattern of sub-concept `TPatt_5-Concept_70`. This latter is more specialized because in addition to link the table to a primary key, it also links it to a foreign key.

3.2 Patterns Lattice Simplification

After obtaining the lattice of transformation patterns, we select in this lattice the useful/relevant patterns or pattern fragments.

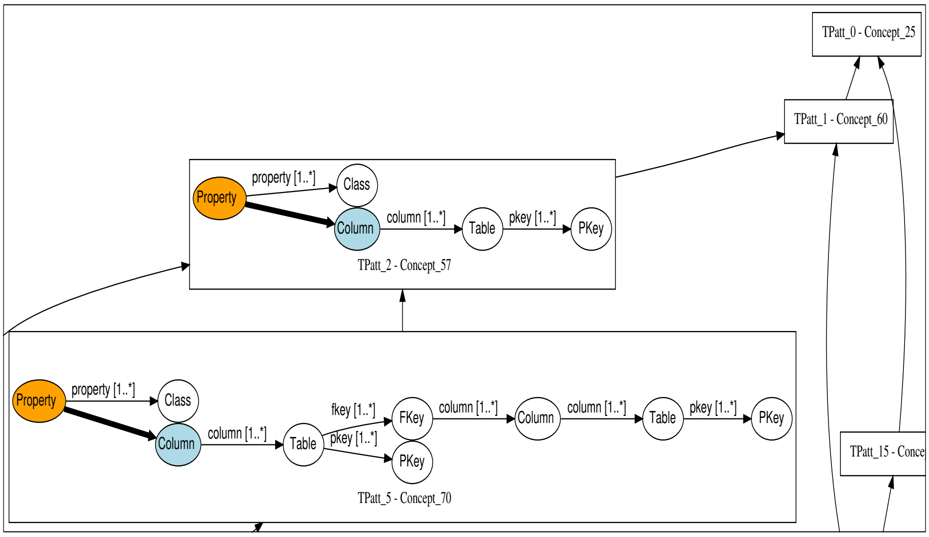


Fig. 4. An excerpt of the obtained hierarchy for the example UML class diagrams to relational models

In the lattice of Figure 4, for instance, concepts `TPatt_0-Concept_25` and `TPatt_1-Concept_60` are empty. They do not contain information about the transformation. They are present in the lattice to link other concepts (representing patterns) not shown in this excerpt. In the final transformation, those empty patterns are automatically removed from the lattice. When an empty concept is removed, we connect all its children with all its parents to keep the order structure.

After the lattice pruning, the remaining patterns are analyzed for simplification purpose. We noticed that some patterns contain a deep premise or conclusion, *i.e.*, a long chain of linked objects. After observing many patterns of this type for many transformation problems, we found that after a certain depth, the linked elements are not useful. For instance, if we look at the pattern `TPatt_5-Concept_70` in Figure 4, the important information is that a property linked to a class must be transformed into a column linked to a table. The other elements are details specific to some examples, that are not relevant to the transformation. Starting from this observation, we implemented a simplification heuristic that prunes the premises and conclusions after the first level (key element and its immediate neighbors).

After pruning the patterns according to the depth heuristic, some patterns could become identical. This is the case of patterns `TPatt_2-Concept_57` and `TPatt_5-Concept_70`. For both, only `Property_Class` and `Column_Table` are kept respectively in the premise and conclusion. For redundant patterns, just the top ranked in the lattice is preserved, and all others are automatically removed. For removed concepts, their children are linked to their parents.

4 From Transformation Patterns to Operational Rules

This section describes the mapping of transformation patterns into operational rules that can be executed using a rule engine. The rule engine used in our project is the Java Expert System Shell (*Jess*) [13]. In sub-section 4.1, this engine is introduced. Then, in sub-section 4.2, the transformation of patterns into Jess rules is detailed.

4.1 Jess

Jess is a rule engine integrated in the Java platform. Java code can be referred by Jess code [5]. With Jess, we can create Java objects, implement Java interfaces, and call Java objects from its Java scripting environment. Despite this, Jess is mainly a declarative language.

A Jess program is usually composed of *facts* and *rules*. Facts encode data, while rules, activated by pattern matching, encode behavior. [13]. A rule contains conditions, called left-hand-side (LHS), and actions, called right-hand-side (RHS). When the condition part is satisfied, the action part is executed. Conditions mainly test the presence of facts, whereas actions produce facts. Syntactically, a Jess rule is written as follows:
IF$<$ (fact1)(fact2)...(factN) $>$ THEN $<$(action1)(action2)...(actionM)$>$

The following example describes a very simple Jess rule which displays the name of each person who has a name.

```
(defrule welcome
  (Person (firstname ?name))
 =>
  (printout t "Hello" ?name "!!!" crlf)
)
```

The conditions in LHS and facts conform to a *template*. A template in Jess is similar to a class in Java. It defines a fact type. A template has a name and a set of slots. A fact, *i.e.* a template instance, has specific values for these slots. The example below shows the declaration of *Person* template:

```
(deftemplate Person (slot firstname))
```

This example declares a template named *Person* with a property *firstname*. To instantiate a person fact, we use the command *assert*:

```
(assert (Person (firstname Peter)))
```

4.2 Patterns to Jess Rules Transformation

In our context of model transformation, facts are model elements and templates are element types defined in the metamodel. A UML class diagram metamodel defines a set of templates such as `Class`, `Attribute`, and `Association`. A specific UML class diagram is described using facts that are instances of these templates such as, `Class Employee`, `Class Position`, and `Association has_position`. Fact `Class Employee` means that the model contains an element "Employee" which is an instance of the type "Class" in the metamodel.

Figure 5 illustrates the steps to follow in order to obtain operational rules from transformation patterns. The transformation process consists of three steps: Meta-model2Templates, Model2Fact, and TransformationPatterns2JessRules.

Meta-models2Templates. Step 1 consists in generating templates from the meta-models. Each metaclass of the metamodel is transformed into a template with the same name. Each meta-attribute is also transformed into a *slot* keeping the same name. The type of the slot is the type of the meta-attribute. To facilitate the description of relations between the metaclasses, each meta-reference is also transformed into a template. Such a template has two slots respectively containing the name of the source element and the target element of the meta-reference. We suppose that the name of each element is its identifier.

Concretely, since we work with the EMF framework, this step corresponds to the following transformations:

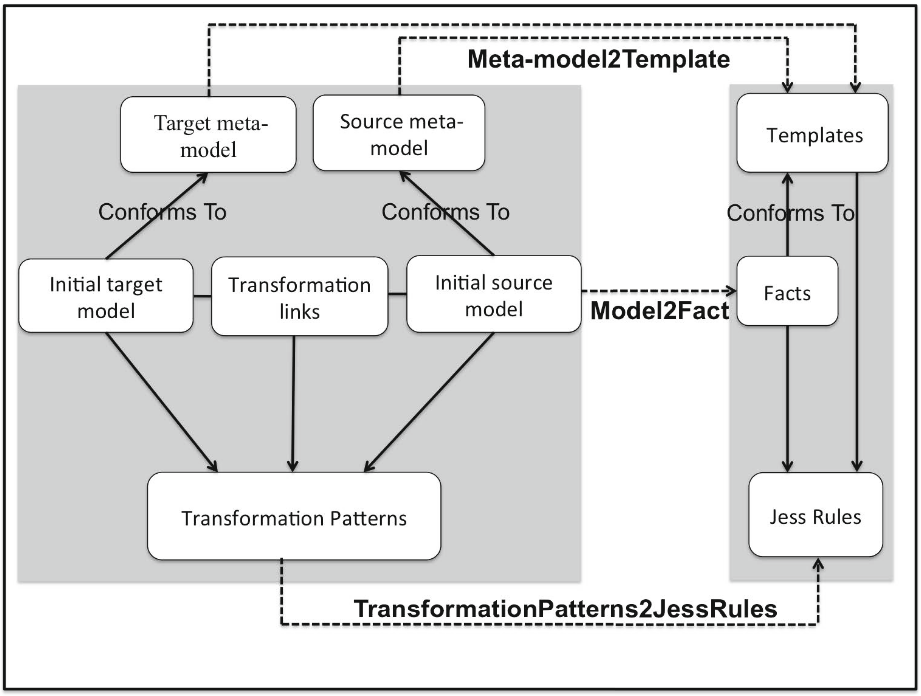


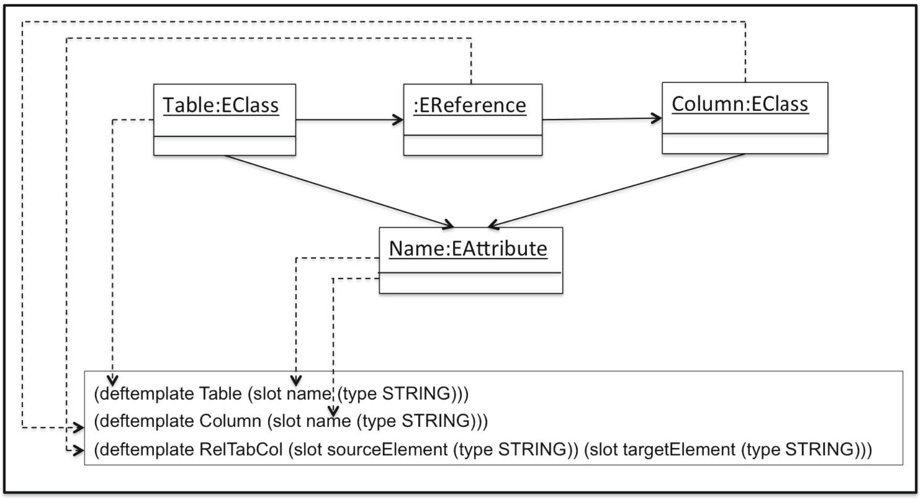
Fig. 5. Transformation Process

– each *EClass* is transformed into a template with the same name,
– each *EAttribute* is transformed into a *slot* with the same name and whose type is the *EDataType* of the *EAttribute*,
– each *EReference* is transformed into a template.

Figure 6 shows the transformation of a partial view of the relational schema meta-model. As indicated by the arrows, the *EClasses* table and column are transformed into templates. The *EAttribute* name is also converted to slot in each template. The *EReference* between table and column is transformed to a template which contains two slots containing the names of source and target elements of the *Ereference*.

Models2Facts. Step 2 aims at transforming models into facts. A model is an instantiation of its meta-model. Accordingly, each instance of a meta-class present in the model is transformed into a fact the same name. The instances of meta-attributes are transformed into slot values of the corresponding template. Each instance of meta-reference between two instances of meta-classes is also transformed into a fact which contains the names of relation elements.

A simple transformation example is presented in Figure 7. The three instances of metaclasses (the table and the two columns) are transformed into three facts. The two instances of meta-relations (from table to column) are transformed into the two facts instantiating the template *RelTabCol*.



```
(deftemplate Table (slot name (type STRING)))
(deftemplate Column (slot name (type STRING)))
(deftemplate RelTabCol (slot sourceElement (type STRING)) (slot targetElement (type STRING)))
```

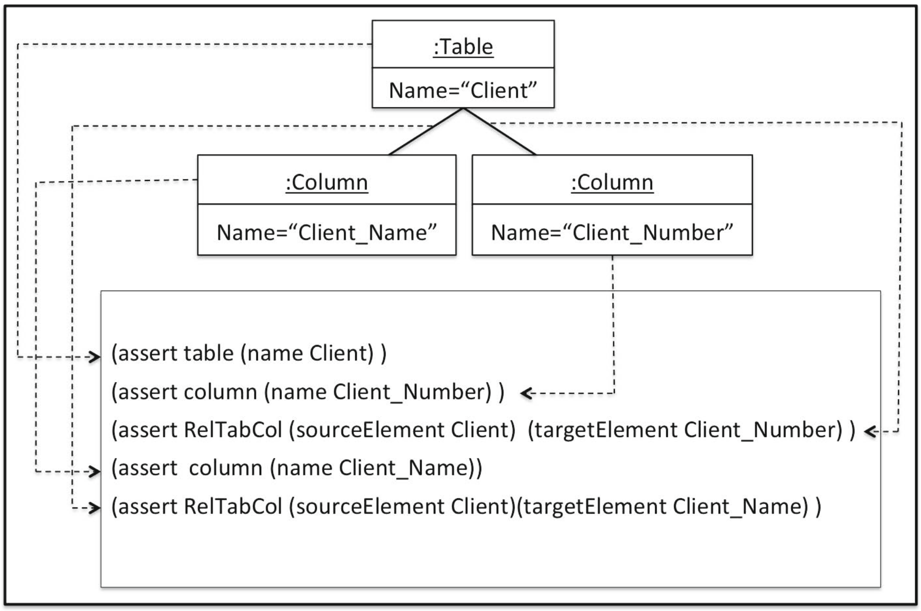
Fig. 6. Transformation of an extract of relational meta-model to Jess

TransformationPatterns2JessRules. Step 3 consists in the actual rule generation from transformation patterns. As it can be seen in Figure 8, there is a similarity between transformation-pattern structure and Jess-rule structure. Both of them are composed of two main parts. The premise of a pattern is equivalent to the LHS of a rule. Both describe the situation to find to fire the rule or to apply the transformation pattern. Similarly, the conclusion is equivalent to the RHS. Both are the action to perform or the conclusion to reach when the first part is satisfied.

The premise is a description of a set of source elements. These elements are linked together. Consequently, each element in the premise is transformed into a Jess condition corresponding to the test of the presence of a fact. As the premise elements are not named, we generate a slot name for each element. When more than one element is involved, conditions corresponding to relations are also generated. As relations do not have names, we named it by concatenating the three first letters of the relation elements names.

The conclusion of a transformation pattern is a description of a set of target elements together with their relations. It it similar to the premise. Consequently, each element in the conclusion is transformed into a Jess fact assertion. Names and relations between facts are also generated.

Figure 8 shows the transformation into a Jess rule of an example of transformation pattern. The premise of the transformation pattern is a class linked to a property. The corresponding Jess rule has for LHS four conditions, respectively checking: the existence of a class i, the existence of a property j, the existence of a relation from class to property, and that the existing relation links i to j. The conclusion of the transformation pattern is a table linked to a column. The corresponding RHS of the generated Jess rule contains three fact assertions, respectively stating: a table i, a column j, and a relation from i to j.



```
(assert table (name Client) )
(assert column (name Client_Number) )
(assert RelTabCol (sourceElement Client)  (targetElement Client_Number) )
(assert  column (name Client_Name))
(assert RelTabCol (sourceElement Client)(targetElement Client_Name) )
```

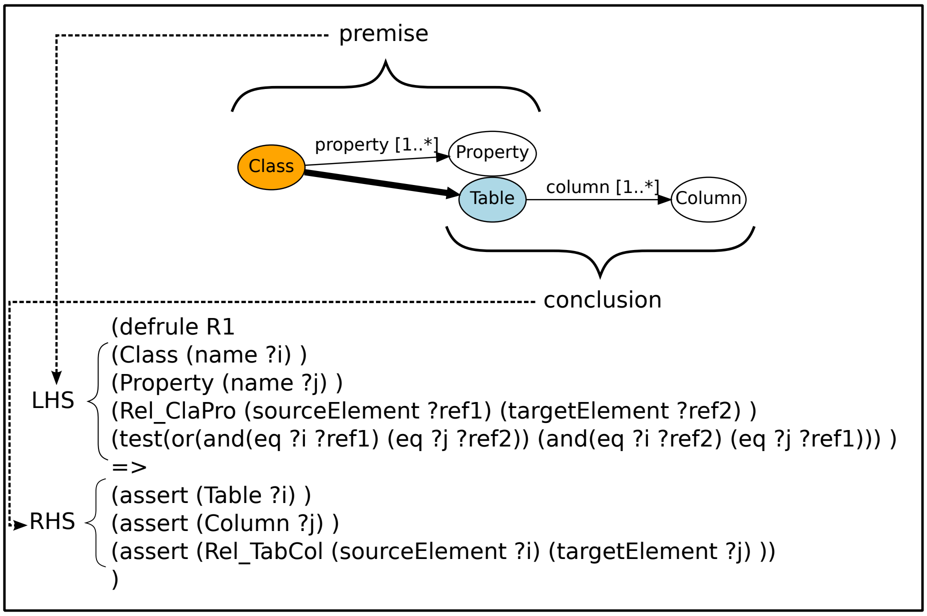
Fig. 7. Transformation of a partial view of relational schema model to Jess

5 Case Study

This section illustrates the rule generation process using a case study. It also reports on the efficiency of our approach through classical precision/recall measures. Like for testing, we compare the target models produced by our executable rules with the expected models. Precision and recall show to what extent the inferred rules perform the correct transformations.

Our case study concerns the transformation of class diagrams into relational schemas. The rule generation is performed starting from a set of 30 examples of class diagrams and their corresponding relational schemas. Some of them were taken from [16], the others were collected from different sources on the Internet. We ensured by manual inspection that all the examples conform to valid transformations.

To take the best from the examples, a 3-fold cross validation was performed, *i.e.*, 30 examples divided into three groups of 10. For each fold, two groups (20 examples) were used for generating the rules, and the remaining third group was used for testing them. Each fold used a different group for testing. Testing consists in executing the generated rules on the source models of the testing examples and in comparing the obtained target models with those provided in the examples. This comparison allows calculating the precision (Equation 1) and the recall (Equation 2) measures.



```
(defrule R1
(Class (name ?i) )
(Property (name ?j) )
(Rel_ClaPro (sourceElement ?ref1) (targetElement ?ref2) )
(test(or(and(eq ?i ?ref1) (eq ?j ?ref2)) (and(eq ?i ?ref2) (eq ?j ?ref1))) )
=>
(assert (Table ?i) )
(assert (Column ?j) )
(assert (Rel_TabCol (sourceElement ?i) (targetElement ?j) ))
)
```

Fig. 8. Example of the transformation of a pattern into Jess

We calculate precision and recall separately for each type T of fact (table, column, etc.).

$$P(T) = \frac{number\ of\ T\ with\ correct\ transformation}{total\ number\ of\ initial\ T} \tag{1}$$

$$R(T) = \frac{number\ of\ T\ with\ correct\ transformation}{total\ number\ of\ generated\ T} \tag{2}$$

Table 1 shows precision and recall averages (on all fact types) of the 10 generated transformations for the 3-folds. The precision and recall averages are higher than 0.70 in all cases. Some models were perfectly transformed (precision=1 and recall=1). For the others, the precision and recall could be better than the ones calculated automatically. This is due to the case of elements which have more than one transformation possibility. For example, if we have a generalization between two classes, we can transform it into a simple table which contains the attributes of general and specific classes. The second transformation method is to transform it into two tables. So, in the case of generalization, two rules are applied and this decreases the precision and the recall. The same problem exists for the aggregation which has also two transformation possibilities (1 or 2 tables).

Table 1. Result of 3-fold cross validation

Examples	Fold1	
	Precision Average	Recall Average
1	1	1
2	0.77	0.75
3	0.70	0.75
4	0.94	0.75
5	1	1
6	1	0.77
7	0.88	0.77
8	1	0.77
9	0.90	0.77
10	0.90	0.85

Examples	Fold2	
	Precision Average	Recall Average
1	0.78	0.79
2	0.90	0.75
3	0.85	0.77
4	0.77	0.79
5	1	0.80
6	1	0.77
7	0.85	0.77
8	0.85	0.80
9	1	0.75
10	1	0.80

Examples	Fold3	
	Precision Average	Recall Average
1	0.80	0.75
2	1	1
3	1	0.85
4	1	0.80
5	0.77	0.75
6	1	0.77
7	1	1
8	1	0.80
9	0.85	0.77
10	0.88	0.80

Discussion

The study presented in this section is a first evaluation of our approach. This evaluation is a proof-of-concept to check if RCA-based derivation and pattern-to-rule mapping are effective. In this context, the obtained results are very satisfactory. They show that the proposed approach allows to find most of the expected transformation rules and that these rules are executable on actual models.

To help us improving the rule generation process, additional experiments have to be conducted, in particular to study the two following issues:

– First, we used a small number of examples, based on small meta-models. Larger meta-models and more numerous examples have to be considered in the future to draw a better portrait on the strengths and weaknesses of the approach.
– Second, we measured the correctness of the obtained model transformation by comparing elements of the produced and expected models without considering their relations. A better and comprehensive correctness measure should be defined in the future.

6 Related Work

Writing model transformations requires time and specific skills: the transformation developer needs to master the transformation language and both transformation source and target meta-models. To our best knowledge, two main tracks have been explored to assist the process of developing a model transformation: using only source and target meta-models linked by the transformation, or using transformation examples.

A first approach is based on meta-model alignment and is inspired by research on ontology alignment and schema alignment. Transformation patterns are then deduced from this alignment. Lopes et al. [22,21] define a two-step process: the alignment algorithm SAMT4MDE computes alignments using a similarity metric on elements with the same type (classes, enumerations, etc.), then the tool MT4MDE generates a model transformation skeleton in ATL language [14]. Del Fabro et Valduriez [6] generate a transformation as a post-processing of a *weaving model*. This weaving model is built using a similarity metric between the elements and propagating similarities thanks to the Similarity Flooding algorithm [23]. Falleri et al. [9] study several configurations for applying Similarity Flooding algorithm in the context of meta-model alignment with the aim of determining which configurations work best. Kappel et al. [15] transpose their meta-models into ontologies and apply COMA++ tool [1]. Alignments on ontologies are brought back to the meta-models.

Meta-model alignment is especially relevant when the source and target meta-models are semantically and structurally closed, *e.g.* when the transformation aims at migrating models from one meta-model version to another, but is inefficient on complex cases. When it can be applied, meta-model alignment reduces significantly the time of the development. Other approaches (MTBE for Model Transformation Based Example) take advantage of transformation examples to learn transformations in more complex cases. One of their strengths is that transformation examples, written in the concrete syntax, are easier to manipulate than meta-models and their creation can be deferred to domain experts who don't need any programming skill.

The MTBE approach has been initiated by Varró [28]. An alignment between representative source and target example models is manually created. Transformation links are annotated by the transformation rule they illustrate (*e.g. ClassToEntity*). Transformation rules are derived from the transformation links and refined by the developer. Rules are validated on new source and target example models. If they are not satisfactory, the process iterates. The proposal of [28] was extended in [2], by using inductive logics programming (ILP [24]) to derive the transformation rules. ILP is a machine learning technique which derives a logic program from existing knowledge (source and target models), positive examples (pairs of model elements connected by transformation links) and negative examples (pairs of model elements that are not connected by transformation links). Considering only the immediate neighbors of each transformation-link end, the ILP engine infers an hypothesis for each transformation rule.

Wimmer et al. [29] propose a similar work but derive ATL transformation rules from examples written in concrete syntax by taking advantage of the constraints explicitly applied by the transformation from the concrete syntax of a language to its abstract syntax. The main advantage of this solution is to be able to use the concrete syntax to define models and transformation links. However, model editors need to be written in a way that permits to extract constraints and to edit transformation links.

Contributions [2] and [29] generate abstract rules and not executable ones. Although abstract rules could be individually correct, they are not a full-edged transformation program. These rules represent fragments of knowledge and must be arranged in a non-trivial way to perform the actual transformation (execution control). Furthermore, concrete rule languages and engines have their own constraints, which make the implementation of abstract rules not straightforward. In this paper, we produce executable rules and test them on real cases.

The work of Garcia-Magarino et al. [11] is also considered as a variant of MTBE approaches. In their approach, the authors generate transformation rules from meta-models which satisfy some developer constraints.

Another MTBE approach [7,8] uses an extension of the anchorPrompt approach [26] to assist the transformation link discovery, and Relational Concept Analysis to derive commonalities between the source and target meta-models, models and transformation links. Compared to the ILP-based proposal, the RCA-based approach does not use annotations on transformation links and proposes a set of transformation patterns organized in a lattice. However, the transformation patterns cannot be directly executed, and this paper proposes to translate them into JESS rules to provide consistency checking and executability.

Model Transformation By Demonstration (MTBD) [20,27], is a similar approach to MTBE. Through direct editing of the source model, users are asked to demonstrate how the model transformation should be done. The recorded actions are then generalized to produce transformation patterns.

Another track in MTBE consists in using the analogy to perform transformations using examples [17,18,19]. The provided examples are decomposed into transformation blocks linking fragments of source models to fragments of target models. When a new source model has to be transformed, its elements are compared to those in the example source fragments to select the similar ones. Blocks corresponding to the selected fragments, coming from different examples, are composed to propose a suitable transformation. Fragment selection and composition are performed through a meta-heuristic algorithm. Compared to the above-mentioned approaches, the analogy-based MTBE does not produce rules. This could be considered as a limitation if the goal is to infer reusable knowledge about transformations.

7 Conclusion

In this paper, we presented an approach that aims at deriving model transformation rules from a set of model transformation examples. A first step of the approach uses a data analysis method, RCA, to learn recurrent transformation patterns. In the second step, the transformation patterns are filtered, refined, and automatically transformed into Jess rules. Those rules constitute the expected transformation. Provided that meta-models and models are written as Jess facts (which is done by automatic transformation), the rules can be executed by the Jess engine to actually transform models. The approach is successfully evaluated on a case study used in previous research work.

Future work includes transforming the obtained Jess facts (after rule application) to produce models conforming to the initial meta-models. Furthermore, we plan to work on rule execution control to select the rule to apply when we have more than one rule for the same source element.

References

1. Aumueller, D., Do, H.H., Massmann, S., Rahm, E.: Schema and ontology matching with coma++. In: Özcan, F. (ed.) SIGMOD Conference, pp. 906–908. ACM (2005)
2. Balogh, Z., Varró, D.: Model transformation by example using inductive logic programming. Software and Systems Modeling 8(3), 347–364 (2009)
3. Bézivin, J., Dupé, G., Jouault, F., Pitette, G., Rougui, J.E.: First experiments with the atl model transformation language: Transforming xslt into xquery. In: 2nd OOPSLA Workshop on Generative Techniques in the Context of Model Driven Architecture (2003)
4. Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: Viatra: Visual automated transformations for formal verification and validation of uml models. In: Proceedings of the 17th IEEE International Conference on Automated Software Engineering. IEEE Computer Society (2002)
5. Daniele, L.M.: Towards a Rule-based Approach for Context-Aware Applications. Ph.D. thesis, University of Twente The Netherlands (May 2006)
6. Del Fabro, M.D., Valduriez, P.: Semi-automatic model integration using matching transformation and weaving models. In: International Conference SAC 2007, pp. 963–970. ACM (2007)
7. Dolques, X., Dogui, A., Falleri, J.R., Huchard, M., Nebut, C., Pfister, F.: Easing Model Transformation Learning with Automatically Aligned Examples. In: France, R.B., Kuester, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 189–204. Springer, Heidelberg (2011)
8. Dolques, X., Huchard, M., Nebut, C.: From transformation traces to transformation rules: Assisting model driven engineering approach with formal concept analysis. In: Supplementary Proceedings of ICCS 2009, pp. 15–29 (2009)
9. Falleri, J.R., Huchard, M., Lafourcade, M., Nebut, C.: Metamodel Matching for Automatic Model Transformation Generation. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 326–340. Springer, Heidelberg (2008)
10. Ganter, B., Wille, R.: Formal concept analysis - mathematical foundations. Springer (1999)
11. García-Magariño, I., Gómez-Sanz, J.J., Fuentes-Fernández, R.: Model Transformation By-Example: An Algorithm for Generating Many-to-Many Transformation Rules in Several Model Transformation Languages. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 52–66. Springer, Heidelberg (2009)
12. Huchard, M., Hacène, M.R., Roume, C., Valtchev, P.: Relational concept discovery in structured datasets. Ann. Math. Artif. Intell. 49(1-4), 39–76 (2007)

13. Jess rule engine, http://herzberg.ca.sandia.gov/jess
14. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
15. Kappel, G., Kapsammer, E., Kargl, H., Kramler, G., Reiter, T., Retschitzegger, W., Schwinger, W., Wimmer, M.: Lifting Metamodels to Ontologies: A Step to the Semantic Integration of Modeling Languages. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 528–542. Springer, Heidelberg (2006)
16. Kessentini, M.: Transformation by Example. Ph.D. thesis, University of Montreal (2010)
17. Kessentini, M., Sahraoui, H., Boukadoum, M.: Model Transformation as an Optimization Problem. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 159–173. Springer, Heidelberg (2008)
18. Kessentini, M., Sahraoui, H., Boukadoum, M.: Méta-modélisation de la transformation de modéles par l'exemple: approche méta-heuristiques. In: Carré, B., Zendra, O. (eds.) LMO 2009: Langages et Modéles á Objets, Cepadués, Nancy, pp. 75–90 (March 2009)
19. Kessentini, M., Sahraoui, H., Boukadoum, M., Ben Omar, O.: Model transformation by example: a search-based approach. Software and Systems Modeling Journal (2010) (to appear)
20. Langer, P., Wimmer, M., Kappel, G.: Model-to-Model Transformations By Demonstration. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 153–167. Springer, Heidelberg (2010)
21. Lopes, D., Hammoudi, S., Abdelouahab, Z.: Schema matching in the context of model driven engineering: From theory to practice. In: Sobh, T., Elleithy, K. (eds.) Advances in Systems, Computing Sciences and Software Engineering, pp. 219–227. Springer (2006)
22. Lopes, D., Hammoudi, S., Bézivin, J., Jouault, F.: Generating Transformation Definition from Mapping Specification: Application to Web Service Platform. In: Pastor, Ó., Falcão e Cunha, J. (eds.) CAiSE 2005. LNCS, vol. 3520, pp. 309–325. Springer, Heidelberg (2005)
23. Melnik, S., Garcia-Molina, H., Rahm, E.: Similarity flooding: A versatile graph matching algorithm and its application to schema matching. In: ICDE, pp. 117–128. IEEE Computer Society (2002)
24. Muggleton, S., De Raedt, L.: Inductive logic programming: Theory and methods. Journal of Logic Programming 19(20), 629–679 (1994)
25. Muller, P.-A., Fleurey, F., Jézéquel, J.-M.: Weaving Executability into Object-Oriented Meta-languages. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 264–278. Springer, Heidelberg (2005)
26. Noy, N.F., Musen, M.A.: Anchor-prompt: Using non-local context for semantic matching. In: Proc. of the Workshop on Ontologies and Information Sharing at IJCAI 2001, Seattle (USA), pp. 63–70 (2001)
27. Sun, Y., White, J., Gray, J.: Model Transformation by Demonstration. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 712–726. Springer, Heidelberg (2009)
28. Varró, D.: Model Transformation by Example. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 410–424. Springer, Heidelberg (2006)
29. Wimmer, M., Strommer, M., Kargl, H., Kramler, G.: Towards model transformation generation by-example. In: HICSS, p. 285 (2007)

Using Feature Model to Build Model Transformation Chains

Vincent Aranega[1], Anne Etien[1], and Sebastien Mosser[2]

[1] LIFL CNRS UMR 8022 Université Lille 1 - France
`firstname.lastname@lifl.fr`
[2] SINTEF IKT, Norway
`sebastien.mosser@sintef.no`

Abstract. Model transformations are intrinsically related to model-driven engineering. According to the increasing size of standardised meta-model, large transformations need to be developed to cover them. Several approaches promote separation of concerns in this context, that is, the definition of small transformations in order to master the overall complexity. Unfortunately, the decomposition of transformations into smaller ones raises new issues: organising the increasing number of transformations and ensuring their composition (*i.e.* the *chaining*). In this paper, we propose to use *feature models* to classify model transformations dedicated to a given business domain. Based on this feature models, automated techniques are used to support the designer, according to two axis: *(i)* the definition of a valid set of model transformations and *(ii)* the generation of an executable chain of model transformation that accurately implement designer's intention. This approach is validated on Gaspard2, a tool dedicated to the design of embedded system.

1 Introduction

Model-Driven Engineering (MDE) advocates the principle of *separation of concerns*, through the extensive use of models in all the steps of the software development cycle [12,18]. In this context, model transformations are used to achieve *integration of concerns* [14,17,3]. Considering the intrinsic complexity of the meta-models in use (*e.g.*, UML 2.x and its profiles), large model transformations (up to ten thousands lines of code) are developed. Such transformations have substantial drawbacks [15], including limited reusability, reduced scalability, poor separation of concerns, limited learnability, and undesirable sensitivity to changes. The separation of concerns paradigm advocates the decomposition of a complex system (*e.g.*, architectures, object-oriented models) into smaller artefacts. Thus, exactly as other artefacts, it is desirable to *decompose* transformations [20]. Other researches have also argued that focusing on such an engineering of transformations improves the uptake of MDE [22]. It is then essential to support the systematic definition of small model transformations with a unique intention [5], to improve scalability, maintainability and reusability of transformations. Such an approach leads to the definition of a family of transformations associated to a given domain that jointly enable to generate systems from a business domain.

The existence of small transformations raises two new issues. First, the chain designer (called *end user* in the remainder of the paper) is in presence of a family of model transformations, which needs to be organised. Secondly, the reification of the dependencies that exist between elements of this family becomes critical. As model transformations cannot be chained anyhow, dependencies that lead to *valid* transformation chains must be captured. One way to automate this development process is to use a *Software Product Line* (SPL) approach. In a SPL, multiple products are *derived* by combining a set of different core assets. One of the most important challenges of SPL engineering concerns variability management, *i.e.*, how to describe, manage and implement the commonalities and variabilities existing among the members of the same family of products. A well-known approach to variability modelling is by means of *Feature Diagrams* (FD) introduced as part of *Feature Oriented Domain Analysis* [9] back in 1990.

Our contribution is to accurately combine model transformations and SPL to support the *end user* while developing transformation-based applications. Business experts' knowledge is reified in a FD to accurately organise the different transformations according to their intentions. Then, automated code analysis techniques are used to accurately generate constraints between these transformations[1], reified in the feature model as *requirements* between features. Thus, it is possible for *end users* to use the FD to accurately define their own products, that is, a valid subset of transformations that matches their intentions. Product derivation mechanisms are then used to automatically generate the model transformation chain that implements what the *end user* asked for. The approach is validated using Gaspard2, a transformation-based tool that supports the modelling of embedded systems.

The remainder of this paper is organised as follows. In Section 2, we motivate this work by exposing the different challenges that need to be addressed in this domain. Then, Section 3 describes the approach we propose to tackle these challenges. Section 4 validates the approach by applying it to the Gaspard2 case study. Finally, Section 5 discusses the related works and Section 6 concludes this paper by exposing some research perspectives.

2 Motivation

In order to enhance reusability, variability, flexibility and verifications, Gaspard2 [8], a co-design environment dedicated to high performance embedded systems based on massively regular parallelism has been designed using Model Driven Engineering (MDE) technologies. Thus it enables the generation of VHDL, SystemC, OpenMP or Lustre code from a UML model enhanced with the *Modelling and Analysis of Real Time Embedded systems* (MARTE) profile. Each

[1] Informally, a transformation τ requires a transformation τ' if the model elements handled by τ are produced by τ'.

language is targeted using a chain composed of three to five dedicated transformations. These large transformations (up to 1500 lines of codes) were not reusable and hardly maintainable even by their own developers.

To introduce flexibility and reusability, the Gaspard2 environment has been re-engineered to rely on smaller transformations. Each transformation has a single intention such as memory management or scheduling and corresponds to 150 lines of code in average. Finally, 19 transformations including 4 *model to text* (M2T) transformations, and thus 15 *model to model* (M2M) transformations were defined. The number of chains that can be constructed from them is humongous. Let $T = \{\tau_1, \ldots, \tau_n\}$ a set of model to model transformations, and $M = \{\mu_1, \ldots, \mu_m\}$ a set of model to text transformations. We denote as $N_{T \cup M}$ the number of chains available in this context. The number of potential model to model chains is equal to the number of sequences without repetition that involve elements defined in T (denoted as $P(k,n)$). Secondly, There is $(m+1)$ potential targets for the previously defined sub-chain (as a transformation chain may not generate text). Finally, it is also possible to only generate text without involving other model transformation (thus, m chains).

$$N_{T \cup M} = m + (m+1) \sum_{k=1}^{n} P(k,n), \qquad P(k,n) = \frac{n!}{(n-k)!}$$

$N_{T \cup P}$ is hardly computable generically. Nevertheless, a sub-optimal approximation is to consider $N_{T \cup P}$ bigger than $(m+1)$ times the highest term of the sum $P(k,n)$ (*i.e.*, $P(n,n)$, that in our case is equals to $5 \times 15!$).

$$N_{T \cup P} \gg (m+1) \times P(n,n) = (m+1) \times n!,\ n = 15, m = 4, N_{T \cup P} \gg 6,5 \times 10^{12}$$

But only a few chains make sense! It becomes crucial to help the designer to built such chains. Thus, the definition of transformation libraries raises new issues such as *(i)* the representation of the transformations highlighting their purpose and the relationships between them; *(ii)* their appropriate selection according to the characteristics of the expected targeted system and *(iii)* their composition in a valid order.

Traditionally, transformations are represented in chains or with their metamodels. Such representations are not adapted to the description of transformation libraries. In preparation for chaining the transformations, it seems indispensable to specify their purpose (*i.e.*, what they handle), in addition to their associated metamodels. To generate systems with their own characteristics (*e.g.*, management of distributed versus shared memory, optimised vs simple scheduling), transformations have to be consequently selected. Thus the *end user* has to select the transformations not only according to the characteristics of the resulting system she would like, but also to the relationships between the transformations. Manually performed, this selection may be tedious and error prone. From the selected transformations, several chains can be built. Transformations cannot be chained arbitrarily, some constraints must be fulfilled [7,11]. If it is often simple to identify the first transformation of the chain (depending on the input metamodels) and the last one (that is a model to text transformation, if code has to be generated), establishing a valid order between the other selected transformations may be difficult. Indeed, existing approaches check if the proposed order is valid, but do not automatically provide a valid one.

In order to support the *end user* in the design of transformation chains, the following challenges have to be addressed:

C_1. Propose to the *end user* a library in which each transformation can be easily identified according to the characteristics of the expected final system (Section 3.1).

C_2. Help the *end user* to select transformations while automatically taking into account the relationships between transformations (Section 3.2).

C_3. Automatically derive the transformation chain from the characteristics selected by the *end user* (Section 3.3).

3 Solution

To tackle the aforementioned challenges, we propose a feature-oriented approach and the associated too set to automatically generate accurate model transformation chains as depicted in Figure 1.

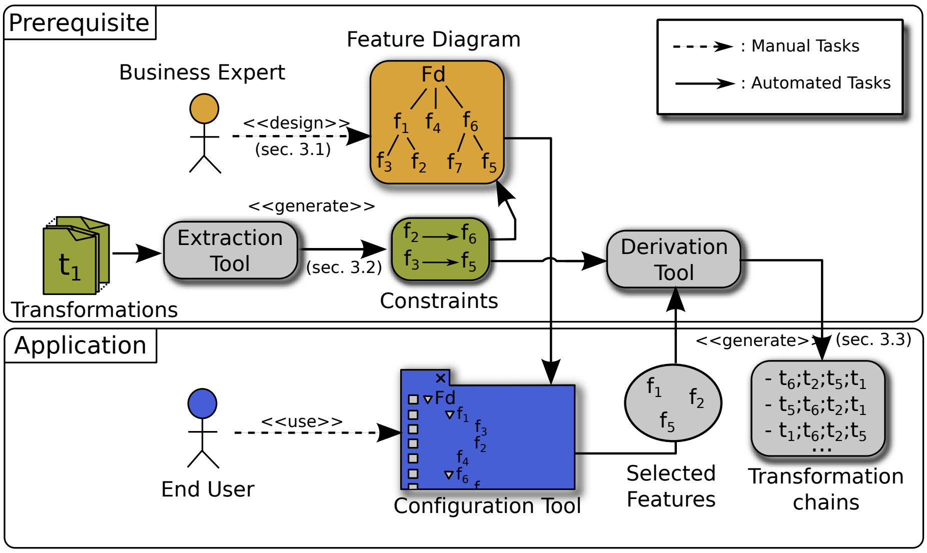


Fig. 1. Approach Process Overview

This approach relies on three pillars: *(i)* the classification of the available transformations as a *Feature Diagram* (FD) produced by the *business expert*, *(ii)* the reification of requirement relationships between transformation (directly generate from the *Transformations* set by the *Extraction Tool*) and *(iii)* the automated generation of transformation chains for a given product (using our *Derivation Tool*) from features selected by the *end user*.

The FD is designed once for all by the *business expert* as a prerequisite. It is nevertheless possible to modify it when new transformations and thus new features become available. The requirement relationships are expressed between the features and automatically computed from the transformation codes by the *Extraction Tool* we provide. The extracted relations enable to derive other features (and then the associated transformation) from the ones selected by the *end user* using a *Configuration Tool* (*e.g.*, FeatureIDE[2]). The requirement relationships are also used by our *Derivation Tool* to order the selected features in order to design valid chains.

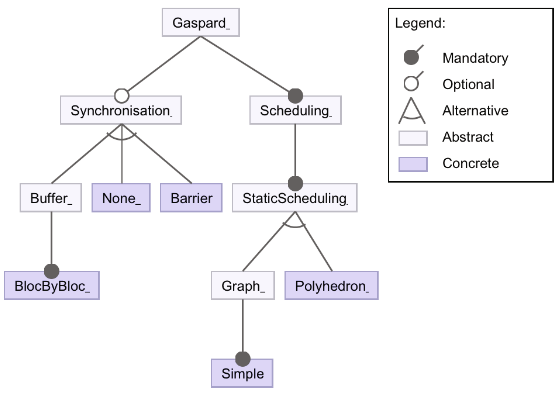
3.1 Structuring the Transformation Set as a Feature Diagram

As a transformation is used to support a given intention according to a business domain, a set of transformations implicitly model the variability of the different intentions associated to a domain. FD were defined to model such a variability, so its use is natural. We represent in Figure 2 an excerpt of the complete FD associated to Gaspard2. Using FD, features (represented as nodes) are classified among others according to constraints such as exclusiveness or optionality. Model transformations are bound to features as assets. A feature f holds a link to the actual model transformation to be used to implement the intention captured by f at run-time. Normally, each feature corresponds to a single transformation and vice versa. However, it occurs in practise that a single transformation may catch many intentions and thus corresponds to many features.

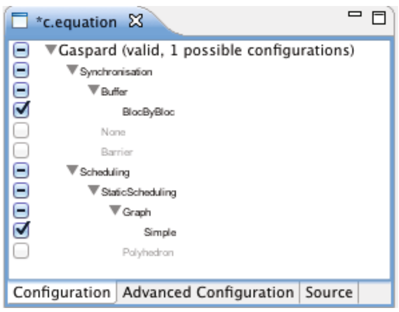
For example, in Figure 2(a), the FD models that a given product *must* contain a `Scheduling` feature, and *may* contain a `Synchronisation` feature. The features `Graph` and `Polyhedron` are exclusive, *i.e.*, the use of one in a given product implies that the other cannot be used in this particular product. We call a product a set of features that respects the constraints modelled in the FD. For example, Figures 2(b) and 2(c) represent two products among the eight valid *w.r.t.* the modelled FD. The first one (Figure 2(b)) considers a system synchronised using a `BlocByBloc` method, and scheduled with a simple `Graph`. The second product (Figure 2(c)) considers a system synchronised with a `Barrier` method, and scheduled with a `Polyhedron` approach. In our context, features reify model transformations: the actual implementation of the transformation is bound as an asset of the associated feature node. Thus, considering a given product, it is possible to automatically infer the set of transformations involved in the transformation chain that supports it.

Key Points. The role of the FD is to capture the business knowledge associated to a given set of transformations. It actually transforms a flat set of transformations into an organised family of products.This classification is done by the *business expert*, that is, someone who deeply knows the different transformations, their

[2] `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/`



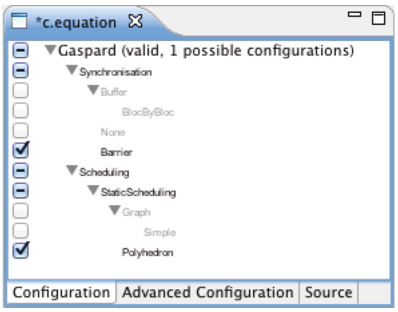
(a) Gaspard2 feature diagram (excerpt)


*c.equation

```
▼Gaspard (valid, 1 possible configurations)
  ▼Synchronisation
    ▼Buffer
      [✓] BlocByBloc
    None
    Barrier
  ▼Scheduling
    ▼StaticScheduling
      ▼Graph
        [✓] Simple
      Polyhedron
```

Configuration | Advanced Configuration | Source

(b) $p = \{BlocByBloc, Simple, \ldots\}$


*c.equation

```
▼Gaspard (valid, 1 possible configurations)
  ▼Synchronisation
    ▼Buffer
      BlocByBloc
    None
    [✓] Barrier
  ▼Scheduling
    ▼StaticScheduling
      ▼Graph
        Simple
      [✓] Polyhedron
```

Configuration | Advanced Configuration | Source

(c) $p' = \{Barrier, Polyhedron, \ldots\}$

Fig. 2. Gaspard2: Feature diagram and associated products (using FeatureIDE)

underlying intentions, as well as the artifact they are handling. The key idea here is that this work is done once by the *business expert*, and capitalised in the FD. Without the use of a FD to support such a classification, it would be up to the *end users* to guess how the different transformations cope with each others before assembling them.

3.2 Recovering Require Relationship from Transformations

On top of constraints expressing the mandatory/optional character of the features as well as the and/or relationships between them, "require" relationships can also be captured in FD. They enable to automatically deduced other features from the selected ones, independently of the tree structure of the FD. Require relationships can be determined manually by the *business expert*. However, when the number of features is huge, omission can happen leading to erroneous products determination. Therefore, we provide an automatic analysis of the transformations to recover the require relationships (bijection) between the features associated to them. A requirement between two features f and f' (denoted as a logical implication, *i.e.*, $f \Rightarrow f'$) means that the transformation bound to f requires the transformation bound to f'. The following question is raised: "*When does a require relationship between two transformations exists?*". In fact, it relies on the element type production and consumption. For two transformations τ and τ', if τ' consumes types created by τ, then it implies that a require relationship exists between τ and τ', denoted as $\tau' \rightarrow \tau$ (for τ' requires τ). For each transformation, it is thus mandatory to automatically determine the element types it produces and it consumes to provide an automatic require relationships determination.

This automatic analysis relies on the different actions performed on element types by a transformation. Four actions are classically performed by transformations: *reading*, *creating*, *deleting* and *modifying*. This analysis does not rely on transformation execution but on static code analysis. Thus an element of the input or the output metamodel of a transformation is considered read if the presence of one on its instance enables the application of a transformation rule. An element is considered created, if at least one of its instance can be created by the transformation, and so on. Thus, τ' requires τ if τ' reads some elements created by τ [6]. As we stated in the previous section, for a feature f from the FD, it exists at most one transformation τ. So, considering two features f, f' in the FD and two transformations τ, τ' mapped to f, respectively f', if $\tau \rightarrow \tau'$, it implies that $f \Rightarrow f'$.

Key Points. The proposed generation of the require relationships relies on a static analysis of the transformation codes. Once the FD designed and the constraints generated, the *end user* can use a *Configuration Tool* to select the features she wants for her transformation chain. The *Configuration Tool* is parametrised by the feature diagram and the generated constraints. Thus, by taking into account the generated constraints, during the feature selection, the *Configuration Tool* can either invalidate features or add required features according to the ones already selected by the *end user*. The automatic characteristic of the generation enables a certain evolutivity of the FD.

3.3 Generating Transformation Chains

Based on the two previous parts of the contribution, it is now possible to *(i)* consider a set of model transformations as a product family and *(ii)* automatically infer the requirement relationships that exist inside the product family. These two contributions act at the level of the FD. According to the global process, the selected features are then passed to a *Derivation Tool*, which uses the generated constraints to propose transformation chains from the selected features.

We consider now a given product $p = \{f_1, \ldots, f_n\}$, *i.e.*, a subset of features that satisfies the constraints modelled in the FD. As stated in Section 3.1, model transformations are bound to features. It is then possible to obtain the set of model transformations associated to p (denoted as T_p) by mapping each feature to its associated transformation: $T_p = \{\tau_1, \ldots, \tau_m\}$. As some features are only used to structure the FD and are not related to any concrete transformation, it should be noted that the cardinality of T_p may be lesser than the cardinality of p. But this set of transformations is not sufficient to properly derive a concrete transformation chain from a given product. The requirement constraints identified in Section 3.2 must be taken into account. Considering two features f and f', if the requirement $f \Rightarrow f'$ exists, then the transformation τ' mapped to f' must be executed before the transformation τ mapped to f. As a consequence, the analysis of the set of requirement constraints leads to the identification of sequences of model transformations. Two situations can be encountered. If the requirement constraints implement a total order on the set of transformations, only one sequence will be identified, *i.e.*, the proper transformation chain to be executed to support the intentions captured by this product. But if the requirement constraints implement a partial order, only *partial sequences* can be identified automatically. But as there is no requirement between these different sub-sequences, their order is not important. Consequently several valid chains are generated. This "subchains approach" is useful to support the *business expert* while assessing the FM consistency. It also helps the non-expert *end user* to construct a valid chain: any of the chains built upon these sub-chains, will by essence respect the dependencies captured by the FM.

Key Points. A concrete chain of model transformations is automatically derived, through the FD, from the transformation set selected by the *end user*. First, the knowledge of a *business expert* is captured in the FD, and then an automatic static analysis is used to properly extract technical constraints from the implementation of the transformations. Finally, it is possible to automatically derive the chain, through the systematic exploration of the identified constraints. As a consequence,the generation of the concrete model transformation chain is automated, and the *end user* does not require any knowledge of model transformation from a technical point of view.

4 Validation: The Gaspard2 Case Study

Gaspard2 is a co-design environment dedicated to high performance embedded systems based on massively regular parallelism. From high level specifications, it automatically generates code for high performance computing, hardware-software co-simulation, functional verification or hardware synthesis using model transformations. Such generations are complex and require intermediary steps, *e.g.*, the explicit mapping of application tasks onto processing units, the mapping of the data onto memories or the scheduling of the tasks. Each transformation has a specific intention and deals with few concepts. Nineteen transformations have been implemented for now but the framework may support even more of them in the upcoming months. It is difficult for a non expert user to easily understand the purpose of each transformation, to select the ones useful to reach the desired platform and finally to order them in order to compose a chain.

In this paper, we used the Familiar tool suite [1] to manipulate feature diagrams. This tool allows us to model FD, and is well integrated in the Eclipse platform. Thus, standard *Configuration Tools* (*e.g.*, FeatureIDE) can be used to allow the *end user* to configure products. But it should be noted that the approach is not bound to this tool nor to this case study from a theoretical point of view, as described in the previous section.

4.1 Step #1: Capturing Business Expert Knowledge in a FD

Embedded systems designers usually do not master model transformation paradigm and underlying technologies. It is then essential to support them while designing the transformation chains used to generate code from high level specifications. The design of these chains consists in the selection of relevant transformations available in a library and in the computation of a valid order. Selecting a transformation requires to easily distinguish one transformation from another and to quickly identify its intention. In order to help the embedded systems designers, we have classified the available transformations based on embedded characteristics using feature model. It is up to the *business expert* to find the most appropriate classification method to be used to support the *end user*. During the implementation of the case study, we applied an incremental definition of the FD. We produced 12 successive versions of the FD. Excepting from one deep refactoring to better handle business requirement constraint, the implementation of the FD by the *business expert* was straightforward.

Most transformations of Gaspard2 have a unique intention representing a specific characteristics of the produced systems such as memory management. The transformations and their associated intentions are listed in Table 1. The Gaspard2 transformation library counts 19 intentions through 15 M2M and 4 M2T transformations. For example, the *scheduling* transformation has the following intention: it manages a simple scheduling of application tasks on computing units. As a consequence of the non mandatory bijection between features and transformations, the *barrier* and the *openMP* features are implemented by a single transformation. This many-to-one (surjection) relationship corresponds to a lack of modularisation of the transformation.

From these intentions, the *business expert* builds the feature diagram by associating a feature to each intention. Moreover, some features are added in the hierarchy in order to specify the relationship AND/OR/XOR between features. Indeed, as stated in Section 3.1, each feature represents at most one transformation. The resulting FD, depicted in Figure 3, gathers, in an non exhaustive way, some characteristics that an embedded system produced by Gaspard2 may possess. For example, the `OpenCL` and `OpenMP` features, introduce a scientific computation intention. However, only one of these two features can be selected. Indeed, the target language is either OpenCL, or OpenMP. In the FD, this choice is designed by the introduction of an intermediary abstract node `ScientificComputation` and an alternative between the two features.

The associated tooling provided by the Familiar platform can be used to query the model, as shown in Figure 4. This FD models up to 200 different available configurations (obtained by the Familiar `counting` algorithm). The `configs` command computes all the available products, returning the set of valid products defined by this FD.

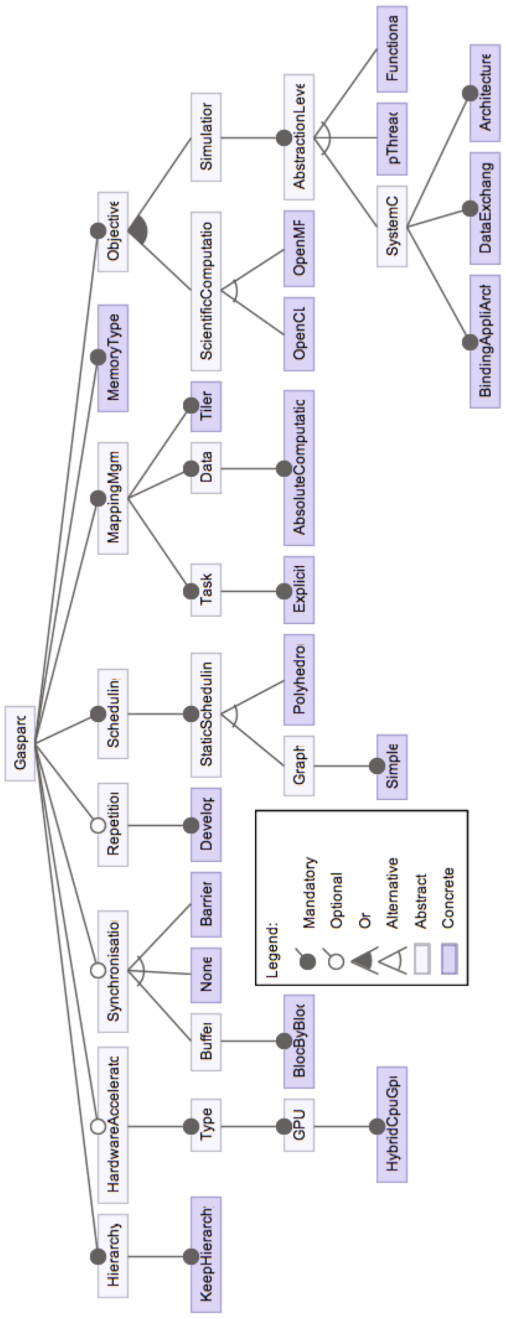
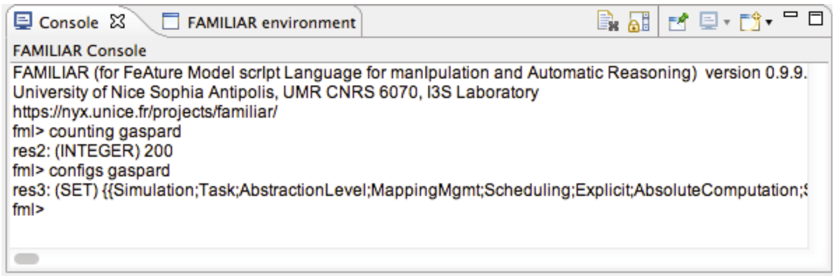


Fig. 3. Feature model associated to Gaspard2

Table 1. Gaspard2 transformation set

Transformation	Intention
tiler2task	- Keep repetitions `hierarchy`
gpuApi	- Manage `hybrid GPU-CPU` computing
pThread	- Manage `buffered synchronisation by bloc`
sequentialC	- Generate `sequential` C code
barrier	- Manage `barrier synchronisation` for `OpenMP`
shape2loop	- `Develop repetitions` in the generated systems
scheduling	- Manage `simple` scheduling
poly_loop	- Manage `polyhedron` optimised `scheduling`
explicitAllocation	- `Explicitly` place tasks on processors
memorymapping	- Manage `absolute` memory `addresses`
tilerMapping	- Manage `tiler` (*i.e.* task distributing data) mapping on computing unit
shared	- Manage the shared `memory type`
openCL	- Generate `OpenCL` code for `scientific computation` purposes
openMP	- Generate `OpenMP` code for `scientific computation` purposes
systemcPA	- `Bind` SystemC `architecture` with SystemC `application`
systemcBind	- Manage SystemC `data exchanges`
systemcStruct	- Manage SystemC `architecture`
pthreadGen	- Generate `pthread` code for simulation purposes
functional	- Introduce `functional` abstraction



```
FAMILIAR (for FeAture Model scrIpt Language for manIpulation and Automatic Reasoning)  version 0.9.9.
University of Nice Sophia Antipolis, UMR CNRS 6070, I3S Laboratory
https://nyx.unice.fr/projects/familiar/
fml> counting gaspard
res2: (INTEGER) 200
fml> configs gaspard
res3: (SET) {{Simulation;Task;AbstractionLevel;MappingMgmt;Scheduling;Explicit;AbsoluteComputation;
fml>
```

Fig. 4. Using the Familiar shell to interact with the FD

4.2 Step #2: Extracting Constraints from the Implementation

This feature model enables the classification and the distinction of the transformations the one from the others. However, in this primary form, it does not gather enough information to build the chains: some others may be required and the selection of one transformation may require the selection of others. Such dependencies between transformations have to be captured and the feature model tools enable to take them into account for the product configuration. Thanks to the *Extraction Tool*, the implementation of the available transformations is automatically analysed. The result of this analysis is a set of "require" constraints between the features modelled in the FD. We represent in Listing 1.1 the set of constraints obtained after the execution of the tool. These constraints are generated using the syntax of the Familiar tool, and thus can be automatically integrated in the FD. Contrarily to the initial FD that captures the knowledge of the *business expert*, these relations reify the implementation constraints that exist between the transformations, from a technical point of view. It then ensures that the products configured *w.r.t.* this FD will be valid at both level: *(i)* business domain and *(ii)* technical implementation.

```
1   AbsoluteComputation -> Develop
2   AbsoluteComputation -> KeepHierarchy
3   AbsoluteComputation -> Polyhedron
4   BindingAppliArchi ->
         AbsoluteComputation
5   BindingAppliArchi -> Architecture
6   BindingAppliArchi -> BlocByBloc
7   BindingAppliArchi -> MemoryType
8   BlocByBloc -> AbsoluteComputation
9   BlocByBloc -> Graph
10  BlocByBloc -> KeepHierarchy
11  BlocByBloc -> MemoryType
12  DataExchange -> Architecture
13  DataExchange -> KeepHierarchy
14  Functional -> Graph
15  Graph -> KeepHierarchy
16  Hybrid -> AbsoluteComputation
17  Hybrid -> Graph
18  Hybrid -> KeepHierarchy
19  Hybrid -> MemoryType
20  MemoryType -> KeepHierarchy
21  Simple -> Graph
22  Tiler -> Graph
```

Listing 1.1. Set of requirement constraints

Considering this set of constraints, the *Configuration Tool* now proposes 37 available products to the *end user* (from 200 at the beginning). This highlights the fact that working with the implementation of the transformation is critical. The technical implementation of the transformations dramatically reduces the initial variability of the domain as it was designed by the *business expert*.

Taking into account the "real" features implementations in the FD (*i.e*, the transformations code in our context) through this set of automatically computed constraints also leads to interesting situations that help the *business expert*. We consider here the feature `Repetition`, defined as optional by the *business expert* (see Figure 3). The generated set of constraints identifies a requirement between the feature `AbsoluteComputation` and the feature `Develop` (line 1 in Listing 1.1). However, `AbsoluteComputation` is mandatory, and selecting `Develop` implies to select `Repetition`. Thus, the `Repetition` feature is automatically identified by the tool suite as a *false optional* feature, that is, a feature modelled as optional but enforced as mandatory by a requirement constraints. In this case, it helped the *business expert* to identify a missing artifact in the FD: it should also contain an alternative implementation for `Repetition` instead of only defining the `Develop` approach.

4.3 Step #3: Deriving Transformation Chains

Based on the FD enhanced with the implementation constraints, we can now ensure that the products configured by the *end user* through the configuration tool are valid. The final step is to use a derivation tool that properly builds the transformation chains associated to a given product. We consider here one of the 37 products available according to this FD, denoted as p corresponding for example to the set of the features selected by the *end user*. The first step is to translate p into T_p, that is, the set of transformations involved by this product. It should be noted that $|p| > |T_p|$, as several features are only used to structure the FD and consequently are not bound to concrete transformations. For example, for the following product, corresponds the associated T_p:

$$\begin{aligned} p = \{&Gaspard, MemoryType, Polyhedron, Data, Barrier, MappingMgmt,\\ &KeepHierarchy, Hierarchy, Tiler, Develop, StaticScheduling,\\ &AbsoluteComputation, Task, Explicit, ScientificComputation,\\ &Scheduling, Objective, Repetition, Synchronisation, OpenMP\}\\ T_p = \{&explicitAllocation, memMapping, openMP, poly\_loop,\\ &shape2loop, tilerMapping, tiler2task\} \end{aligned}$$

The second step is to map the constraints between features as a partial order among the transformations. The requirements involved in p are the following:

$$\begin{array}{rl} \textbf{Feature Requirement} \rightsquigarrow & \textbf{Transformation Ordering}\\ AbsoluteComputation \rightarrow Develop \rightsquigarrow & memMapping \rightarrow shape2loop\\ AbsoluteComputation \rightarrow Polyhedron \rightsquigarrow & memMapping \rightarrow poly\_loop\\ MemoryType \rightarrow KeepHierarchy \rightsquigarrow & memMapping \rightarrow tiler2task \end{array}$$

Based on this partial order, it is possible to compute[1] the following sets of "independent" sub-chains involved in this product, as a chain template, that is, a partition of the transformation set taking into account the partial order:

$$\begin{aligned} tpl_p = [\ &[openMP], [explicitAllocation], [tilerMapping] && (1)\\ &[memMapping, [shape2loop, poly\_loop, tiler2task]\ ]\ ] && (2) \end{aligned}$$

Among the computed sub-chains, the *openMP* transformation is a "model to text" transformation and will always be the last one executed in the chain. In line 2, the partial order indicates that the *memMapping* transformation must be preceded by the 3 transformations listed, without specifying any order between them. Thus, there is up to 6 ways to combine these transformations according to this constraint. As the *explicitAllocation* and *tilerMapping* transformations can be executed independently of these sub-chains, they can be executed before, after or inside the previously described sub-chains. As a consequence, up to 180 chains can be obtained from this product. Following the sub-chains computed by our derivation tool, a valid transformation chain could be:

$$\begin{aligned} &explicitAllocatlion \rightarrow tiler2task \rightarrow tilerMapping \rightarrow \cdots\\ &\qquad\qquad \cdots poly\_loop \rightarrow shape2loop \rightarrow memMapping \rightarrow openMP \end{aligned}$$

Without any lead, the *end user* has only one constraint: the model to text transformation must be the last of the chain. From the product p and its associated

[1] We used a set of logical predicates implemented using he Prolog language to implement the *Derivation Tool.*

set of transformations T_p, it means that the *end user* has the choice to organise 6 transformations. Thus, she has $P(6, 6) = 720$ choices to organise the model to model transformations. Among the 720 chaining possibilities, many are not valid because the require relationships are not considered. So, without any indication, the *end user* has to choose from 720, potentially non valid, chains, whereas with our methodology, the choice is reduced to 180 valid chains only.

To sum up, our methodology and the associated tool have allowed the *end user* (without any knowledge about transformations) to easily build chains. She has selected transformations based on embedded system features *i.e.* using terms she is familiar with. Finally, she has to choose among 180 valid chains whereas initially she was confronted to a huge number of possible chains that she has to build by scrutinizing the transformation code.

5 State of the Art

In order to enhance the reusability of transformations, several authors promote the decomposition of transformations into smaller ones. However transformations have then to be chained. Vanhooff *et al.* proposed an approach based on the explicit and manual identification of the required and provided concepts by the chain developer for example using a profile in order to later build the chain [21]. Our approach relies on the feature model to compose the transformations.

Several approaches have been proposed to build chains. Transformations are considered as functions to compose if their domains are compliant [13] or UML activity that can be chained using different operators: composition, conditional composition, parallel composition and loop [16]. However, in both cases, the transformation chain has to be manually specify by the designer, without any specific help. In the latter case, they are executed using the provided model transformation orchestration tool. Our approach could be used upstream to identify the useful transformations and to compose them.

Transformation chaining relies on constraints that can be automatically identified, *e.g.* using the distinction between concepts copied and those mutated [4]. This approach only deals with endogenous transformations (even if a possible extension to heterogeneous transformations is suggested). With the "require" constraints, we have extended this approach to heterogenous transformations.

Several approaches propose to deal with the complexity of large systems with a feature-based approach. For example, feature models were accurately used to model the intrinsic variability of the Linux Kernel [10], and support end-user during the kernel configuration task. The approach proposed in this paper follow the same idea, that is, the use of feature modelling to leverage a highly variable systems into an entity configurable by the end-user.

Being able to extract the features from the implementation is a challenge [2]. The most difficult part is the extraction of the feature hierarchy from the "flattened" implementation [19]. Inferring such a hierarchy relies on domain heuristics that rank the possible hierarchy, and the final assessment of these ranks by a domain expert. In this paper, we do not consider the automatic extraction of the features from the transformation set, and only rely on the business expert to properly model the feature diagram. Being able to support the business expert during this task is an interesting perspective of this work.

Feature models are also used to support the reverse engineering of large scale systems [1]. For example, the FraSCAti platform (an open source implementation of the SCA standard) was accurately reverse-engineered to support its assessment. Based on a dedicated tool that extracts the architecture from the implementation, the authors confront the automatically extracted feature model with the one defined by the *business expert*. This approach complements ours, as we also rely on a tool that automatically infers feature information from the actual implementation of the system (in our case requirements between features). But instead of assessing the model defined by the business expert, we focused on its enrichment, by merging the set of automatically identified information in this feature model. We were able to identify several situations where the actual system was not "as variable" as the business expert thought.

6 Conclusions and Perspectives

In order to be reusable and maintainable, model transformations are written according to a single intention and complex transformations are built as the chaining of smallest ones. In this paper, we proposed an approach based on FD to support the design of model transformation chains. Based on a classification of the transformations made by a *business expert*, This approach allows an *end user* to build such chains, without any prior knowledge of model transformation technologies. The implementation of the transformations is also automatically taken into account to ensure that the built chains are valid from a run-time point of view. From an implementation point of view, the approach is independent of any tools and can be easily coupled to existing approaches (*e.g.*, FeatureIDE, Familiar). The approach was validated on the Gaspard2 case study, and we are currently pursuing another validation study in the domain of website engineering.

The resulting chains are valid according to a type based approach [7]. However, two transformations that can be chained into both orders from a syntactic perspective are not obviously commutable from a business point of view: the execution of the two successive transformations on whatever models may not always lead to the same result. A perspective of this work is to enhance the expressiveness of the requirement detection mechanisms to address this issue. Another perspective concerns the FD refinement. Indeed, the FD being manually designed by the *business expert*, some constraints between features may have been omitted. The automatic requirement relationships extraction could be a first help to highlight a badly / incompletely designed FD. To help the *business expert* in the definition or the refinement of the FD, we plan to automatically extract features from the documentation written by the transformation developers.

References

1. Acher, M., Cleve, A., Collet, P., Merle, P., Duchien, L., Lahire, P.: Reverse Engineering Architectural Feature Models. In: Crnkovic, I., Gruhn, V., Book, M. (eds.) ECSA 2011. LNCS, vol. 6903, pp. 220–235. Springer, Heidelberg (2011)
2. Acher, M., Cleve, A., Perrouin, G., Heymans, P., Vanbeneden, C., Collet, P., Lahire, P.: On extracting feature models from product descriptions. In: Eisenecker, U.W., Apel, S., Gnesi, S. (eds.) VaMoS, pp. 45–54. ACM (2012)
3. Bézivin, J., Gerbé, O.: Towards a Precise Definition of the OMG/MDA Framework. In: 16th IEEE International Conference on Automated Software Engineering (ASE 2001), San Diego, CA, USA, pp. 273–280 (2001)
4. Chenouard, R., Jouault, F.: Automatically Discovering Hidden Transformation Chaining Constraints. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 92–106. Springer, Heidelberg (2009)
5. Cordy, J.: Eating our own Dog Food: DSLs for Generative and Transformational Engineering. In: GPCE (2009)
6. Etien, A., Aranega, V., Blanc, X., Paige, R.: Chaining Model Transformations. In: Submitted to ECMFA Conference (2012)
7. Etien, A., Muller, A., Legrand, T., Blanc, X.: Combining Independent Model Transformations. In: Proceedings of the ACM SAC, Software Engineering Track, pp. 2239–2345 (2010)
8. Gamatié, A., Le Beux, S., Piel, É., Ben Atitallah, R., Etien, A., Marquet, P., Dekeyser, J.-L.: A Model Driven Design Framework for Massively Parallel Embedded Systems. ACM Transactions on Embedded Computing Systems 10(4) (2011)
9. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) - Feasibility Study. Technical report, The Software Engineering Institute (1990)
10. Lotufo, R., She, S., Berger, T., Czarnecki, K., Wąsowski, A.: Evolution of the Linux Kernel Variability Model. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 136–150. Springer, Heidelberg (2010)
11. Mens, T., Taentzer, G., Runge, O.: Detecting Structural Refactoring Conflicts Using Critical Pair Analysis. Electronic Notes in Theoretical Computer Science 127(3), 113–128 (2005)
12. Miller, J., Mukerji, J.: MDA Guide Version 1.0.1. Technical report, Object Management Group, OMG (2003)
13. Oldevik, J.: Transformation Composition Modelling Framework. In: Kutvonen, L., Alonistioti, N. (eds.) DAIS 2005. LNCS, vol. 3543, pp. 108–114. Springer, Heidelberg (2005)
14. Ossher, H., Harrison, W., Tarr, P.: Software engineering tools and environments: a roadmap. In: ICSE 2000: Proceedings of the Conference on The Future of Software Engineering, pp. 261–277. ACM, New York (2000)
15. von Pilgrim, J., Vanhooff, B., Schulz-Gerlach, I., Berbers, Y.: Constructing and Visualizing Transformation Chains. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 17–32. Springer, Heidelberg (2008)
16. Rivera, J.E., Ruiz-Gonzalez, D., Lopez-Romero, F., Bautista, J., Vallecillo, A.: Orchestrating ATL Model Transformations. In: Proc. of MtATL 2009, Nantes, France, pp. 34–46 (July 2009)
17. Schmidt, D.C.: Guest Editor's Introduction: Model-Driven Engineering. IEEE Computer 39(2), 25–31 (2006)

18. Selic, B.: The Pragmatics of Model-Driven Development. IEEE Software 20(5), 19–25 (2003)
19. She, S., Lotufo, R., Berger, T., Wasowski, A., Czarnecki, K.: Reverse engineering feature models. In: Taylor, R.N., Gall, H., Medvidovic, N. (eds.) ICSE, pp. 461–470. ACM (2011)
20. Vanhooff, B., Ayed, D., Berbers, Y.: A Framework for Transformation Chain Development Processes. In: Proceedings of the ECMDA Composition of Model Transformations Workshop, pp. 3–8 (2006)
21. Vanhooff, B., Berbers, Y.: Breaking Up the Transformation Chain. In: Proceedings of the Best Practices for Model-Driven Software Development at OOPSLA 2005, San Diego, California, USA (2005)
22. Wagelaar, D., Van Der Straeten, R., Deridder, D.: Module Superimposition: a Composition Technique for Rule-based Model Transformation Languages. In: Software and Systems Modeling (2009) (online first)

A Generic Approach Simplifying Model-to-Model Transformation Chains

Gerd Kainz[1], Christian Buckl[1], and Alois Knoll[2]

[1] fortiss, Cyber-Physical Systems,
Guerickestr. 25, 80805 Munich, Germany
{kainz,buckl}@fortiss.org

[2] Faculty of Informatics, Technical University Munich,
Boltzmannstr. 3, 85748 Garching, Germany
knoll@in.tum.de

Abstract. The model-driven architecture proposes stepwise model refinement. The resulting model-to-model (M2M) transformation chains can consist of many steps. For realizing the transformations two approaches exist: Exogenous transformations, where input and output use different metamodels, and endogenous transformations, that use the same metamodel for input and output. Due to the particularities of embedded systems, using only endogenous transformations is not appropriate. For exogenous transformations, problems arise with respect to creation and maintenance of the subsequent metamodels. Another problem of these M2M transformation chains is that for one transformation step typically large parts of the model data remain unchanged. The resulting M2M transformation does often include many copy operations that distract the developers from the "real" transformations and increase implementation overhead. This paper introduces a generic approach that solves these issues by a (semi-) automatic metamodel construction and copy operation of unchanged model data between subsequent steps.

Keywords: Transformation Chain, Model-to-Model Transformation, Metamodel-to-Metamodel Transformation, Model-driven Software Development, Model-driven Architecture.

1 Introduction

The model-driven architecture (MDA) [1] has been successfully used to cope with large and complex systems. MDA suggests transforming platform independent models (PIMs) by a series of model-to-model (M2M) transformations into platform specific models (PSMs). Especially in the context of model-driven software development (MDSD) [2] of embedded systems, this stepwise refinement is very helpful. Embedded systems are characterized by the importance of extra-functional requirements, timing issues, and the heterogeneity of the involved components and platforms. Therefore, the transformations from PIM to PSM have to take into account several tasks. To enhance readability and maintainability, every M2M transformation step should ideally perform one task.

These tasks could be for example the mapping of software to hardware components or the calculation of an execution schedule.

In the domain of embedded systems the metamodels used for user input and for code generation very often differ significantly [3]. Due to this difference, a big metamodel, whose structure is suited both for user input and code generation, would be inadequate. The ideal transformation process consists of a series of refinements resulting in intermediate models based on different metamodels. In this approach of handling only one task within one M2M transformation step, the changes between steps at the metamodel and model level are rather small and only represent intermediate steps of the transformation towards the final metamodel and model.

One problem of M2M transformation chains is the creation and maintenance of the related metamodels. Successive metamodels typically have large parts in common. As there is currently no tool support available for constructing these metamodels, they are created manually, typically using Copy&Paste. An additional problem arises if later a metamodel in a M2M transformation chain is changed, e.g., by adding a new attribute. Usually the same adaptation has to be applied to subsequent metamodels as well. A manual execution of such changes is error-prone and tedious, hence should be avoided. Very often these problems are avoided by reducing the number of steps in a M2M transformation chain. This paper presents an approach to create and maintain the metamodels based on difference descriptions.

If transformations between models with different metamodels (exogenous transformation) [4] are implemented using an operational M2M transformation language, such as Xtend[1] or QVTOperational [5], the unchanged parts of the system have to be copied manually. As a result, the size of the code for the "real" M2M transformation is very often negligible compared to these manual copy operations. To avoid the additional overhead for implementing these copy operations, different refinement steps are very often combined or even only one large M2M transformation is used. Such M2M transformations contradict state-of-the-art in modern software engineering, which is based on modularity and demands to focus on one task at a time. Therefore, this paper presents a way to deal with the copy of data between successive models.

Section 2 clearly defines the problem statement and the focus of this paper. An overview of our solution is given in Section 3. The approach supports both the creation and maintenance of M2M transformation chains with respect to the two above mentioned problems. Section 4 presents an incremental definition of subsequent metamodels on the metamodel level. A semi-automatic conduction of data copy and type transformation operations for unchanged parts[2] between models based on different metamodels is described in Section 5. The "real" M2M transformation still needs to be specified manually. The implementation and the evaluation of the approach in the context of two MDSD tools for embedded

[1] Xtend/Xpand: `http://wiki.eclipse.org/Xpand`

[2] Underlying metamodel structure has not changed.

systems are contained in Section 6. The paper is concluded by a discussion of related work in Section 7 and a summary in Section 8.

2 Problem Statement

Following the MDSD methodology [2] user-defined models are stepwise combined and refined to a model adequate for code generation. This is especially useful when applying MDSD in the domain of embedded system. Due to the high heterogeneity of platforms and hence of implementations, PIMs abstract from the underlying implementations to simplify the modeling task for developers. In the process of stepwise refinement, a PSM should be calculated that is an optimal representation for code generation of a specific platform. One example is schedule specification. While it is simpler to model the execution through dependencies between tasks, the code generation is simplified if a concrete schedule with start times is calculated during M2M transformations. The same is true for the combination of models. To separate concerns and to reduce complexity, the description of embedded system is very often done using several, aligned models targeting different aspects of the system. Aligned models are models, which were created with respect to each other. They share information and can reference elements of each other without any problems. By working with different models, the developers can concentrate on selected system aspects and their associated data. Examples can be a model describing the used hardware and a model to describe the application. The code generation is simplified when these different "views" are merged.

Ideally the M2M transformation between the input model(s) and the final output model is split up in many small transformations. Each of these transformations then focuses on one task, e.g., schedule calculation or identifier assignment. Hence, a transformation changes only a small part of the model data. The transformations in the chain can be implemented using exogenous or endogenous transformations [4]. Exogenous transformations are transformations between models based on different metamodels. In contrast, endogenous transformations are transformations between models based on the same metamodel. It is possible to perform transformations with big structural changes through endogenous transformations. However, due to the big difference between PIMs and PSMs for embedded systems, the use of endogenous transformations alone is usually not advisable. Hence, this paper focuses on the **support of step-wise model refinement using exogenous transformations**[3].

One problem with exogenous transformations is the necessity to create and maintain further metamodels with large common parts. Very often the similarity between these metamodels leads to a construction using Copy&Paste. During maintenance, problems can arise when metamodels are extended and adapted to new needs. This usually requires applying the same changes in subsequent metamodels. Depending on the length of the M2M transformation chain this can be

[3] The problems with exogenous transformations discussed in the following, do not exist for endogenous transformations.

very time-consuming. Moreover, the refactoring is tedious and error-prone. Figure 1 shows the structure of a M2M transformation chain from PIM to PSM. Metamodel evolution [6] is very similar as it considers the migration of models, after the corresponding metamodel has been changed. The major difference between a M2M transformation chain and metamodel evolution is the life cycles of metamodels. In metamodel evolution, only the latest metamodel is of concern as this metamodel presents the latest version of the tool. The migration of models is only performed once. In transformation chains, all metamodels are required and the full chain of metamodels is processed every time the tooling is invoked for an application. This difference causes some practicability issues that are discussed in the following sections. M2M transformation chains and metamodel evolution are orthogonal to each other as shown in Figure 1. This paper proposes an approach for **creating and maintaining metamodels in exogenous M2M transformation chains based on difference specifications between metamodels**.

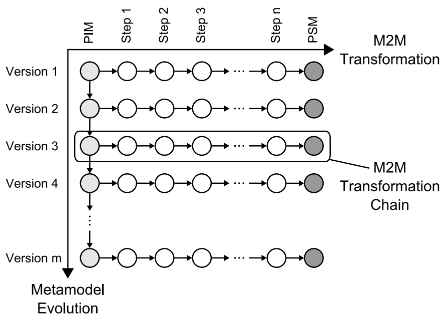


Fig. 1. Relation between Transformation Chains and Metamodel Evolution

Another disadvantage of exogenous transformations is the need to transform all the data of input model(s) into the output model. This transformation also includes copying data, which is not modified by the current M2M transformation, but needs to be transformed into the namespace of the new model. For operational (imperative) M2M transformation languages these copy operations have to be specified by the developers for all model elements. This is a very time-consuming and tedious job. Furthermore, the developers have to ensure that all data are copied from the input model(s) to the output model. The resulting M2M transformation code is very often a mixture of copy and "real" M2M transformation operations. As a consequence, these copy operations hinder the identification of the essential parts and ideas of the M2M transformation itself. To avoid the additional overhead, different M2M transformation steps are often combined or even only one large M2M transformation is used. This paper proposes an approach to **(semi-) automatically copy unchanged parts between models through the use of a function library**.

3 General Approach

The problem statement targets both the metamodel and model level of M2M transformation chains. To simplify discussion, we will deal with each of the problems in a separate section. As model transformations are based on the metamodels created by metamodel transformations, they can only be executed after the metamodel transformations. Hence, we start with the discussion of metamodel-to-metamodel (MM2MM) transformations. Typically, the transformations on models are executed more often than transformations on metamodels. The reason for this is that metamodel transformations belong to a change in the tool, whereas model transformations are part of the tool application to create new applications. Performance is therefore mainly an issue for M2M transformations and can be neglected to a certain extent for MM2MM transformations.

Figure 2 shows the proposed approach. The models / metamodels of the different steps are connected through transformations belonging to a M2M transformation chain. Numbers indicate the designated order of steps. The developers start with defining the input metamodels (1). Afterwards a difference model (2) [7,8,9] is used to create the metamodel of the next step (3). Based on this new metamodel the developer can define the model transformation containing function calls to copy unchanged model data and the "real" transformation (4). Since the "real" transformations represent the intelligence of the tool, they still have to be implemented by the developers without any further support. These steps can be repeated as often as needed (5–8). To create a new application the user needs to define the required models (9) and start the processing (10–12). In the approach only the differences between steps are specified manually. Similarities are handled automatically.

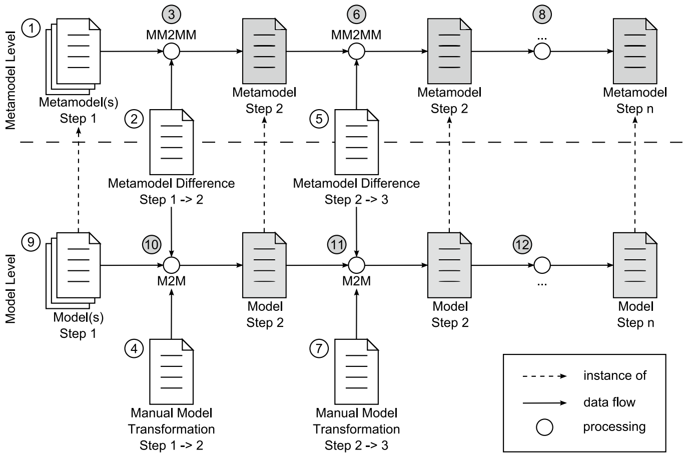


Fig. 2. Schematic Illustration of Model-to-Model Transformation Chain Approach. Gray elements indicate generated artifacts and automatic steps.

4 Metamodel-to-Metamodel Transformations

For supporting M2M transformation chains on the metamodel level, an easy way of creating the metamodels used in the different steps is needed. To keep the overhead for managing the metamodels as small as possible, we suggest specifying only the changes between metamodels (model deltas), e.g., adding, deleting, or modifying of packages, classes, attributes, references, or operations. This is closely related to metamodel evolution. The main difference is that metamodel evolution usually only focuses on calculating a difference model to (semi-) automate the M2M transformation (model migration). The difference model is either calculated by tracking changes of the developers when changing the metamodel or by comparing the old and new metamodel. In contrast, we use the difference model to calculate a successor metamodel out of the given ones. It is important to note that the predecessor metamodels might be affected by changes to the preceding metamodel chain. Hence, the tool must also support the developers by notifying if a difference model becomes partly invalid. In addition, our difference model must be able to specify combinations (used to merge different views) and adaptations of more than one metamodel, whereas metamodel evolution can only relate two metamodels with each other.

Example. Before the approach is presented in detail, a simple example is given. A system consisting of hard- and software components is modeled using separate models (views). These models shall be merged and then further modified. A suitable metamodel is needed to store the newly calculated data. Therefore, the metamodels have to be merged into one metamodel. The merge of the two metamodels raises a conflict as both contain a class *Component*. To resolve the conflict the *Component* classes are renamed into *HWComponent* respectively *SWComponent*. The commonalities of the classes are moved into a new abstract *Component* class. In addition, an *id* attribute is added to give all *Component*s unique identifiers. Figure 3 shows on the left side the preceding metamodels and on the right side the newly generated metamodel. The figure also depicts an intermediate step, which will be described in the discussion of the algorithm.

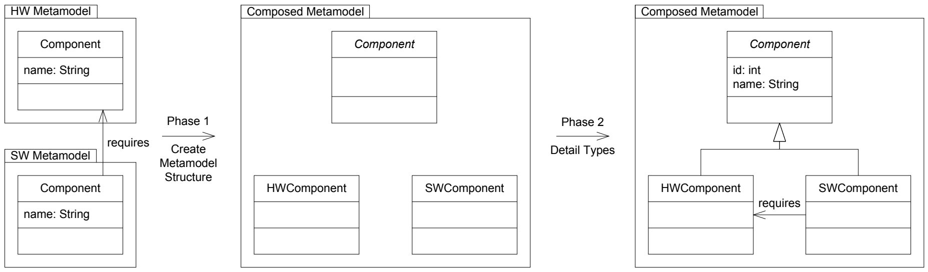


Fig. 3. Simple Example of a Metamodel-to-Metamodel Transformation

To create a subsequent metamodel the developers only need to specify the input metamodels and the changes, which shall be applied. A tool then creates the new metamodel. Figure 4 shows the difference model used to create the new metamodel. Any number of difference models can be specified to create an arbitrary number of subsequent metamodels, where one MM2MM transformation with its difference model builds upon the result of the previous one.

```
Transformation: composed (www.fortiss.org/tooling/m2m/composed)
  Metamodel: hardware (www.fortiss.org/tooling/m2m/hardware)
    Class: Component -> HWComponent [super class = Component] => Modify
      Attribute: name => Delete
  Metamodel: software (www.fortiss.org/tooling/m2m/software)
    Class: Component -> SWComponent [super class = Component] => Modify
      Attribute: name => Delete
  Class: Component [abstract] => Add
    Attribute: id [int] => Add
    Attribute: name [String] => Add
```

Fig. 4. Difference Model Used to Create the new Metamodel of the Simple Example

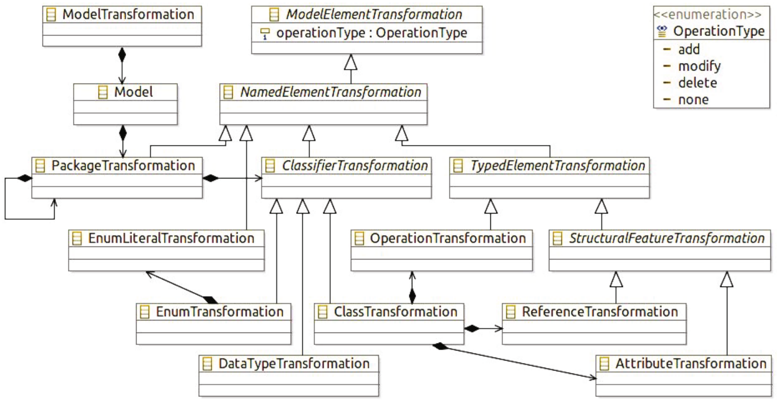


Fig. 5. Simplified Metamodel to Specify Metamodel Transformations. *ModelTransformation* constitute the root element. *OperationType* defines the kind of operation to perform, where *none* is used if only child elements are affected by transformations.

Supported Operations. To specify the adaptations of the metamodels in an unambiguous way, we provide a metamodel for specifying the difference model for MM2MM transformations. Figure 5 shows the basic structure of the metamodel used to specify MM2MM transformations. Based on this metamodel, it is easy to define all changes. The difference model allows to specify adding, deleting, and modifying of metamodel elements. The low level specification of changes gives high flexibility and allows a fine grained transformation of metamodels. Figure 6 shows a table with all supported operations. The column *Add To / Remove From* state where the elements can be added / deleted and *Modify* contains a list of changeable properties.

Element	Add To / Delete From	Modify
Package	package	name
Class	package	name, abstract class, super class
Attribute	class	name, type, multiplicity
Reference	class	name, type, multiplicity, containment
Operation	class	name, return type, parameters
Enumeration	package	name
Enumeration literal	enumeration	name, value
Data type	package	name, instance type name
Annotation	element	key, value

Fig. 6. Supported Metamodel Transformations

Transformation Algorithm. For the MM2MM transformation a difference model is taken as input, which references the metamodels of the previous step and includes a specification of all changes to apply. As result an adapted and potentially combined metamodel is returned. In addition, the specified changes are checked for consistency to cope with potential changes to the preceding metamodels. This prevents the generation of inconsistent metamodels, e.g., metamodel containing classes with same name or usage of not existing data types.

The actual transformation is carried out in two phases. In the first phase, all packages and types (classes, enumerations, and data types) are created. For each package and type the algorithm checks, whether it is not specified to be deleted, before creating them in the new metamodel. This is done for all packages and types of the referenced input metamodels. New packages and types are created in addition. The first phase takes care of creating types incorporating type renamings, without creating the internal structure of classes. By executing the transformation in this way, it can be ensured that all types already exist before they are used by other metamodel elements, e.g., data type of an attribute, super class. After creating all types in the new metamodel, a second phase takes care of the correct construction of the internal structure of classes. This includes the creation of attributes, references, and operations. Assignment of super classes is also part of this second phase.

Due to the fact that only changes between subsequent metamodels are specified through a difference model, changes to metamodels are automatically propagated to all subsequent metamodels along the M2M transformation chain. The implicit propagation of changes along the M2M transformation chain relieves the developers from applying the same adaptation many times and helps to focus on the differences between consecutive metamodels. Furthermore, careless mistakes are avoided by automating the metamodel adaptations.

5 Model-to-Model Transformations

The second aspect of M2M transformation chains targets M2M transformations themselves. The focus of this paper lies on removing the burden of writing data copy operations in operational transformation languages from developers. This is achieved by copying all unchanged (unaffected) model data from the input models to the output model. The developers can then fully concentrate on the "real" transformation. In comparison to metamodel evolution, our focus lies not on specifying the complete M2M transformation, so we split the M2M transformation in a generic part realizing the copy of unchanged data and a manual part. The developers still have to write the code for the "real" transformation. Furthermore, our approach supports the combination of more than one model describing a system from different viewpoints. In addition, we want to reuse the information from the MM2MM transformation to perform a more comprehensive M2M transformation considering exogenous transformations including renaming of metamodel elements.

Example. The approach is illustrated in Figure 7 based on the example of the previous section. The transformation consists of a (semi-) automated and a manual phase. First, generic copy operations based on the difference model are invoked to copy as much data as possible to the successor model. Afterwards, a manual transformation specified by the developers calculates the values for the new id attributes.

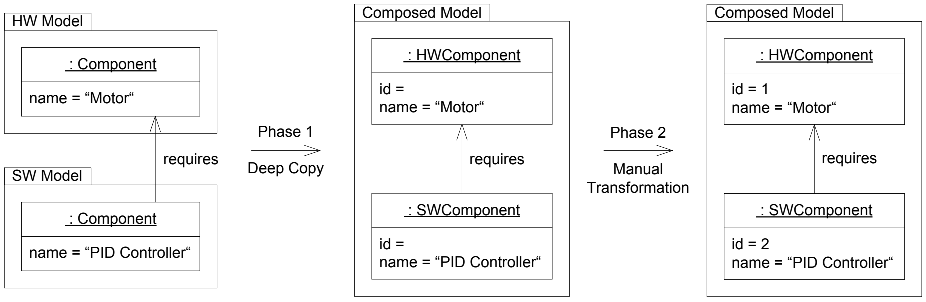


Fig. 7. Simple Example of a Model-to-Model Transformation

To reduce the effort for transformation encoding, all unchanged data between the steps are copied through calls of library functions. The library function *transformObject* takes care of converting the objects between the different namespaces – a deep copy is performed. Through the information contained in the difference model of the MM2MM transformation, the function is also capable of handling renamings of classes, attributes, or references, e.g., class *Component* $\rightarrow$ *HWComponent*. Thus, the developers can concentrate on the "real" transformation (assignment of unique identifiers to components). Figure 8 shows the encoding of the M2M transformation from a *HW* and *SW Model* to a *Composed Model* in the M2M transformation chain. The transformation is encoded using Xtend and contains 4 library functions calls. For usual examples the number of calls shall be lower than 10 and follow a similar structure.

```
//Manual transformation function
Void calculateAndAssignId(Component component):
  component.setId(component.eContainer.components.indexOf(component) + 1);

//Orchestration function of M2M transformation
create ComposedModel this (hw::SWModel hwmodel, sw::SWModel swmodel,
                           ModelTransformation transformation)
  initM2MTransformation(transformation) -> //Initialize M2M transformation

  //Call M2M transformation function copying unchanged data (deep copy)
  this.components.addAll(hwmodel.components.transformObject()) ->
  this.components.addAll(swmodel.components.transformObject()) ->

  //Manual M2M transformation
  this.components.calculateAndAsignId();

  finiM2MTransformation(this); //Finalize M2M transformation
```

Fig. 8. Manual Specification of a Model-to-Model Transformation. For simplification reasons each model contains a root element which stores all the other elements.

Transformation Algorithm. The transformation algorithm is started on an object of an input model. From there it traverses all reachable objects. Every time an object is reached, the algorithm tries to create an equivalent object in the output model and copies as much data as possible between those objects. This includes the transformation between data types of different namespaces. The class of the object, which has to be created in the output model, is determined by using the information contained in the difference model of the metamodels. If elements like classes or attributes do not exist in the next metamodel of the M2M transformation chain, e.g., they are deleted, their data is ignored. The same holds for newly created elements, for which no data exists. To simplify the algorithm all objects are created when they are reached for the first time regardless whether they are reached through a containment or normal association. For keeping track of already created objects a map is used, which relates input objects with their corresponding output objects. Later on, this map can be used during the manual transformation to navigate from input to output objects and vice versa and access their data as needed.

Since the algorithm starts at a specific object, it is possible that only a sub tree of the input model is traversed. The developers have to orchestrate the M2M transformation, so that all required parts are copied. This task is simplified by the fact, that the provided *transformObject* abstracts whether the object has already been transformed. The *finiM2MTransformation* in addition takes care that the resulting model contains no objects without corresponding container (storing) object.

The algorithm is provided as a JAVA library. This library contains functions for initializing a transformation (*initM2MTransformation*), transforming objects (*transformObject*), storing relations between input and output objects (*storeMapping*), getting related objects (*getDestinationObjects* and *getSourceObjects*), and finalizing a transformation (*finiM2MTransformation*). In the example the library has been used from Xtend, but it can be used with any other model transformation language. It is even possible to perform the generic transformation in JAVA and do the manual part with a model transformation language.

The library can also be used directly without specifying the metamodel differences. In this case the output metamodel needs to be specified. Unchanged parts are then copied to the output model by relying only on the information represented in the structure of the underlying metamodels. The type, attribute, and reference names are than used as matching criteria. This requires unique type names over all input metamodels, which are combined.

6 Implementation and Evaluation

In the following, the approach and its implementation are evaluated in the context of two MDSD tools of the embedded systems domain: FTOS and ϵSOA. The tools are built according to state-of-the-art for embedded systems development and rely on M2M transformations to calculate data needed for code generation. In both tools, the approach has been integrated to simplify their M2M transformations. Both tools are based on the Eclipse Modeling Framework (EMF)[4] and use the languages Xtend and Java for M2M transformations.

6.1 Implementation Details

The presented approach has been realized based on EMF, which can be considered as an implementation of the Essential Meta Object Facility (EMOF) [10]. The implementation consists of a metamodel used to specify MM2MM transformations, a script to perform MM2MM transformations based on difference models, and a library to support the (semi-) automatic copy of data in exogenous M2M transformations for operational (imperative) model transformation languages. Figure 9 shows the size of the implementation containing support for both M2M and MM2MM transformations, where the code for MM2MM transformations forms the majority. The implementation and an extended example are available at `http://tooling.fortiss.org/`.

Criteria	**JAVA Code**	**Xtend Code**
# Functions	76	14
# Statements	999	28

Fig. 9. Size of Implementation Supporting Model-to-Model Transformation Chains

6.2 Evaluation of FTOS

FTOS [3,11] targets fault-tolerant real-time systems. It generates an application specific run-time system including automated selection and configuration of appropriate fault-tolerance mechanisms. FTOS is based on four input models with their corresponding metamodels. In the hardware model, developers can describe the hardware topology (nodes and networks). A software model is used to specify the application components with a coarse schedule. The set of faults

[4] EMF: `http://www.eclipse.org/modeling/emf/`

that might occur in the system are defined in a fault model. The fault tolerance model is used to select fault detection tests and fault tolerance strategies.

During M2M transformations the four models are merged into one model, appropriate fault-detection mechanisms, and a refined schedule are calculated. A detailed discussion of the different calculations can be found in [3]. To avoid nasty copy operations, a huge and complex M2M transformation was used instead of applying a series of fine-grained M2M transformations. Another problem was the manual creation of the output metamodel, since the input metamodels are frequently extended to support further hardware components or other fault-tolerance mechanisms.

For FTOS, we applied the approach without relying on a difference model for MM2MM transformations. Hence, there is no support for metamodel changes in the M2M transformation chain and for the handling of renamed objects during M2M transformation. Figures 10 and 11 depict the results of the improvement. As can be seen in Figure 10, the code size reduction of the M2M transformation is significant. This results only from the elimination of copy statements. Even in this bad setup by using only one big transformation containing a lot of calculations, the ratio of simple copy instructions contained is high. The increase of the runtime for an equivalent transformation by using the generic library instead of a manual optimized transformation is instead negligible, as stated in Figure 11. Even without using MM2MM transformations the major benefits are the significant reduction of transformation functions and statements. This reduction can be explained by the fact that many elements and their properties are simply copied during the M2M transformation. In addition, the readability of the manual M2M transformation code has been improved, since the remaining code mainly contains code describing the "real" transformation.

Tool	**# Metamodel Elements**	**Criteria**	**JAVA Code**			**Xtend Code**		
			Old Vers.	**M2M Vers.**	**Improvement**	**Old Vers.**	**M2M Vers.**	**Improvement**
FTOS	101	# Functions	124	93	25.0 %	406	286	29.6 %
		# Statements	1285	1045	18.7 %	1881	1146	39.1 %
ϵSOA*	13 (+ 79)**	# Functions	8	1	87.5 %	22	6	72.7 %
		# Statements	59	20	66.1 %	72	22	69.4 %

Fig. 10. Evaluation Results without and with the Presented Approach (* Only code related to M2M transformation and handling of instances of manual created metamodel elements are considered. Other code is ignored, e.g., routing calculation or handling of instances of generated metamodel objects. ** Generated metamodel elements).

6.3 Evaluation of ϵSOA

ϵSOA [12] is used to develop sensor / actuator networks. During M2M transformations communication routes are calculated and the routing tables are prepared. ϵSOA is based on four input models with their corresponding metamodels.

Developers define and configure the nodes engaged in the system and their connection with each other in the nodes and network model. The available services are defined in the service model. In the application model developers can instantiate services on nodes and configure their communication relations.

During M2M transformations an appropriate network routing is calculated for the specified communication. Along with this, unique identifiers are assigned to instantiated services. Most of the calculations and storing of data are done in Java. This setting contradicts the main philosophy of MDSD as calculated data is stored outside of models. The major reason why this approach was selected, was to avoid the extension of the underlying metamodel, since the affected part of the metamodel is generated and quite often changed. Otherwise the integration of the manual changes had to be repeated after each regeneration of the corresponding metamodel part. For comfort reasons only one M2M transformation was implemented.

In the context of ϵSOA, both MM2MM and M2M transformations were applied. The advantages of the MM2MM transformation support are obvious, since the metamodels of the input models are extended frequently. By using MM2MM transformations, the changes between metamodels in the M2M transformation chain needed to be specified only once and can now be reapplied whenever an input metamodel changes. The MM2MM transformation consists of the merge of 6 metamodels. In addition, 1 new class is added and 2 are modified (not abstract anymore, renaming due to a name conflict), 1 enumeration is renamed due to a name conflict, 6 references are added and 1 is deleted, and 5 attributes are added and 1 is deleted. Figures 10 and 11 depict the results of the improvement. As can be seen in Figure 10, the code size of the M2M transformation could be dramatically reduced. Even by ignoring the improvements on major parts. The main reason for the huge reduction is that the M2M transformation is mainly a model combination. Since a combination of models consists predominantly of copy operations, it was easy to get rid of these operations by our approach. Figure 11 shows that even a decrease in the runtime for an equivalent transformation by using the generic library has been achieved. The difference between the run times observed in the context of ϵSOA and of FTOS can be motivated by the fact that the manual M2M transformations of FTOS require an additional traversing of the model whereas most of the transformations of ϵSOA are already realized by the library. The speed up is motivated by the execution of compiled Java code compared to interpreted Xtend code.

Tool	**# Objects in Application Model**	**Runtime**		
		Old Version	**M2M Version**	**Improvement**
FTOS	121	468.2 ms	478.8 ms	-2.3 %
ϵSOA	112	212.2 ms	199.4 ms	6.0 %

Fig. 11. Runtime without and with the Presented Approach (average over 5 runs)

7 Related Work

The M2M transformation part of our approach is highly related to model transformation languages like Xtend[5], QVTOperational [5], or the imperative part of ATL[6] [13] and constitutes an extension of such languages through a library. This extension is used to relieve the developers from specifying copy operations for unchanged model data by providing support for deep copy. As demonstrated in this paper, the amount of copy operations in a M2M transformation can be rather high. For similar reasons ATL offers a refine mode, which can be used to copy model elements, but works only for endogenous transformations.

The specification of differences between metamodels is closely related to the representation of model differences [7] and delta models used in software product lines (SPLs) [8,9]. Differences in SPLs are called features [14] and are used to integrate functionality into a base configuration. Features are ordered relatively to each other to ensure a consistent integration. When creating a new product, all the required features are selected. The features are ordered and their applications lead to the configured product. In our approach, we support the construction of M2M transformation chains defining a fixed order of the transformations. We do not only take care of transforming the model (product), but also consider the creation of the metamodels required by M2M transformation chains.

Glue Generator Tool (GGT) [15,16] is a framework dedicated to the reuse of PIMs and PSMs of existing applications. Composition rules are specified using GGTs own metamodel. Correspondence rules are used to relate model elements. For composition merge rules are used. Modifications are handled by override rules. In contrast to GGT, our approach is more concerned with the various calculations done in M2M transformation chains and their optimal support. Therefore, only aligned models are considered.

Epsilon Merging Language (EML) [17] is a metamodel based language for expressing model merges. It contains a model comparison and transformation language. Like GGT it is rule based. Match rules specify matching elements, which are then merged through according rules. Not matched elements are handled by transformation rules. EML is concerned with the merge of models based on a specification including copy operations. Our solution is focused on the automation of those copy operations based on type equality. Along with this, our approach offers a way to describe the adaptation of metamodels.

Epsilon Flock [18] is a model migration tool build on top of EML. It contains a rule based transformation language used to define adaptations for metamodel evolutions. This language includes a conservative copy algorithm, which is used to copy unchanged model elements to the new model version. As Epsilon Flock is used to adapt models after a metamodel evolution happened, it does not consider changing metamodels by itself. But as has been shown in this paper, the support of metamodel changes is important for M2M transformation chains. The same holds for other metamodel evolution tools, e.g., COPE [19] or Ecore2Ecore [20].

[5] Xtend/Xpand: `http://wiki.eclipse.org/Xpand`

[6] ATL: `http://www.eclipse.org/atl/`

Atlas Model Weaver (AMW)[7] is a model composition framework that uses a specification for model transformations called weaving model to produce an executable model transformation. The weaving model contains composition operators specifying the relation between the various input models. This weaving model is used by AMW to compose various models. In this sense, it is more a model transformation language. Copy operations can be automated based on the various relations stored in the weaving model. However, the construction of the output metamodel is not in the focus of AMW and not further supported through weaving models.

8 Conclusions

MDA proposes a model refinement in several steps from PIMs to PSMs. However, this requires the management of many similar metamodels and the copy of data between the corresponding models. If large parts of the model remain unchanged, the developers have to specify many copy operations. To avoid this problem, the developers typically use only few steps between PIMs and PSMs.

In this paper an approach was presented that supports on the one hand the (semi-) automatic metamodel construction to specify metamodel chains and to cope with later changes. On the other hand the (semi-) automatic copy of unchanged model data during M2M transformations is supported. The MM2MM transformation support has been applied to one MDSD tool, clearly showing its benefits there. The M2M transformation was applied to two MDSD tools. Both tools show a significant reduction of the code for M2M transformations (up to 70 %). This reduction is only related to avoiding simple copy operations. However, besides lower effort for specifying M2M transformations, the readability is improved drastically.

References

1. Miller, J., Mukerji, J.: MDA Guide Version 1.0.1 (June 2003)
2. Stahl, T., Völter, M.: Model-Driven Software Development: Technology, Engineering, Management. Wiley (2006)
3. Buckl, C.: Model-Based Development of Fault-Tolerant Real-Time Systems. Dissertation, Technische Universität München, München, Germany (2008)
4. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science 152, 125–142 (2006)
5. Object Management Group (OMG): Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification Version 1.1 (January 2011)
6. Wachsmuth, G.: Metamodel Adaptation and Model Co-adaptation. In: Bateni, M. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 600–624. Springer, Heidelberg (2007)
7. Cicchetti, A., Ruscio, D.D., Pierantonio, A.: A metamodel independent approach to difference representation. Journal of Object Technology 6(9), 165–185 (2007)

[7] AMW: `http://www.eclipse.org/gmt/amw/`

8. Schaefer, I., Bettini, L., Damiani, F., Tanzarella, N.: Delta-Oriented Programming of Software Product Lines. In: Bosch, J., Lee, J. (eds.) SPLC 2010. LNCS, vol. 6287, pp. 77–91. Springer, Heidelberg (2010)
9. Clarke, D., Helvensteijn, M., Schaefer, I.: Abstract delta modeling. In: Proceedings of the Ninth International Conference on Generative Programming and Component Engineering (GPCE 2010), pp. 13–22 (2010)
10. Object Management Group (OMG): Meta Object Facility (MOF) Core Specification Version 2.0 (January 2006)
11. Buckl, C., Sojer, D., Knoll, A.: FTOS: Model-driven development of fault-tolerant automation systems. In: 15th International Conference on Emerging Technologies and Factory Automation (ETFA 2010), Bilbao, Spain, pp. 1–8 (2010)
12. Buckl, C., Sommer, S., Scholz, A., Knoll, A., Kemper, A., Heuer, J., Schmitt, A.: Services to the field: An approach for resource constrained sensor/actor networks. In: International Conference on Advanced Information Networking and Applications Workshops (WAINA 2009), Bradford, UK, pp. 476–481 (2009)
13. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
14. Batory, D., Azanza, M., Saraiva, J.A.: The Objects and Arrows of Computational Design. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 1–20. Springer, Heidelberg (2008)
15. Bouzitouna, S., Gervais, M.P.: Composition rules for PIM reuse. In: 2nd European Workshop on Model Driven Architecture with Emphasis on Methodologies and Transformations (EDWMDA 2004), pp. 36–43 (2004)
16. Bouzitouna, S., Gervais, M.P., Blanc, X.: Model reuse in MDA. In: International Conference on Software Engineering Research and Practice (SERP 2005), Las Vegas, USA, pp. 354–360 (2005)
17. Kolovos, D., Rose, L., Paige, R.: The Epsilon Book (2010)
18. Rose, L.M., Kolovos, D.S., Paige, R.F., Polack, F.A.C.: Model Migration with Epsilon Flock. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 184–198. Springer, Heidelberg (2010)
19. Herrmannsdoerfer, M., Benz, S., Juergens, E.: COPE - Automating Coupled Evolution of Metamodels and Models. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 52–76. Springer, Heidelberg (2009)
20. Paternostro, M., Hussey, K.: Advanced features of the eclipse modeling framework. In: EclipseCON (March 2006)

An Approach for Synchronizing UML Models and Narrative Text in Literate Modeling

Gunnar Schulze[1], Joanna Chimiak-Opoka[1], and Jim Arlow[2]

[1] Institute of Computer Science, University of Innsbruck, Austria
gunnar.schulze@gmail.com, joanna.opoka@uibk.ac.at

[2] Clear View Training Limited, London, United Kingdom
jim@clearviewtraining.com

Abstract. A major challenge in adopting UML in industrial environments is the lack of accessibility and comprehensibility of some diagram types by non-technical stakeholders. Literate Modeling improves comprehension of these diagrams by adding narrative text, but lacks good tool support for synchronizing model and text. This paper presents an approach for keeping model and text synchronized by effectively combining state-of-the-art natural language processing technology with OCL model querying. Thereby, consistency of element names in the UML model with their counterparts in the text is achieved by using text annotations to provide the semantic link. At a structural level, we propose an algorithm that checks element relationships in the UML model using a set of validation constraints when particular sentence characteristics are detected. An analysis of the runtime complexity shows the feasibility of including the proposed solution in one of today's CASE tools.

1 Introduction

The Unified Modeling Language (UML) has become the de-facto standard modeling language. According to [20] 80 %– 90 % of questionnaire respondents use UML for Model Driven Software Engineering. Empirical studies have shown that even usage of UML models as software documentation is beneficial. A systematic literature review [11] based on 23 empirical papers concludes that the benefits of UML for developers outweigh the costs and risks. [12] finds that using UML documentation can increase by 54 % the functional correctness of changes in comparison to the control group.

Despite these benefits, our experience in working with several large companies throughout Europe is that UML modeling continues to be underused. We have seen time and again, that this is largely due to the lack of accessibility and comprehensibility of UML models by business users, domain experts, managers, some programmers, and even some analysts. In [1,2] we described our observations on the lack of accessibility of UML models due to the need for complex and expensive modeling tools, and the lack of comprehensibility due to the complex visual syntax and semantics of UML. In that paper we proposed a simple and effective solution to some of these problems—Literate Modeling.

In Literate Modeling, the UML model is embedded in an explanatory narrative to create a document called a Business Context Document. This is accessible because it is in the form of a text document that anyone can read. It is made comprehensible by the convention that the narrative part of the document must stand alone, and be completely comprehensible without any knowledge of UML. Of course, the introduction of an explanatory narrative creates the problem of keeping the narrative and the model in sync. How do you know that what the model says is accurately reflected in the narrative and vice versa? In [1,2] this problem was addressed by the use of a controlled vocabulary whereby model elements we referred to directly by name in the narrative. One could then read the narrative and, by inspection, see that it accorded with the model, or vice-versa. The Business Context Document closes the comprehensibility gap between the modeler and their non-technical audience and allows for effective review and feedback on the models by non-technical stakeholders.

Beginning with the work in [1] and continuing to apply Literate Modeling through domains as diverse as travel, defence, insurance and investment banking, we have consistently found that Literate Models are more accessible, comprehensible and therefore of higher value to our customers than plain UML models. However, there are two key issues that we have discovered in Literate Modeling: Firstly, successful Business Context Documents need good writing skills. We address this issue to some degree with a proposed set of Literate Modeling writing standards to be published in [3]. Secondly, Business Context Documents are difficult to write because of the need to reference model elements by name and then use inspection to check the consistency of the narrative and the model.

In the ideal case, the Business Context Document should be generated directly from the UML model. But as was pointed out in [1] this isn't really practicable because the UML model doesn't contain enough information to create a compelling narrative. In particular, all business context around the model, why it is important, what parts need to be emphasized, how it fits into the rest of the business, who it is important to, what "story" the model tells about the business, is typically absent from UML models. A more pragmatic approach is to create an environment that helps the writer to write Business Context Documents by automating (to some degree) the referencing and inspection requirements. Such an environment could be integrated into existing CASE tools, allowing the author to develop model and text in a synchronized manner.

In this paper, we present an approach that uses existing natural language processing tools in conjunction with OCL to address the referencing and inspection issues that must be handled in such an environment. The remainder of this paper is structured as follows: Section 2 establishes the background of our research. In Section 3, we present our synchronization approach, followed by a description of the complete synchronization algorithm. Section 5 analyzes the runtime complexity of the proposed solution. Section 6 briefly describes our prototype editor for Literate Modeling, followed by a discussion of related work in Section 7. The paper concludes in Section 8 with a summary of its contributions and possible future work.

2 Background

During individual activities in software development, the conceptual information of a software system is captured using different representations, which exhibit different degrees of formality. Initially, the requirements of a software system are expressed in natural language, as they are usually developed in discussion with the customer. Typically, these descriptions are then formalized using a modeling language such as UML. The resulting model serves as the basis for the actual implementation.

Moving between these representations involves some kind of transformation, which is shown in Fig. 1. Typically, descriptions in natural language are informal and may contain additional information that is irrelevant for the actual development of the software. UML can be seen as a semi-formal language, as it contains both formal and informal elements, whereas code is a completely formal representation of software.

The transitions between these representations cannot always be carried out fully automatically. The process of moving from natural language to UML models mainly suffers from the ambiguity of natural language, which makes it infeasible to automatically construct an accurate model from the text. The transformation of models into code is limited by missing detail in the model. Analogously, constructing models from source code may not be satisfactory as the resulting artifacts typically contain too much detail, contrasting with the intention of every model to provide a meaningful abstraction of a system. Similarly, descriptions in natural language remain on the same level of abstraction as the model, which does not contain any additional information that may be relevant from a business perspective.

Most modern CASE tools offer the possibility to generate source code from UML models, which can serve as a skeleton for the implementation. Some of them have reverse engineering capabilities as well, that allow construction of UML models from the code. What is much more challenging is closing the gap between natural language and models. For the direction from model to text, [15] describes a tool that generates customizable descriptions of object oriented

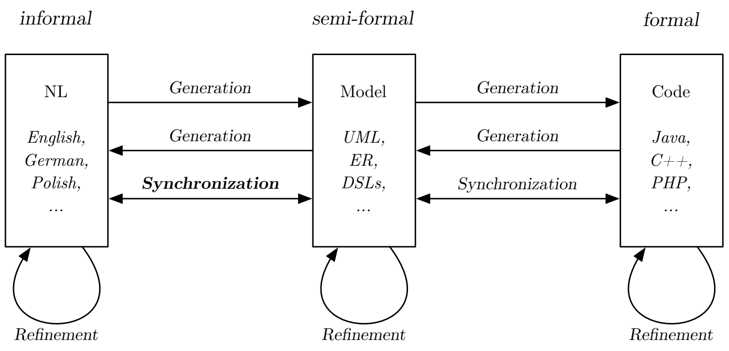


Fig. 1. Transformation between different representations of software

models. [16] presents an approach for generating natural language specifications from UML class diagrams. For the other direction, [13] describes a natural language based CASE tool that generates object-oriented models from requirements in natural language with assistance from the user or fully automatically. [8] proposes a tool for generating static UML models from natural language requirements. An approach that transforms UML Use Case models into analysis models and additionally establishes traceability links between these representations has been proposed in [22].

In practice, changes may occur in all three representations during the software development process, leading to incorrect and inconsistent information. In this regard, generative methods clearly have their limitations. A more flexible approach to address these consistency issues is synchronization. Between model and code, this synchronization is already provided by several CASE tools[1], known as Round-trip Engineering (RTE). However, to the best of our knowledge, there does not exist a solution for synchronizing models and natural language.

3 Synchronization Approach

The synchronization approach presented in this paper is based on two considerations. First, names of model elements that are referenced from the text need to be synchronized with their counterparts in the UML model. Second, sentences that contain references to model elements need to be checked whether they are in sync with the structural features of the model they describe.

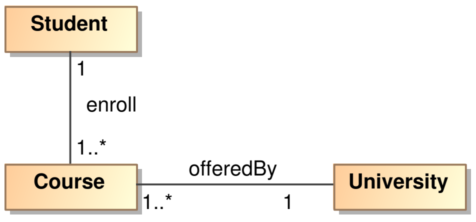
3.1 Running Example

To facilitate the explanation of our approach, this subsection provides a running example. In this paper, we focus on class diagrams as they are among the most important UML diagram types. However, the synchronization approach is not merely limited to the UML elements used in class diagrams, but provides a general mechanism to decrease the gap between natural language and UML models.

The UML class diagram in Fig. 2 shows a fragment of a university model explained using Literate Modeling. Model elements are referred to by their name, and typeset using a special font. In addition, some names for relationships like **offeredBy** are composed of two words, a verb (**offered**) and a preposition (**by**).

When changing the model or the text, inconsistencies may arise, as shown in Fig. 3. Here, the multiplicity of the association end attached to **Course** of the association **enrolls** has been changed from *one or more* to *zero or more*. Moreover, the association name **offeredBy** has been changed to **providedBy**. For the second inconsistency, only a single element needs to be considered, whereas for the first inconsistency, the complete sentence needs to be analyzed. Moreover, changing multiplicities may cause the grammatical number of nouns and verbs to

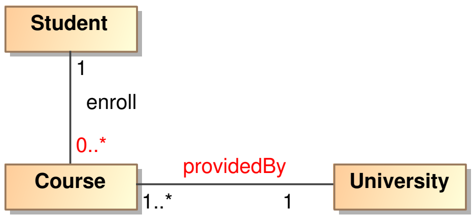
[1] Examples include MagicDraw UML (`http://www.magicdraw.com`) or Visual Paradigm (`http://www.visual-paradigm.com`)



(1) A **Student enrolls** for one or more **Courses**.

(2) The **Courses** are **offeredBy** the **University**.

Fig. 2. Class diagram fragment explained using Literate Modeling



(1) A **Student enrolls** for one or more **Courses**.

(2) The **Courses** are **offeredBy** the **University**.

Fig. 3. Inconsistencies between model and text

change, which must be considered as well. Using a conventional text processor, these changes may remain undetected, resulting in incorrect and inconsistent documentation.

3.2 Synchronization of Individual Elements

For synchronization of individual elements, we use text annotations that provide the semantic link between a particular word of a sentence and the corresponding element of the UML model, as shown in Fig. 4. In this way, model elements can be clearly identified as such and are easily distinguishable from words that do not describe some part of the model, even if they are lexically identical. Moreover, changes of model elements can be handled quickly since their occurrences within the text are known. Each annotation contains the unique resource identifier (URI) of the model element, as well as the fully qualified name of the element within the model, based on the idea of [2, p. 110]. By having both the unique identifier and the fully qualified name, changes can be detected even if the model has been edited separately from the text. For sentence (2) of Fig. 3, the system

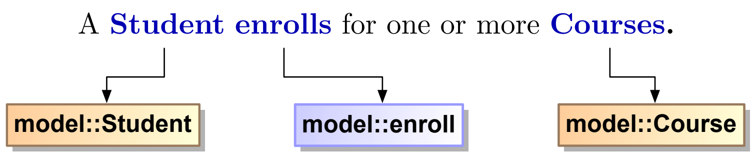


Fig. 4. Semantic annotation of text

would detect the difference between the strings **providedBy** and **offeredBy**, and change the name in the text accordingly.

3.3 Validation of Sentences

To take account of the model's structural features, sentences that contain references to model elements need to be validated. This is achieved by checking each sentence using a set of validation constraints. A validation constraint has the form of an implication

$$(C_S \Rightarrow C_M) \tag{1}$$

where C_S denotes a constraint that checks certain properties on the structure of a sentence, and C_M represents a meta-level constraint on objects of the UML model. Whenever the constraint C_S holds for a sentence, by implication C_M has to hold for the model. Note that the opposite direction does not hold in the general case. Particular properties of the model do not enforce a specific grammatical structure on the observed sentence, as those properties can be described using different sentences.

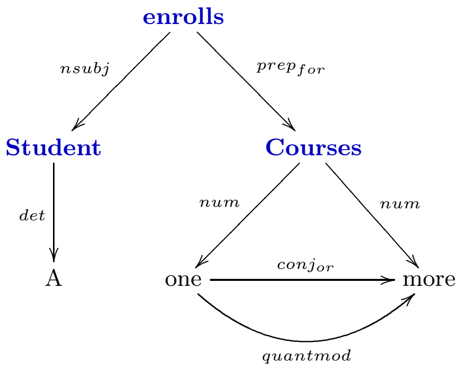
Sentence Constraints. For analyzing the structure of each sentence, we have used the Stanford PCFG Parser [14]. To correctly handle names of elements in a sentence, we modified the parser to account for non-meaningful names as well. This process is described in detail in [19].

In addition to traditional phrase structure trees, the Stanford parser provides a more meaningful representation in terms of grammatical relations, the Stanford Typed Dependencies (SD) representation [7]. Hereby, a sentence is represented as a binary relation between two sentence words, consisting of a *governor* and a *dependent*. Fig. 5 shows the typed dependencies for sentence (1) in both graphical and textual representation. In the graphical representation, each relation is represented by an arrow from the governor to its dependent, labeled with the name of the relation.

Based on this representation, we have developed a language that allows us to search for specific grammatical relationships between individual constituents of a sentence, as well as specify the type of the UML model elements particular words have to be associated with. A sentence constraint expressed in this language consists of three parts, a *context declaration* (keyword: **context**), one or more *type declarations* (keyword: **let**) and a *constraint expression* (keyword: **where**).

– The *context declaration* is used to specify the name of the variable that should be used as context for the corresponding model constraint.
– *Type declarations* are used to restrict particular variables to be of a specific type, which can be either a UML type, or one of the primitive types *Integer* and *String*.
– Finally, the *constraint expression* allows us to search for specific grammatical relationships between individual constituents of the sentence.

An example sentence constraint that matches sentence (1) is shown in listing 1.1.



(a)

det(**Student**, A)

$nsubj$(**enrolls**, **Student**)

$prep_{for}$(**enrolls**, **Courses**)

num(**Courses**, one)

$conj$(or, one, more)

$quantmod$(one, more)

num(**Courses**, more)

(b)

Fig. 5. Typed dependencies of sentence (1) shown in graphical (a) and textual form (b)

Listing 1.1. Example sentence constraint C_S

```
context verb

let(subj: uml('uml::Class'), verb: uml('uml::Association'),
    obj: uml('uml::Class'), lb: Integer)
where(nsubj(verb, subj) and prep(verb, obj)
      and quantmod(lb, ub) and num(ub, obj) and ub = 'more')
```

The constraint checks whether the observed sentence has a subject that is connected to an object using a verb and a preposition, with subject and object each being linked to an object of type `uml::Class` and the verb being linked to an object of type `uml::Association`. In addition, the existence of a quantification modifier is checked, with its first parameter being of type `Integer` and the second one equaling to the string "more".

Internally, the sentence constraints are translated into an equivalent Prolog query and evaluated using a light-weight Prolog system for Java [10] that allows the contribution of additional Prolog predicates written in the host language. Our evaluation engine automatically determines whether a particular word can be converted to the specified type, which works for numbers as well as numerals.

Model Constraints. For expressing the second component of the validation constraint, we use the Object Constraint Language (OCL) [17]. Validation constraints are required to work with all possible user models, hence we use OCL on the UML metamodel layer (M2) to constrain elements of the user model (M1). More precisely, the invariant mechanism of OCL is used to check whether a set of classes conforms to certain structural requirements.

To make this more clear, we take the sentence constraint of listing 1.1 and write a corresponding model constraint C_M, which is shown in listing 1.2. Note the missing context declaration in the OCL constraint. Here, a separate declaration is not needed as the actual context element and the values for variables are obtained from the successful evaluation of the sentence constraint.

Listing 1.2. Example model constraint C_M.

```
self.relatedElement->asSet() = Set{subj, obj}
and self.memberEnd->any(e: Property | e.type = subj).lower = 1
and self.memberEnd->any(e: Property | e.type = subj).upper = 1
and self.memberEnd->any(e: Property | e.type = obj).lower = lb
and self.memberEnd->any(e: Property | e.type = obj).upper = *
```

The constraint first checks whether the association relates the two elements at all. If this is true, the multiplicities of the association ends are investigated, i.e. whether they match the values obtained from the sentence constraint. Thereby, the string `'more'` in the sentence constraint has to be considered explicitly and is reflected by the multiplicity $*$ in the model constraint.

For sentence (1), the evaluation of the sentence constraint assigns the value 1 to the variable lb. The corresponding model constraint evaluates to false for the UML model of figure 3, as the lower bound of the **Course** end of the association **enroll** is 0, and not 1. In this case, a meaningful error message should be raised, e.g.

"There does not exist an association **enroll** between the classes **Student** and **Course** that has multiplicity 1..*".

Above, a configurable and extensible mechanism for mapping the semantics of sentences in natural language to UML was presented. By writing a sufficiently large set of constraint pairs ($C_S \Rightarrow C_M$), a variety of different writing styles can be covered.

In the next section, we present an algorithm that uses this approach to indicate inconsistencies between a UML model and accompanying narrative text.

4 Algorithm

The synchronization algorithm presented in this paper takes a UML model and an annotated piece of text as input and returns a set of problem markers, which each contains information about a particular inconsistency, i.e. the location within the text, the particular error message as well as any additional information on how to resolve the error.

Fig. 6 shows an activity diagram of the synchronization process, which is carried out in three stages. The *Pre-Processing* stage is responsible for synchronization of individual elements, as well as splitting the text into sentences that serve as input for the subsequent stages. The *Validation* stage is responsible for validating the sentences obtained from the previous stage using a set of validation constraints. Finally, the *Post-Processing* stage attempts to check whether inflection of nouns and verbs is correct with respect to the multiplicities of the involved model elements.

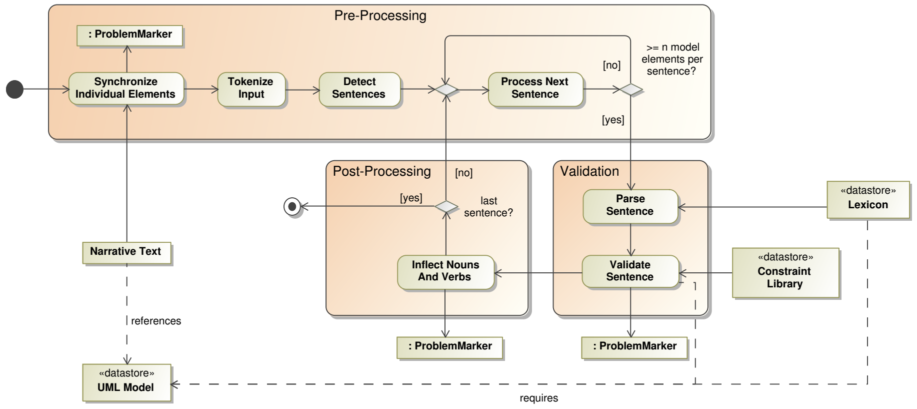


Fig. 6. Activity diagram of the synchronization process

4.1 Pre-Processing

The *Pre-Processing* stage synchronizes annotated words in the input text with their counterparts in the UML model. When the name of a model element has changed, the inflection operation that was applied on the previous name is figured out and repeated for the new name. For this task, we use an algorithmic approach for pluralization that is presented in [5]. If the choice of inflection is ambiguous, a problem marker is added. After all elements have been synchronized, the input text is tokenized and split into individual sentences. Thereby, a sentence is a candidate for validation if it contains at least a predefined number of semantic annotations, corresponding to the minimum number of model elements that are involved in a relationship. In that case, the validation stage is entered, otherwise, the system continues with the next sentence.

4.2 Validation

The second stage is concerned with the validation of each sentence using a predefined set of the validation constraints presented in subsection 3.3, which may be extended through additional constraints defined by the end user. Each constraint pair is evaluated on the current sentence. If C_S is valid, the corresponding model constraint C_M is evaluated on the involved elements of the UML model. If C_M holds, the algorithm continues with the next constraint pair, otherwise, a problem marker is added on the current sentence and the next sentence is processed. If a sentence passes all validation constraint checks, it may be considered as correct with respect to the constraints defined in the library.

4.3 Post-Processing

The last stage of the algorithm deals with the grammatical intricacies related to inflection, i.e. determining whether nouns and verbs have the correct grammatical number with respect to the multiplicities imposed by the model constraint. Inconsistencies at this stage can happen if the user changes multiplicities of relationships or attributes in the model. The inflection process consists of the two steps described below.

1. *Model Element to Noun Agreement:* In the first step, multiplicities for model elements are extracted from the OCL constraint if possible. Model elements that have a noun role within the sentence are validated with respect to correct inflection according to the multiplicities extracted from the OCL constraint. We argue that usually only one attribute or relationship involving multiplicities is expressed in a constraint pair, which makes the multiplicity-extraction process non-ambiguous.
2. *Noun to Verb Agreement:* In the second step, verbs that are related to a model element having a subject role are validated with respect to correct inflection. This is done by examining the corresponding relations of the typed dependencies for active and passive voice.

5 Complexity

In this section, we want to estimate the runtime complexity of our approach. The time required for synchronizing one sentence containing model elements constitutes of the time required for synchronizing the individual model elements (T_E) in the sentence and the time required for performing structural synchronization (T_S):

$$T_{sync} = T_E + T_S$$

T_E is linear with respect to the number of model elements in the sentence. This is usually a small constant less than or equal to 3, which corresponds to the typical *subject–verb–object* structure found in declarative sentences. Therefore, the runtime complexity is constant for a single sentence:

$$T_E \in O(1)$$

The time T_S for performing structural synchronization constitutes of the time required for performing natural language processing (T_{NLP}) and the time for evaluating all constraints (T_C):

$$T_S = T_{NLP} + T_C$$

The complexity for natural language processing using the Stanford PCFG parser is cubic with respect to the number of words n in a sentence [14]:

$$T_{NLP}(n) \in O(n^3)$$

Although the computational complexity does not look encouraging in the first place, it is acceptable considering the average sentence length of an English sentence of 15–20 words.

The time for evaluating all constraints constitutes of the time required to evaluate all sentence constraints (T_{C_S}) and the corresponding model constraints (T_{C_M}), respectively:

$$T_C = T_{C_S} + T_{C_M}$$

Each sentence constraint is precompiled into an equivalent Prolog expression, which is evaluated against a set of facts corresponding to the typed dependencies of each sentence

For our prototype, we used a Prolog interpreter implemented as an object-oriented SLD-solver [9]. Each sentence constraint is translated into a Prolog query that does not contain any nested terms, the typed dependencies of each sentence are translated into a set of ground terms. This reduces the worst-case complexity of unifying one query term from exponential to linear time with respect to the number of terms in the fact base. The complexity introduced by nesting of boolean operators in the query is negligible, as it was never necessary to write complicated sentence constraints in our practical tests. Under these assumptions, The resulting query is a set of Horn clauses that can be solved in linear time [6]. The number of typed dependencies in a sentence grows in linear fashion with the number of sentence words n, and the number of typed dependencies in the constraint approximately corresponds to the constant number of model elements. With $\#C$ being the number of constraints in the library, the time complexity of evaluating all sentence constraints is

$$T_{C_S} \in O(\#C \cdot n)$$

Model constraints are expressed as OCL invariants, whose evaluation can take exponential time in the worst case [4]. The actual complexity however depends on the constraint entered by the user. Moreover, model constraints are only evaluated if the associated sentence constraint evaluates to true, which happens usually once or twice per sentence. As a result,

$$T_{C_M} \text{is negligible}$$

Based on this observation, one can conclude that the bottleneck of the approach is the NLP part. This is also supported by the results of the experiments we have conducted [19]. Based on three chapters of [2], we measured the total time required to synchronize each document, as well as the execution time for individual steps of the synchronization. With a constraint library consisting of 20 constraints, we observed that 98 % of the synchronization time is consumed by NLP and only 2 % by sentence constraint evaluation. The time for element synchronization and model constraint evaluation was negligible. On our test machine, a MacBook Pro with a 2.53 GHz Intel Core 2 Duo processor and 4 GB of RAM, 151 ms were required on average to synchronize one sentence. For instance, for the Party Archetype Pattern chapter in [2] having 447 sentences, our prototype implementation needed 41.02 s for synchronization.

6 Tool Support

The presented approach has been implemented in a prototypic editor for Literate Modeling, called LiMonE[2], which was used to conduct the experiments discussed above. The tool has been developed as a plugin for Eclipse, and is designed to interoperate with different modeling tools built on top of the Eclipse platform. Currently, adapters for the Papyrus UML Modeler and the UML2 Tools of Eclipse are provided.

The editor allows the user to create a Business Context Document for a particular model, and embed the diagrams of the model in the document. Model elements can be inserted via an auto-completion feature, which allows the user to select from a list of suggested items. UML model and text can be edited simultaneously, with changes to individual model elements being updated immediately in the Business Context Document. In case structural changes have been made to the model, those sentences containing text annotations are validated after the model or the text document have been saved. As already mentioned in section 3.2, changes can be detected even when the model is edited without the editor being open. A list with the detected inconsistencies is shown in a separate view, with the corresponding sentences being underlined in the editor.

7 Related Work

In this section, we discuss related approaches and compare them to our solution. We only consider research that deals with synchronized development of models and text, as opposed to transformation between those representations.

The Development Environment For Tutorials (DEFT) [21] allows including various development artifacts like source code or UML models within a document. The source artifacts are transformed into an appropriate graphical or textual representation, the system updates these representations whenever the source artifacts change. Whereas DEFT focuses on enforcing consistency on an artifact level, our approach attempts to provide fine-grained synchronization of sentences in natural language with parts of the UML model.

Literate Process Modeling (LiProMo) [18] applies the idea of Literate Modeling to the domain of Business Process Modeling. The prototypical editor allows the user to edit the actual process model and its textual description side-by-side. Thereby, model elements can be linked to passages in the textual description. These text passages are highlighted automatically when the associated model elements have been selected, and vice versa. Compared to our approach, integration is provided on a paragraph level, but no synchronization on changes on individual model elements is supported.

8 Conclusion and Future Work

In this paper, we have presented an approach to address a major problem in Literate Modeling: consistency between model and text. The problem of keeping

[2] LiMonE: `http://squam.info/limone`

UML models and text synchronized has been addressed by using text annotations and a synchronization algorithm that uses natural language processing in conjunction with OCL. The presented approach maintains a certain degree of expressional freedom for the author, as the algorithm uses a set of constraint pairs that can be easily extended to cover different writing styles. As consistency checks are made on a per-sentence basis, the presented approach is more efficient than attempting to construct a model from the text for the sake of synchronization.

An estimation of the computational complexity of our approach shows that the time required to parse one sentence is a magnitude higher than the time required to check its validity with respect to the UML model. In the current state of the prototype, only those sentences that contain references to model elements are parsed, but there are other possible optimizations that could be applied. For instance, one could only synchronize those sentences that have been edited by the user. Moreover, sentences could be synchronized on the fly while the user is typing, i.e. immediately after a sentence has been completed. If a structural change occurs in the model, only those sentences containing references to the model elements affected by the change need to be considered. To improve accuracy of the NL parser, one could investigate in more detail the relationship between UML elements and how they are related to part of speech. Moreover, the parser could be trained on an appropriate training corpus to further increase the accuracy of the generated syntax trees. At the moment, model elements have to be referred to by their name in the text, meaning pronouns that refer to model elements are not considered. Support for this case could be added by either linking the pronoun to the appropriate element in the model, or by employing anaphora resolution to determine these elements automatically.

Although neither of these optimizations will make the synchronization work perfectly accurately, we consider our approach a valuable contribution to enforce consistent documents in Literate Modeling. In this regard, an integration with a more mature framework like DEFT would be certainly promising.

References

1. Arlow, J., Emmerich, W., Quinn, J.A.: Literate Modelling — Capturing Business Knowledge with the UML. In: Bézivin, J., Muller, P.-A. (eds.) UML 1998. LNCS, vol. 1618, pp. 189–199. Springer, Heidelberg (1999)
2. Arlow, J., Neustadt, I.: Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML. The Addison-Wesley Object Technology Series. Addison-Wesley, Boston (2003)
3. Arlow, J., Neustadt, I.: Secrets of Analysis: Generative Analysis with M++, UML 2 and Literate Modelling (book, to be published)
4. Berardi, D., Cali, A., Calvanese, D., Di Giacomo, G.: Reasoning on UML class diagrams. Artificial Intelligence 168, 2005 (2003)
5. Conway, D.M.: An algorithmic approach to English pluralization. In: Proceedings of the 2nd Annual Perl Conference, San Jose, CA, USA (August 1998)
6. Cook, S., Nguyen, P.: Logical Foundations of Proof Complexity, 1st edn. Cambridge University Press, New York (2010)

7. de Marneffe, M.-C., Manning, C.D.: The Stanford typed dependencies representation. In: Coling 2008: Proceedings of the Workshop on Cross-Framework and Cross-Domain Parser Evaluation, CrossParser 2008, Manchester, UK, pp. 1–8 (August 2008)
8. Deeptimahanti, D.K., Sanyal, R.: An Innovative Approach for Generating Static UML Models from Natural Language Requirements. In: Kim, T.-h., Fang, W.-C., Lee, C., Arnett, K.P. (eds.) ASEA 2008. CCIS, vol. 30, pp. 147–163. Springer, Heidelberg (2009)
9. Denti, E., Omicini, A., Ricci, A.: *tu*Prolog: A Light-Weight Prolog for Internet Applications and Infrastructures. In: Ramakrishnan, I.V. (ed.) PADL 2001. LNCS, vol. 1990, pp. 184–198. Springer, Heidelberg (2001)
10. Denti, E., Omicini, A., Ricci, A.: Multi-paradigm Java-Prolog integration in *tu*Prolog. Science of Computer Programming 57, 217–250 (2005)
11. Dzidek, W.J.: Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance. PhD thesis, University of Oslo (2006)
12. Dzidek, W.J., Arisholm, E., Briand, L.C.: A Realistic Empirical Evaluation of the Costs and Benefits of UML in Software Maintenance. IEEE Transactions on Software Engineering 34, 407–432 (2008)
13. Harmain, H.M., Gaizauskas, R.: CM-Builder: A natural language-based CASE tool for object-oriented analysis. Automated Software Engineering 10, 157–181 (2003)
14. Klein, D., Manning, C.D.: Accurate unlexicalized parsing. In: Proceedings of the 41st Annual Meeting on Association for Computational Linguistics (ACL 2003), Sapporo, Japanvol. 1, pp. 423–430 (May 2003)
15. Lavoie, B., Rambow, O., Reiter, E.: Customizable descriptions of object-oriented models. In: Proceedings of the 5th Conference on Applied Natural Language Processing (ANLP 1997), Washington D.C., WA, USA, pp. 253–256 (March 1997)
16. Meziane, F., Athanasakis, N., Ananiadou, S.: Generating Natural Language Specifications from UML Class Diagrams. Requirements Engineering 13(1), 1–18 (2008)
17. Object Management Group. Object Constraint Language. Version 2.2 (February 2010)
18. Pinggera, J., Porcham, T., Zugal, S., Weber, B.: LiProMo — Literate Process Modeling. In: Proceedings of the 24th International Conference on Advanced Information Systems Engineering (CAiSE 2012), pp. 163–170 (2012)
19. Schulze, G.: Synchronization of UML models and narrative text using model constraints and natural language processing. Master's thesis, University of Innsbruck (June 2011)
20. Whittle, J., et al.: Empirical Assessment of the Efficacy of MDE (2010), `http://www.comp.lancs.ac.uk/~eamde/`
21. Wilke, C., Bartho, A., Schroeter, J., Karol, S., Aßmann, U.: Elucidative Development for Model-Based Documentation. In: Furia, C.A., Nanz, S. (eds.) TOOLS 2012. LNCS, vol. 7304, pp. 320–335. Springer, Heidelberg (2012)
22. Yue, T., Briand, L., Labiche, Y.: Automatically deriving a UML analysis model from a use case model. Technical Report 2010-15, Simula Research Laboratory (2010)

Model Matching for Trace Link Generation in Model-Driven Software Development

Birgit Grammel, Stefan Kastenholz, and Konrad Voigt

SAP Research Dresden,
Chemnitzer Str. 48, 01187 Dresden, Germany
{birgit.grammel,stefan.kastenholz,konrad.voigt}@sap.com

Abstract. With the advent of Model-driven Software Engineering, the advantage of generating trace links between source and target model elements automatically, eases the problem of creating and maintaining traceability data. Yet, an existing transformation engine as in the above case is not always given in model-based development, (i.e. when transformations are implemented manually) and can not be leveraged for the sake of trace link generation through the transformation mapping. We tackle this problem by using model matching techniques to generate trace links for arbitrary source and target models. Thereby, our approach is based on a novel, language-agnostic concept defining three similarity measures for matching. To achieve this, we exploit metamodel matching techniques for graph-based model matching. Furthermore, we evaluate our approach according to large-scale SAP business transformations and the ATL Zoo.

Keywords: Traceability, Model Matching, Software Quality.

1 Introduction

In the IEEE Standard Glossary of Software Engineering Terminology[1] the notion of traceability is defined as: *The degree to which a relationship can be established between two or more products of the development process, especially products having a predecessor-successor or master-subordinate relationship to one another.*

Traceability data in Model-driven Software Development (MDSD) can be understood as the runtime footprint of model transformation execution according to [7]. Essentially, trace links provide this kind of information by associating input and output elements with respect to the execution of a certain transformation rule. Trace links have a manifold application domain [11,23]: a) *System comprehension* to understand system complexity by navigating via trace links along model transformation chains; b) *Model transformation debugging* to locate bugs during the development of transformation programs and later in shipped applications; c) *Change impact analysis* to analyze the impact of a model change on existing generated output; d) *Coverage analysis* to check and ensure that all relevant parts of the input model are actually utilized by a transformation.

[1] No. Std 610.12-1990.

According to [7,28] model transformation approaches either generate trace links implicitly or explicitly. That is, in the former case, either provide an integrated support for traceability (e.g. QVT [21], MOFScript [19]) or in the latter one, rely on a developer to encode traceability as a regular output model (e.g. ATL [4], oAW [2]). Yet, these traceability solutions all depend on the existence of a transformation engine. This is only one side of the coin regarding traceability in MDSD, while taking into account: a) manually implemented transformations and b) proprietary transformation engines. Our approach tackles the problem of trace link generation for these cases. In doing so, we propose to use model matching techniques to generate trace links for arbitrary source and target models. Essentially, we base our matching process on a graph-based internal data model in accordance with typed attributed graphs. The contribution of this paper is a novel, language-agnostic concept defining three similarity measures upon which trace links are generated. It turns out that the key-enabler for leveraging model matching for trace link generation is the exploitation of metamodel matching techniques.

The content of this paper is structured into the main sections: Problem definition (cf. Section 2), our approach (cf. Section 3), evaluation (cf. Section 4) and related work (cf. Section 5). We conclude this paper in Section 6 with a summary and an outlook on future work.

2 Problem Definition

We start off to derive different possibilities on trace link generation, being relevant and necessary to MDSD in order to span the scope of our work. Essentially, three such cases are derived in Figure 1:

1. **Generation of Trace links through Transformation**: Using the integral model mapping of model transformations to derive trace links in parallel to the execution of a model transformation is a wide-spread practice [12,1,2]. The integral model mapping is directed through the transformation program's rules at model transformation runtime. According to the above-mentioned definition, traceability is the runtime footprint of transformation execution, therefore the record of the model mapping due to a model transformation.

2. **Generation of Trace links after Transformation and Processing of Input and/or Output**: The post-processing of input and output models is a common practice in MDSD [27], for model-to-model as well as for model-to-text transformations. For example, artefacts that cannot be generated automatically and thus have to be added manually, or in general due to the evolution of artefacts. While changing input and/or output model after the transformation resp. trace link generation, an update of traceability data might be necessary and thus entails the generation of trace links after transformation execution.

3. **Generation of Trace Links independent from Transformation**: The first two cases are based on the existence of a model transformation, i.e. the generation of trace links is dependent on the use of a transformation engine. Since this does not generally hold for the domain of MDSD, the third case covers the

generation of trace links, while assuming the non-existence of a transformation program. The latter case relates to either of two subcases (cf. Figure 1), that is: a) **Bridgeable transformation gap**: the transformation program is missing, yet not impossible to write (called bridgeable transformation gap in Figure 1), e.g. if a transformation was implemented manually in Java, or the transformation engine is proprietary or generally a third party component. (model-based not model-driven development as often the case at SAP) and b) **Unbridgeable transformation gap**: it is impossible to write a transformation program, since the difference in level of abstraction of potential source and target is too great to be able to be bridged through a transformation, while still preserving the semantics (called unbridgeable transformation gap in Figure 1), e.g. when mapping features to design models. Current traceability solutions in MDSD provide support for the 1. category, yet not for the 2. and 3. category which is the focus of our work. As key-enabling technique for the latter two categories, we propose model matching technniques from the field of ontology alignment and schema matching [6,22] to generate traceability data. The promising idea of this technique is its potential in the automation of trace link generation without executing a model transformation [29], which is restricted in categories 2 and 3. For a solution regarding category 1, we refer to our previous work [12]. Both above-mentioned technniques of trace link generation are incorporated into a framework, called generic traceability framework, which may be seen as the broader sense of our work.

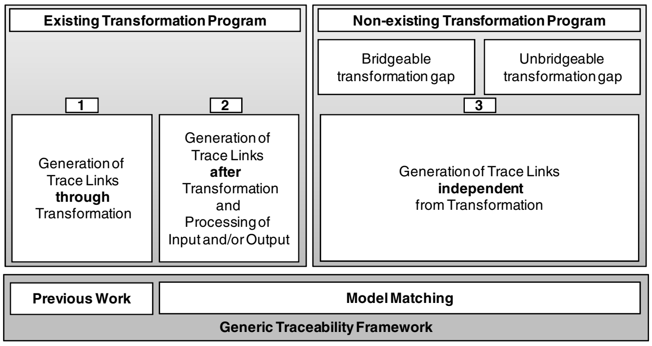

Existing Transformation Program
Non-existing Transformation Program
Bridgeable transformation gap
Unbridgeable transformation gap
1
2
3
Generation of Trace Links through Transformation
Generation of Trace Links after Transformation and Processing of Input and/or Output
Generation of Trace Links independent from Transformation
Previous Work
Model Matching
Generic Traceability Framework

Fig. 1. The Big Picture

3 Traceability Model Matching System

In this section, we describe our proposed approach on model matching in more detail, called traceability model matching system, essentially, taking two models as input and creating a mapping, that is, correspondences between model elements, as output. This mapping is further analysed with respect to extracting potential trace links. In the following, we explain the processing steps of our proposed matching system, as depicted in Figure 2:

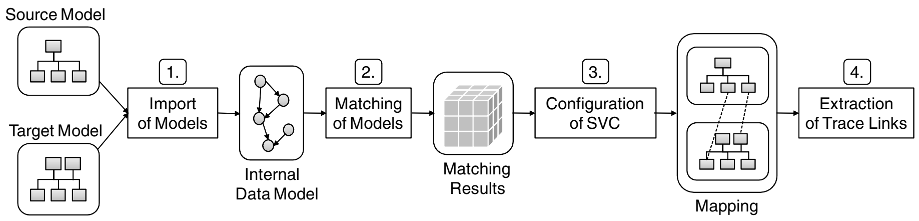


Fig. 2. Process Steps of the Traceability Model Matching System

1. **Import of Models**: The available source and target models need to be imported into a common data model, to have a common basis for arbitrary matching algorithms.
2. **Matching of Models**: Different matching algorithms are applied to the imported source and target model in order to identify model elements referring to the same conceptual entity. The matching system can be configured by choosing which matching algorithms should be involved in the matching process.
3. **Configuration of Similarity Value Cube**: Each matching algorithm provides separate results for a certain source and target model, where the results describe a similarity value for all source and target model element combinations, called similarity value matrix. All of these matrices are arranged into a cube, called similarity value cube (SVC), as depicted in Figure 2. To derive a mapping (or matches) between source and target elements out of these results, the similarity value cube needs to be configured, e.g. to form an aggregation matrix by calculating the average of similarity values, or selection matrix by selecting all elements exceeding a certain threshold.
4. **Extraction of Trace Links**: The resulting mapping is analysed and trace links are extracted according to certain heuristics, or configurations.

3.1 Import of Models

The first process step requires models to be imported into a common data model for the sake of generalisation. Alternatively, one could abstain from this and adapt matching algorithms individually, amenable to each source and target model. Yet, this would result in a higher implementation effort. Thus, we choose the first option and implement an importer for each model-specific language.

To be able to base our work on the field of graph theory to make use of graph matching algorithms, we require the internal data model to have a graph structure. We require a graph formalism on the basis of which arbitrary models can be expressed a) uniformly as graphs, yet with an adequate expressiveness and b) in relation to their corresponding metamodels since we make use of this "instance-of" relationship in our matching approach (cf. Section 3.3). The formalism of Typed Attributed Graphs holds for the above premises [9] and thus serves as the foundation of our graph construction. Please note, for those readers who wish to omit the formal definition of the theory of Typed Attributed Graphs, it is still important to know the following basics. Metamodels are represented by Attributed Typed Graphs (ATG), models by Attributed Graphs (AG). We will make use of Typed Attributed Graphs (TAG) to express the "instance-of" relationship between ATGs and AGs, since a TAG is an AG together with a special mapping from AG to ATG. This mapping is needed to render the metamodel types of certain models elements.

3.2 Running Example

To underline our matching process, we introduce an illustrative example based on a model transformation from certain entities to object-oriented class specifications. Both the source and target model with their corresponding metamodels are depicted in Figure 3 as AGs resp. ATGs. We adopt the graph notation from [9] for E-graphs. An E-graph has two different kinds of nodes, representing the graph and data nodes, and three kinds of edges, the usual graph edges and special edges used for the node and edge attribution. The solid nodes and arrows are the graph nodes V_G and edges E_G, respectively. The dashed nodes are the (used) data nodes V_D and dashed arrows represent node and edge attribute edges.

The *ATG*-Source Metamodel includes Entities, which contain Features. Both, Entities and Features, are characterized by a name of type String. The *AG*-Source Model includes an Entity called *Person* owning two Features, carrying the names, *name* and *age*. Furthermore, the *ATG*-Target Metamodel specifies Classes that consists of Fields and Methods. Again, all model elements own a name attribute of type String. The *AG*-Target Model describes a *Person* Class containing two Fields, namely *name* and *age*, and according getter methods, namely *getName* and *getAge*.

3.3 Matching of Models

Based on our internal E-graph model, we identified the following similarity measures upon which the calculation of similarity values is based on:

1. **Attribute Similarity Measure:** Similar *data nodes* from source and target graphs, indicate shared characteristics, which we refer to as attributes, and thus a potential similarity between the graph nodes the data nodes are connected to.
2. **Connection Similarity Measure:** The similarity between a *set of source and target children nodes* acts as a similarity measure. Two parent graph nodes from source and target graph with similar children graph nodes likely refer to the same entity. Thus, the connectivity of a graph node to its children graph nodes is used to propagate the similarity from child to parent node.
3. **Instance-of Similarity Measure:** We base the matching process on model level on the results of metamodel matching by making use of the "instance-of" relation. Thus, we investigate the outcome of propagating the similarity of metamodel elements to their conforming model elements.

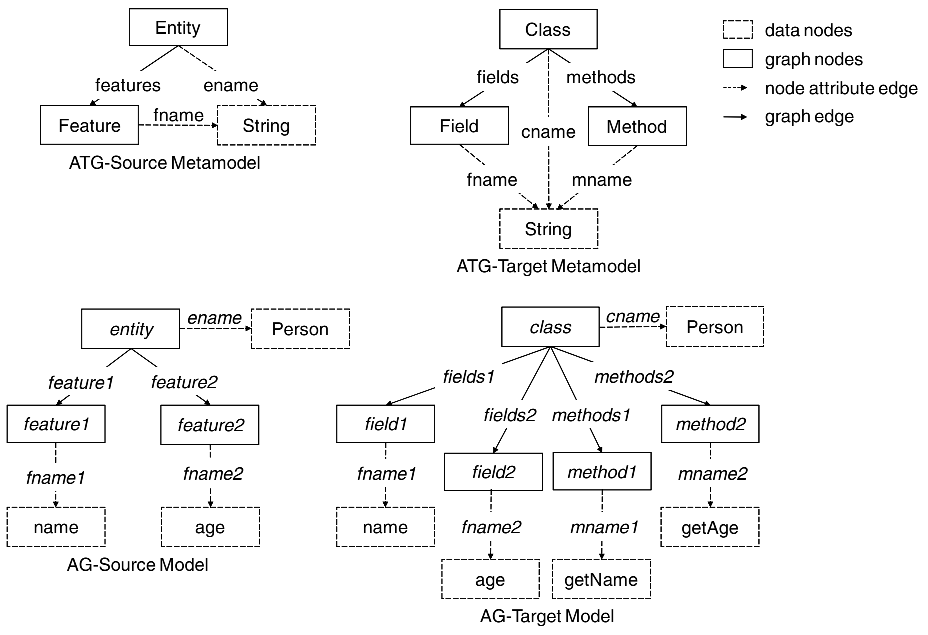


Fig. 3. Illustrative Example

In the next subsections, we propose a configurable base matching algorithm making use of the above-mentioned similarity measures. Depending on the chosen similarity measure, the algorithm's functionality is defined.

Model Matching. The model matching process with regard to the first two similarity measures is outlined in Alg. 1 and 2. The core idea of Alg. 1 is to match an attributed source and target graph and to return a mapping between corresponding source and target graph nodes. The procedure MATCHGRAPHS (line $1-8$) matches two attributed graphs AG^1 and AG^2 on the basis of the $similarityMeasure$ from the set $\{\text{ATTRIBUTES}, \text{CONNECTIONS}\}$. In either way the Cartesian product of source and target graph nodes is calculated (line $2-3$), thereby assigning to each Cartesian pair (s_i, s_j) a similarity value through the $matchNodes$ function (line 4 resp. $9-16$) and depending on the similarity measure chosen in line 1. Accordingly, all similarity values are arranged in a similarity value matrix denoted by $SIM_{|V_G^1||V_G^2|}$ (line 4). The RETRIEVEMATCHES function (line 7) receives $SIM_{|V_G^1||V_G^2|}$ as input and renders a mapping between source and target graph nodes. In case the similarity measure ATTRIBUTES is used, the similarity value assigned to a certain Cartesian pair (s_i, t_j) is calculated through the MATCHATTRIBUTES function (Alg. 2, line $4-13$). In particular, the set of data nodes of s_i as well as t_j is rendered (line $5-6$), denoted by $Nodes^{ATT}_{source}$ resp. $Nodes^{ATT}_{target}$, and their Cartesian product is calculated (line $7-8$). For each such Cartesian pair, the degree of similarity is calculated through the COMPUTESIMILARITY function (line 12 resp. $1-3$) and placed into a sim-

Algorithm 1. Matching of two attributed graphs

```
Require: $similarityValues = \{r | r \in \mathbb{R} \text{ and } 0 \leq r \leq 1\} \cup \{\text{UNKNOWN}\}$
Ensure: $similarityMeasure \in \{\text{ATTRIBUTES}, \text{CONNECTIONS}\}$

 1: procedure MATCHGRAPHS($AG^1, AG^2, similarityMeasure$)
 2:     for all $s_i \in V_G^1$, $i = \{1, \dots, |V_G^1|\}$ do
 3:         for all $t_j \in V_G^2$, $j = \{1, \dots, |V_G^2|\}$ do
 4:             $SIM_{|V_G^1||V_G^2|} \ni sim_{ij} \leftarrow matchNodes(s_i, t_j, similarityMeasure)$
 5:         end for
 6:     end for
 7:     $matches \leftarrow retrieveMatches(SIM_{|V_G^1||V_G^2|})$
 8: end procedure

 9: function MATCHNODES($s_i, t_j, similarityMeasure$)
10:     if $similarityMeasure = \text{ATTRIBUTES}$ then
11:         return $matchAttributes(s_i, t_j)$
12:     else if $similarityMeasure = \text{CONNECTIONS}$ then
13:         return $matchConnectedNodes(s_i, t_j)$
14:     end if
15:     return UNKNOWN
16: end function
```

ilarity matrix $SIM^{ATT}_{Max(k)Max(l)}$ (line 9). The resulting similarity matrix is reduced to a single similarity value by applying the set similarity function called *computeSetSimilarity*[2] (line 12) as similarity value between the two given graph nodes (s_i, t_j).

Alternatively, the similarity of two graph nodes is calculated on the basis of the similarity measure CONNECTIONS and thus, the *matchConnectedNodes* function (line $14 - 23$) is called. The function takes as input a Cartesian pair (s_i, t_j) from Alg. 1, renders the set of all children graph nodes from s_i (line 15) resp. from t_j (line 16), denoted by $Nodes^{CON}_{source}$ resp. $Nodes^{CON}_{target}$ and calculates the Cartesian product of these sets (line $17 - 18$). For each such Cartesian graph node pair a similarity value is calculated on the basis of the function MATCHATTRIBUTES and placed into a similarity matrix $SIM^{CON}_{Max(p)Max(q)}$ (line 19). Finally, the *computeSetSimilarity* reduces this matrix to a single similarity value holding for the two given initial graph nodes (s_i, t_j).

Example: In the following, we demonstrate model matching on the basis of the similarity measures ATTRIBUTES and CONNECTIONS. We match the two AGs depicted in Figure 4. First the Cartesian product of the set of all source graph nodes ($entity, feature1, feature2$) and the set of all target graph nodes ($class, field1, field2, method1andmethod2$) is calculated, yielding a similarity

[2] We assume the computeSetSimilarity function to calculate a single similarity value from the matrix of similarity values. In general, there are numerous ways of implementation.

Algorithm 2. Similarity of two graph nodes

```
Require: $compare : D_s^1 \times D_s^2 \rightarrow similarityValues$
Ensure: $dataNode_{source} \in D_s^1 \subseteq V_D^1$ and $dataNode_{target} \in D_s^2 \subseteq V_D^2$

 1: function COMPUTESIMILARITY($dataNode_{source}, dataNode_{target}$)
 2:     return $compare(dataNode_{source}, dataNode_{target})$
 3: end function
 4: function MATCHATTRIBUTES($s_i, t_j$)
 5:     $Nodes^{ATT}_{source} \leftarrow \{target^1_{NA}(e) | e \in E^1_{NA}$ and $source^1_{NA}(e) = s_i\}$
 6:     $Nodes^{ATT}_{target} \leftarrow \{target^2_{NA}(e) | e \in E^2_{NA}$ and $source^2_{NA}(e) = t_j\}$
 7:     for all $s_{i_k} \in Nodes^{ATT}_{source}$ , $k = \{1, \ldots, |V_D^1|\}$ do
 8:         for all $t_{j_l} \in Nodes^{ATT}_{target}$, $l = \{1, \ldots, |V_D^2|\}$ do
 9:             $SIM^{ATT}_{Max(k)Max(l)} \ni sim_{i_k j_l} \leftarrow computeSimilarity(s_{i_k}, t_{j_l})$
10:         end for
11:     end for
12:     return $computeSetSimilarity(SIM^{ATT}_{Max(k)Max(l)})$
13: end function

14: function MATCHCONNECTEDNODES($s_i, t_j$)
15:     $Nodes^{CON}_{source} \leftarrow \{target^1_G(e) | e \in E^1_G$ and $source^2_G(e) = s_i\}$
16:     $Nodes^{CON}_{target} \leftarrow \{target^2_G(e) | e \in E^2_G$ and $source^2_G(e) = t_j\}$
17:     for all $s_{i_p} \in Nodes^{CON}_{source}$, $p = \{1, \ldots, |V_G^1| - 1\}$ do
18:         for all $t_{j_q} \in Nodes^{CON}_{target}$, $q = \{1, \ldots, |V_G^2| - 1\}$ do
19:             $SIM^{CON}_{Max(p)Max(q)} \ni sim_{i_p j_q} \leftarrow matchAttributes(s_{i_p}, t_{j_q})$
20:         end for
21:     end for
22:     return $computeSetSimilarity(SIM^{CON}_{Max(p)Max(q)})$
23: end function
```

matrix of 15 cells. For the similarity measure ATTRIBUTES, the MATCHATTRIBUTES function is called. For the sake of simplicity, we assume the *computeSimilarity* function to calculate string similarity values on the basis of the following definition: Given the data sorts $D_s^1 = D_s^2 = String$ (where a sort is defined as the set of data nodes of a certain label type, i.e. *String*) with $v \in D_s^1$ and $w \in D_s^2$, we define the function $compare : D_s^1 \times D_s^2 \rightarrow similarityValues$ such that the following holds:

$$compare(v, w) = \begin{cases} 1 & \text{if } v = w \\ 0.5 & \text{if } v \subseteq w \\ 0 & \text{if } v \neq w \end{cases}$$

Thus, for the pair $(entity, class)$ the data node *Person* is retrieved for the source graph nodes *entity* as well as for the target graph node *class*. Since their labels (i.e. strings) are identical, a similarity value of 1 is assigned to the data node pair $(Person, Person)$. Since the resulting similarity value matrix SIM^{ATT}_{11} (Alg. 2 line 12) contains only one cell with the similarity value 1, the *computeSetSimilarity* function assigns the same value to the node pair

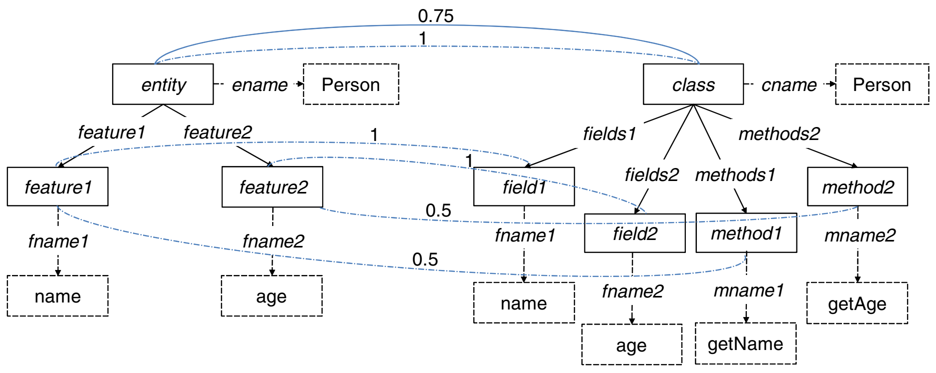


Fig. 4. Model Matching Mappings for AG-Source Model and AG-Target Model

$(entity, class)$. For the pair $(feature1, class)$ a similarity value of 0 is calculated, since the labels of the data nodes *name* and *person* are unidentical. In analogy, the mappings of the other graph nodes are calculated, as depicted by the dashed lines with an according similarity value in Figure 4. In case the similarity values are calculated on the basis of the similarity measure CONNECTIONS, the MATCHCONNECTEDNODES function is invoked. This entails the retrieval of all children graph nodes per Cartesian graph node pair. For example, the graph nodes $feature1, feature2$ resp. $field1, field2, method1, method2$ are retrieved for the source graph node *entity* resp. target graph node *class*. Traversing the Cartesian product of the retrieved sets, we assign to each Cartesian pair a similarity value on the basis of the similarity measure ATTRIBUTES as described above. For this example, we assume for the *computeSetSimilarity* function to reduce the similarity matrix to a similarity value by taking the average of all matching results from corresponding source-target children nodes. Thus a similarity value of 0.75 is calculated and propagated to the node pair $entity, class$, as depicted by the full line in Figure 4. Afterwards, the similarity of all remaining Cartesian graph nodes pairs are calculated analogously. Since $feature1$ and $feature2$ do not hold any related children graph nodes, no further similarity values are calculated.

Metamodel-Driven Model Matching. In the following section, we introduce an algorithm to realize the idea of how mappings due to metamodel matching may be used for improving model matching results. Essentially, this improvement can be achieved by verifying or rejecting found matches based on the metamodel mapping. This idea is called metamodel-driven model matching and implements the third similarity measure, called INSTANCEOF. For this purpose, we extend the above-mentioned algorithms by the similarity measure INSTANCEOF. To avoid repetition, we only describe the extended part of the algorithm. In case the INSTANCEOF parameter is used, the MATCHNODETYPES function is invoked, which is defined in Alg. 3. The MATCHNODETYPES function takes as input a certain graph node pair (s_i, t_j) from Alg. 1,

Algorithm 3. Similarity of two typed graph nodes

```
Ensure: $s_i \in V_G^1$, $Node_{source}^{ATG\_Type} \in \mathcal{V}_G^1$, $t_j \in V_G^2$ and $Node_{target}^{ATG\_Type} \in \mathcal{V}_G^2$
Ensure: $typeSimilarityMeasure \in similarityMeasure \setminus \{\text{INSTANCEOF}\}$

1: function matchNodeTypes($s_i, t_j$)
2:     $Node_{source}^{ATG\_Type} \leftarrow t^1(s_i)$
3:     $Node_{target}^{ATG\_Type} \leftarrow t^2(t_j)$
4:     return $matchNodes(Node_{source}^{ATG\_Type}, Node_{target}^{ATG\_Type}, typeSimilarityMeasure)$
5: end function
```

thus working on respective source and target AGs. For each such pair (s_i, t_j) the corresponding graph nodes $Node_{source}^{ATG\_Type}, Node_{target}^{ATG\_Type}$ of source and target ATGs are returned by the attributed graph morphism t (line $2-3$). For each pair $Node_{source}^{ATG\_Type}, Node_{target}^{ATG\_Type}$ a similarity value is calculated through the matchNodes function (Alg. 3, line 4), the same way as in Alg. 1. As a consequence, the similarity value of $Node_{source}^{ATG\_Type}, Node_{target}^{ATG\_Type}$ is propagated to (s_i, t_j) (Alg. 1, line 4). Please note, we restrict the use of the similarity measure INSTANCEOF to avoid potential cycles.

Example. We illustrate the similarity measure INSTANCEOF according to our running example and assume a given metamodel mapping according to Alg. 1. The metamodel mapping is depicted in the upper layer of Figure 5 with a mapping from *Entity* to *Class* as well as from *Feature* to *Field* and *Method*. For each Cartesian pair the matchNodeType function is invoked. For the graph nodes *entity* resp. *class*, the attributed graph morphism t yields the graph nodes *Entity* resp. *Class* (being the metmodel types) Since a mapping exists between *Entity* and *Class*, the similarity value of 1 is propagated to the pair $(entity, class)$. Analogously, the similarities of the other Cartesian pairs is calculated as depicted by the mappings in Figure 5.

Implementation. We based our implementation on the metamodel matching framework, called Matchbox [31]. This framework is build up on the SAP Auto Mapping Core, an implementation inspired by COMA++ [8], a schema matching framework. The reason for choosing Matchbox is its language genericity and broad scope of optimized metamodel matchers, fully aligned with our conceptual work on deriving three similarity measures for model matching. Essentially, we adopted seven matchers from Matchbox: Name and NamePath matcher in terms of the Attribute matcher; Children, Parent, Leaf, Sibling, Graph Edit Distance [30] and Pattern matcher [20] being structural matchers as variations of the Connection matcher. Furthermore, we implemented an Instance-of matcher in analogy to the data type matcher. Apart from "instance-of" matching, we investigated blocking techniques [14] as part of metamodel-driven model matching. In our case blocking is based on constraining the matching process to model elements whose metamodel elements match (provided a correct and complete

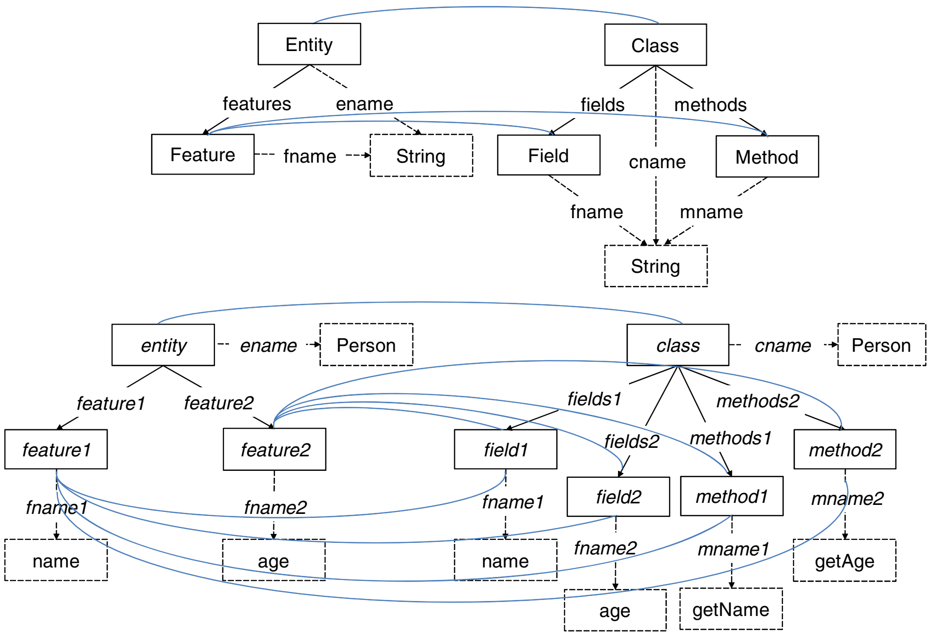


Fig. 5. Mappings of Metamodel-driven Model Matching

metamodel mapping). As a consequence, mapping results are more likely to be correct and complexity is reduced. The difference between instance-of matching and blocking is the point in time. The former takes place during the matching process, while the latter in advance.

4 Evaluation

Our evaluation is based on 41 model transformations from the ATL Zoo[3] as well as a SAP business application, called Sales Scenario (SalesS) [10]. The motivation for these scenarios stems from our requirements regarding the mapping tasks: a) source and target models conforming to the same as well as different metamodels, b) broad language scope and c) application domain. While the ATL Zoo covers model-to-model transformations, the SalesS covers Xpand [2] model-to-text transformations, thus fulfilling a). Both mapping scenarios complement each other w.r.t. the language scope resp. applicability. In the SalesS (source and target model size: $72-1231$ resp. $95-2593$ model elements) domain-specific models are transformed to Java source code in order to generate a complete business application through corresponding large-scale transformations, on the other hand the ATL Zoo (source and target model size: $41 - 3253$ resp. $14 - 1813$ model elements) comprises common languages like UML, XML, and KM3 as well as

[3] http://www.eclipse.org/m2m/atl/atlTransformations/

Table 1. Summary of average results for Sales Scenario and ATL Zoo

		Default	**Profile**	**Blocking^D**	**Blocking^P**	**InstanceOf**
SalesS	Recall	0.500	0.291	0.193	0.787	0.136
	Precision	0.174	0.539	0.556	0.971	0.104
	Fmeasure	0.204	0.338	0.257	0.851	0.072
ATL Zoo	Recall	0.144	0.261	0.366	0.885	0.143
	Precision	0.724	0.734	0.951	0.975	0.170
	Fmeasure	0.210	0.313	0.477	0.901	0.084

domain-specific ones with transformations ranging from technical space bridges over refactorings to model refinements.

4.1 Setup

To measure the quality of our matching results, we chose measures from information retrieval, namely: precision, recall and F-measure [24]. Secondly, our evaluation is based on the brute force [17] method, entailing the variation of all parameters w.r.t. to their values to gain all possible configurations upon which to choose the qualitative best results. We adopted the parameters from [31] regarding the strategies, *Aggregation*, *Selection*, *Direction* and a *Combination* of them. The goldmapping retrieval for the ATL Zoo is based on the ATL higher order transformation [13], while for the Sales Scenario the connector-based traceability extraction from [12] was implemented.

4.2 Quality

With our evaluation, we answer the following questions: a) What is the average quality of our matching results? b) What is the best quality per mapping? and c) What is the influence of metamodel-driven matching?

For a) the average F-measure for a certain configuration over all SalesS and ATL Zoo mappings is examined. Based on the latter, configurations providing the maximum average F-measure are identified as default configurations. It turns out, that the SalesS and ATL Zoo are characterized by significantly different default configurations. Regarding the matcher combination the results are particularly striking. While the *Attribute Matcher* is most successful for the SalesS, the *GraphEditDistanceMatcher* best accounts for the ATL Zoo. The average results pertaining to these configurations are listed in Table 1 under default.

Regarding b) the configurations with the highest F-measure for each mapping are considered as depicted in Figure 6 for the SalesS (top) and for the ATL Zoo (two bottom rows). In general, the average of this highest F-measure (called profile in Table 1) increases by a factor between $1.5 - 1.7$ in comparison to the default configuration.

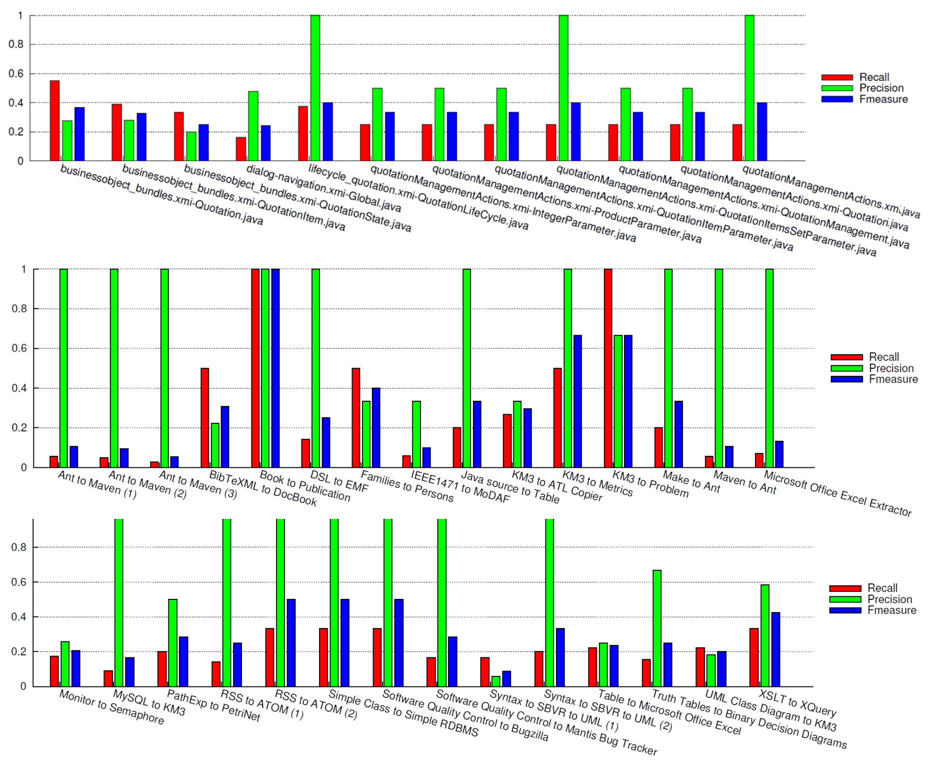


Fig. 6. Maximum of Precision, Recall and F-measure against Mappings

Regarding c), metamodel-driven blocking performed on the default configuration mappings (denoted with Blocking^D) improves matching results by a factor of 1.3 – 2.3 w.r.t. the default configuration. Applying blocking on the profile mappings (cf. Blocking^P) even improves results by a factor of 4.2 – 4.3. Moreover, Instanceof turns out to be effective only for models with a small amount of instances per metamodel element.

4.3 Results

In order to successfully conduct traceability scenarios (cf. Section 1), the collected traceability data needs to account for an adequate data expressiveness. The above results show that metamodel-driven matching effective as *blocking* is the key-enabler for leveraging model matching for trace link generation by raising the matching quality up to an F-measure of 0.9. In doing so, our approach achieves a traceability data expressiveness of approximately 90% with respect to trace link correctness and completeness.

5 Related Work

In the following, we give an overview on related work, dealing with model resp. metamodel matching. None of the following approaches have addressed nor tackled the problem of generating trace links on the basis of model and/or metamodel matching with the same language genericity, quality (precision and recall) of acquired matches and evaluation (authenticity and model size of evaluation scenario(s)) as our approach.

Model matching is related to the field of schema matching and ontology matching (also called alignment). Yet, the ideas of these matching approaches are based on finding correspondences between source and target being on the same abstraction level. For a detailed survey we refer to [6,22,26]. Our approach additionally allows for source and target models to conform to different metamodels, thus allowing models to be on different levels of abstraction. The same applies to entity matching, which focuses on identifying entities (objects, data instances) referring to the same real-world entity for the sake of data integration and data cleaning [14].

More closely related is the field of metamodel matching [31]. Yet, these approaches do not account for model-specific matching requirements, such as, leveraging on model-specific attributes for matching, containment semantics, import mechanism from a model into a tree structure etc.

A range of technologies are available for differencing and comparing models [16], both being amenable to model matching. For the following approaches, we differentiate between intra-matching and inter-matching. Our work makes a point of finding matches between source and target models conforming to the same (referred to as intra-matching) as well as different (referred to as inter-matching) metamodels. Additionally, we exploit metamodel-driven matching techniques.

One possibility to computing model differences is static identity-based matching. For example, in [3] a metamodel-independent algorithm is proposed to calculate the difference and union of MOF-based models in the context of a version control system. This approach necessitates a closed development environment in which all model editors and other tools which modify models assign and maintain a persistent unique identifier at each model element. In such a context, one can efficiently compute differences on the basis of persistent unique identifiers. Yet, this approach only applies to mappings, where source and target conform to the same metamodel and neither to models constructed independently from each other, nor technologies that do not support maintenance of unique identifiers.

The Epsilon Comparison Language (ECL) [15] is a metamodel-agnostic and technology-independent rule-based comparison language, accounting for model comparison as foundation for model composition and model transformation testing. Instead of relying on an internal data model, on which different similarity measures are applied, ECL requires the user to specify domain-specific match rules. These rules are dedicated to a certain source-target metamodel combination and specify matchers (in terms of our approach) with the help of a model query language (EOL). In contrast to our matching system, this approach depends on metamodel-specific information to a high degree, yet on the other hand, benefits from (potentially) more accurate matching results.

SiDiff [25] is based on a generic difference algorithm for UML models, supporting the three major state of the art matching strategies, i.e. ID-based, signature-based and similarity-based matching. The internal data model basically is a tree with typed elements that can be decorated with attributes and additionally may have graph-like cross-references. In contrast to our graph model, this data model is limited to containment (and reference) relationships and lacks e.g. representations of inheritance or instantiation. SiDiff uses an intra-matching approach with an algorithm traversing a tree bottom up and top down similar to our *Connection* similarity measure propagation. However, SiDiff is language-specific, while our approach is a configurable language-agnostic matching framework. On the other hand, SiDiff makes use of UML-characteristis, by weighting similarity results according to element types and therefore evaluation results are likely to outperform our aproach in this certain domain.

DSMDiff [18] provides for UML-specific intra-matching by using type information of metamodels to encode characteristics in strings being compared. The structural information used is as simple as the number of children or references.

EMFCompare [5] provides an out-of-the-box model comparison and merge support and is the closest to our approach. The generic matching and differencing engine are metamodel agnostic, thus support inter-matching in terms of Ecore. Regarding the matching engine, matches are calculated through a simple algorithm which computes label and string edit distances. The comparative evaluation between EMFCompare and our matching framework based on our evaluation scenarios showed that EMFCompare identified correct matches for 29% of the available source-target combinations, with no correspondences in the Sales Scenario. The resulting average F-measure of 0.1 is doubled resp. tripled by the default configuration resp. profiles of our matching system. When applying metamodel-driven matching techniques, in particular, blocking, EMFCompare is outperformed by a factor of at least 4.7.

6 Conclusion and Future Work

In this paper, we proposed a traceability model matching system for generating trace links for arbitrary source and target models in MDSD. The above system accounts for the trace link generation categories 2 and 3 (cf. Section 2), that is, in case a transformation engine is non-existing and thus, the integral model mapping cannot be leveraged on, and/or in case the source and target models have evolved after transformation execution. The methodology to develop the matching system is founded on the idea of using a graph-based internal data model based on Typed Attributed Graphs upon which the matching process takes place. On grounds of this data model, we derived three similarity measures for the sake of model matching. Furthermore, the implementation of our approach is build upon the metamodel matching framework, Matchbox, which we extended to eight matchers aligned with the three derived similarity measures.

The evaluation of our approach is based on the ATL Zoo and a SAP business application. The results show that configuration profiles achieve $1.5 - 1.7$ times better matching results w.r.t. default configurations. Metamodel-driven blocking on default configurations resp. on profile configurations improves matching results by a factor of $1.3 - 2.3$ resp. even $4.2 - 4.3$.

For future work, we envision to improve the current set of matchers and the integration of more language-specific matchers in order to optimize default configurations, requiring no configuration effort for the user. Secondly, we investigate the scalability for matching large-scale models (i.e. UML models), e.g. through clustering. Furthermore, we look into the automatic derivation of configuration profiles and their relation to similar matching scenarios.

References

1. MOFscript, http://www.eclipse.org/gmt/mofscript/
2. openArchitectureWare, http://www.eclipse.org/gmt/oaw/
3. Alanen, M., Porres, I.: Difference and Union of Models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 2–17. Springer, Heidelberg (2003)
4. Atlas Transformation Language, http://www.eclipse.org/m2m/atl/
5. Brun, C., Pierantonio, A.: Model Differences in the Eclipse Modeling Framework. European Journal for the Informatics Professional (UPGRADE) IX (2008)
6. Choi, N., Song, I., Han, H.: A Survey of Ontology Mappings. Sigmod Rec.h. No. 35(3), 34–41 (2006)
7. Czarnecki, K., Helsen, S.: Feature-based Survey of Model Transformation Approaches. IBM Systems Journal 45(3) (2006)
8. Do, H.H.: Schema Matching and Mapping-based Data Integration. VDM Verlag Dr. Mueller e.K (2006)
9. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (2006)
10. Elsner, C., Jaeger, M., Fiege, L., Schwanninger, C., Grammel, B.: Deliverable D5.4: Description of all case studies. Tech. rep., AMPLE (2009)
11. Goran, K., Oldevik, J.: Scenarios of Traceability in Model to Text Transformations. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA. LNCS, vol. 4530, pp. 144–156. Springer, Heidelberg (2007)
12. Grammel, B., Kastenholz, S.: A Generic Traceability Framework for Facet-based Traceability Data Extraction in Model-driven Software Development. In: Proceedings of the ECMFA Traceability Workshop (2010)
13. Jouault, F.: Loosely Coupled Traceability for ATL. In: Proceedings of the ECMDA Traceability Workshop (2005)
14. Koepcke, H., Rahm, E.: Frameworks for Entity Matching: A Comparison. Data and Knowledge Engineering (2009)
15. Kolovos, D., Paige, R.F., Polack, F.A.C.: Model Comparison: A Foundation for Model Composition and Model Transformation Testing. In: Proceedings of the International Workshop on Gobal Integrated Model Management (2006)
16. Kolovos, D., Ruscio, D., Pierantonio, A., Paige, R.: Different Models for Model Matching: An Analysis of Approaches to support Model Differencing. In: Proceedings of CVSM 2009 (2009)

17. Lewis, H., Denenberg, L.: Data Structures and Their Algorithms. Addison Wesley (1991)
18. Lin, Y., Gray, J., Jouault, F.: DSMDiff: A Differentiation Tool for Domain-specific Models. European Journal of Information Systems 16, 349–361 (2007)
19. MOFscript, http://www.eclipse.org/gmt/mofscript/
20. Mucha, P.: Musterbasiertes Abbilden von Metamodellen. Master's thesis, TU Dresden (2010)
21. Object Management Group: MOF 2.0 Query View Transformation (ad/2005-03-02) (2005)
22. Rahm, E., Bernstein, P.: A Survey of Approaches to Automatic Schema Matching. The VLDB Journal. No. 10(4), 334–350 (2001)
23. Ramesh, B., Jarke, M.: Towards Reference Models for Requirements Traceability. IEEE Transactions on Software Engineering 21 (2001)
24. Rijsbergen, C.J.V.: Information Retrieval. Butterworth-Heinemann (1979)
25. Schmidt, M., Gloetzner, T.: Constructing Difference Tools for Models using the SiDiff Framework. In: Proceedings of ICSE (2008)
26. Shvaiko, P., Euzenat, J.: A Survey of Schema-Based Matching Approaches. In: Spaccapietra, S. (ed.) Journal on Data Semantics IV. LNCS, vol. 3730, pp. 146–171. Springer, Heidelberg (2005)
27. Stahl, T., Voelter, M.: Model Driven Software Development. John Wiley & Sons (2006)
28. Vanhooff, B., Baelen, S.V., Joosen, W., Berbers, Y.: Traceability as Input for Model Transformations. In: Proceedings of the ECMDA Traceability Workshop (2007)
29. Voigt, K.: Semi-automatic Matching of Heterogeneous Model-based Specifications. In: Proceedings of Software Engineering Workshop (2010)
30. Voigt, K., Heinze, T.: Metamodel Matching Based on Planar Graph Edit Distance. In: Tratt, L., Gogolla, M. (eds.) ICMT 2010. LNCS, vol. 6142, pp. 245–259. Springer, Heidelberg (2010)
31. Voigt, K., Ivanov, P., Rummler, A.: MatchBox: Combined Metamodel Matching for Semi-automatic Mapping Generation. In: Proceedings of the SAC 2010 (2010)

Matching Business Process Workflows across Abstraction Levels

Moisés Castelo Branco[1], Javier Troya[2], Krzysztof Czarnecki[1], Jochen Küster[3], and Hagen Völzer[3]

[1] Generative Software Development Laboratory, University of Waterloo, Canada
{mcbranco,kczarnec}@gsd.uwaterloo.ca
http://gsd.uwaterloo.ca

[2] Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Spain
javiertc@lcc.uma.es

[3] IBM Research Zurich, Switzerland
{JKU,HVO}@zurich.ibm.com

Abstract. In Business Process Modeling, several models are defined for the same system, supporting the transition from business requirements to IT implementations. Each of these models targets a different abstraction level and stakeholder perspective. In order to maintain consistency among these models, which has become a major challenge not only in this field, the correspondence between them has to be identified. A correspondence between process models establishes which activities in one model correspond to which activities in another model. This paper presents an algorithm for determining such correspondences. The algorithm is based on an empirical study of process models at a large company in the banking sector, which revealed frequent correspondence patterns between models spanning multiple abstraction levels. The algorithm has two phases, first establishing correspondences based on similarity of model element attributes such as types and names and then refining the result based on the structure of the models. Compared to previous work, our algorithm can recover complex correspondences relating whole process fragments rather than just individual activities. We evaluate the algorithm on 26 pairs of business-technical and technical-IT level models from four real-world projects, achieving overall precision of 93% and recall of 70%. Given the substantial recall and the high precision, the algorithm helps automating significant part of the correspondence recovery for such models.

Keywords: BPMN Matching, Consistency Management, Change Extraction.

1 Introduction

A growing number of enterprises use Model-Driven Engineering (MDE) based on Business Process Modeling (BPM) to automate their business processes. BPM typically requires collaboration of many stakeholders, including Business Analysts, Systems Analysts, IT Architects and Developers. The distribution of responsibilities among these roles usually results in the creation of several models of the same business process, each residing at a different abstraction level. These models range from business-oriented ones, which are technology-independent and easily understandable by business people, to IT-oriented ones, constructed by taking into consideration technical facilities of existing systems. Specialized modeling languages have been developed to represent such models. One such language, standardized by the OMG, is Business Process Modeling and Notation (BPMN) [11].

A key challenge in BPM is maintaining the consistency among these different models. Maintaining consistency is important in order to ensure that business-level process models are implemented correctly by executable models, and that the business-level models reflect the implemented processes correctly, for example, for auditing purposes. Checking and maintaining consistency between a business-level model and its IT-level counterpart requires knowing the correspondences between the activities in the first model and the activities in the other one. Unfortunately, such correspondences are often missing in practice since the models are created using different tools or languages. For example, business-level models are often created using some variant of BPMN, either in a dedicated BPMN tool or in a diagramming tool such as Visio, and the IT-level models are often created in executable workflow language BPEL, targeting a specific process execution engine. With the introduction of BPMN 2.0, both business- and IT-level models can be expressed using the same language, improving tool interoperability across levels of abstraction. Nevertheless, organizations having existing business- and IT-level models still face the challenge of establishing the correspondence among these models. For example, IT personnel at the Bank of Northeast of Brazil (BNB), our industry partner, has faced this challenge as part of a regulatory compliance project. In the absence of adequate tool support, this task is very tedious and time consuming.

This paper presents a heuristic matching algorithm for determining such correspondences. The algorithm is based on an empirical study of process models at BNB [1]. The study revealed frequent correspondence patterns between models spanning multiple abstraction levels, including adding or modifying the information on individual activities and changing the models structure, for example, by behavioral refactoring and adding IT-specific tasks. Consequently, our algorithm has two phases, first establishing correspondences based on similarity of model element attributes such as types and names and then refining the result based on the structure of the models. Our algorithm can recover complex correspondences relating whole process fragments of one model to such fragments in the other model—a capability needed in practice [1]. Previous work on process models has focused on either one-to-one correspondences between activities (e.g., [3]) or one-to-many correspondences relating activities and process fragments ([17]).

We evaluate the algorithm on 26 pairs of business-technical and technical-IT level models from four real-world projects, achieving overall precision of 93%

and recall of 70%. Given the substantial recall and the high precision, the algorithm helps automating significant part of the correspondence recovery for models spanning multiple abstraction levels.

The remainder of the paper is structured as follows: Section 2 provides background on BPM and important concepts used throughout the paper. Section 3 describes the heuristic algorithm using a running example. Section 4 presents the results of the empirical evaluation including threats to the validity of the work. Section 5 discusses related work on process model matching. Finally, Sect. 6 summarizes and concludes the paper.

2 Background

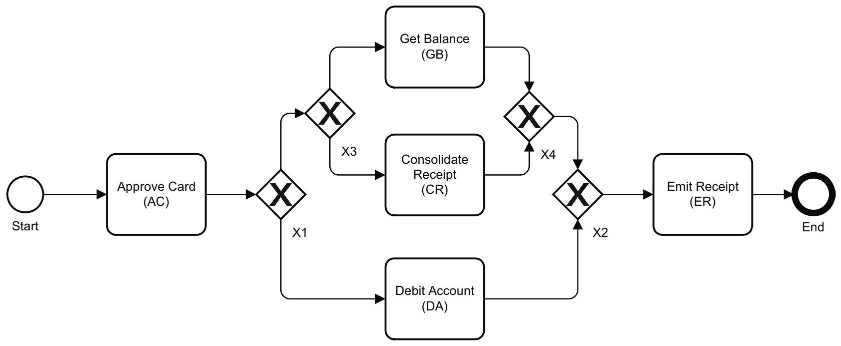
2.1 BPMN, SESE, and PST

This paper assumes that the models to be matched are expressed in BPMN 2.0 [11]. BPMN 2.0 allows businesses to represent their internal business procedures in a graphical notation and communicate them in a standard way for both documentation and execution. Models expressed in other languages, such as BPMN 1.0 and BPEL, can be translated into BPMN 2.0 without adversely impacting the information used by our algorithm (cf. Sect. 4.1). BPMN inherits and combines elements from a number of previously proposed notations, including the Activity Diagrams component of the Unified Modeling Notation (UML).

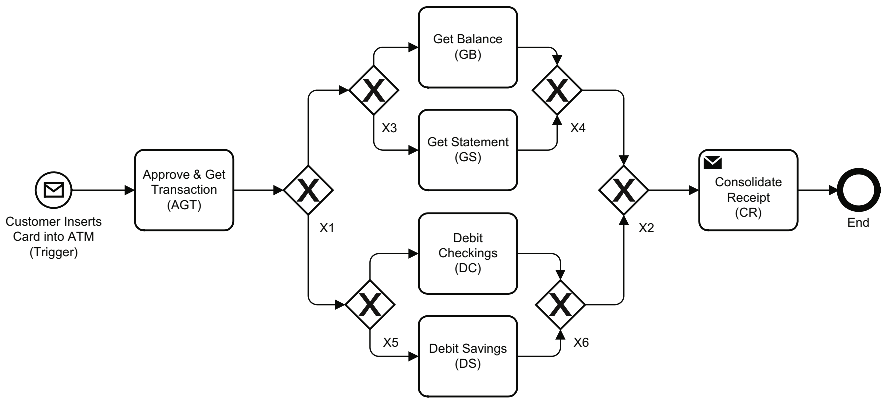
Figure 1 shows two BPMN process models. We added shorter names in parentheses (e.g., *(AC)*) to later facilitate concisely representing correspondences between the models. The notation displays activities by rounded rectangles, events by circles, gateways by diamonds, and sequence flows by arrows. Each model has a start, usually modeled by a start event (e.g., *Customer inserts Card into ATM*), a flow of activities governed by decisions (e.g., *X1*), and an end point. A larger, realistic example is given elsewhere [1].

Any workflow graph (a BPMN process model in our case) can be uniquely decomposed into single-entry single-exit (SESE) regions [15]. Let $G = (N, E)$ be a workflow graph, where N is the set of nodes and E the set of edges. A SESE region $R = (N', E')$ is a nonempty subgraph of G, i.e., $N' \subseteq N$ and $E' = E \cap (N' \times N')$ such that there exist edges $e, e' \in E$ with $E \cap ((N \backslash N') \times N') = \{e\}$ and $E \cap (N' \times (N \backslash N')) = \{e'\}$; e and e' are called the *entry* and the *exit* edge of R, respectively. According to the formal definition, a SESE region is any region in the workflow graph that has a single entry at the beginning and a single exit at the end. In this way, an activity itself is a SESE region, and so is the whole workflow graph.

The Process Structure Tree (PST) for a BPMN process model is a tree representing the decomposition of the model into SESE regions [15], similar to the much older notion of a program structure tree [9]. Figure 2 shows the PSTs corresponding to the BPMN process models. There is a unique PST for each BPMN model. The root represents the whole process model since a process model is a SESE itself. Leaves represent model elements, i.e., activities, gateways and events. Inner nodes represent SESE regions. In particular, the parent of a region R is the smallest region R' that contains R.



(a) Business Specification



(b) Technical Specification

Fig. 1. BPMN Models

2.2 Differences between Business and IT Process Models

Our target scenario involves matching business-level models specified by business analysts and the corresponding IT-level models implemented by IT specialists. IT specialists usually refine the original specification to meet technical requirements of the underlying IT infrastructure, such as invoking existing and new services, adding exception treatment, and changing the control flow to satisfy application protocols and optimize the execution. In previous work [1], we have studied over 70 models from BNB and interviewed their creators and maintainers, compiling a catalog of 11 recurrent patterns used to refine business-level models into IT-level models. These patterns include (i) adding or modifying properties of model elements, such as changing the name or type of an activity or adding service call details, and (ii) changing the flow structure. The latter category includes behavioral refinement and refactoring and adding additional behavior, such as technical exception flow. An example from category (i) is the renaming and retyping of the empty start event *Start* (Fig. 1.a) into the message-driven event *Customer inserts card into ATM* (Fig. 1.b). An example from category (ii) is the refinement of the task *Debit Account* (Fig. 1.a) into the block consisting of the gateways *X5* and *X6* and two other tasks *Debit Checkings* and *Debit Savings* (Fig. 1.b). Examples of other patterns are given in the study [1].

3 Matching Algorithm

We assume that the models to be matched represent the same process, but at different levels of abstraction, as described in Section 2.2. We also assume that, although the models are intended to be consistent, inconsistencies can occur during their evolution. Thus, the models may include inconsistencies, such as order of activities switched during refinement or business-relevant activities added to the IT-level model but not reflected in the business-level model (see [1] for other examples).

The algorithm identifies a correspondence between two models residing at different abstraction levels. The algorithm operates on the PST representations of the models. As stated in Sect. 2, leaves in a PST represent *model elements*; inner nodes represent SESE regions, or *regions*, for short. The algorithm computes a *(model) correspondence*, which is a set of *correspondence links* among PST nodes; each link connects a single node in the PST of the first model with a single node of the PST of the other model. Thus, our algorithm is able to identify correspondence links of different cardinality with respect to model elements: *1:1* (link among two model elements or two regions with only one model element each), *1:n* (link between a region with one model element in the first PST and a region with more than one model elements in the second PST), and *m:n* (link between regions with more than one model element each).

Our algorithm has two phases: *attribute matching* and *structure matching.* The first phase deals with the search of correspondence links based on the attributes of model elements such as names and types; the second phase tries to find correspondence links based purely on the structures of the PSTs and the links established in the first phase. Note that the first phase also considers the structure of the PSTs since it matches both model elements and regions. The next subsection presents the similarity measures for model elements and regions. The following two subsections explain the two matching phases using the running example from Fig. 1. The pseudo-code of the algorithm is available at `http://gsd.uwaterloo.ca/matchingbpm`.

3.1 Matching Criteria for Model Elements and Regions

Our algorithm uses two attribute matching criteria for PSTs: one for matching individual model elements and another for matching regions. We adapted them from previous work on matching source code represented as abstract syntax trees (ASTs) [6]. The original criteria use *bigram string similarity* to match the values of AST leaves and inner nodes. Fluri et al. [6] achieved better results for source code matching using Dice Coefficient with bigrams as string similarity compared to other measures such as the Levenshtein Distance [10]. In particular, the bigram-based similarity tolerates word re-orderings, which also occur in process refinement (e.g., ApproveCard vs. CardApproval).

We have adapted the original matching criteria by Fluri et al. to the process matching context, by using the information available in PSTs and refining the criteria based on experiments with sample models. In particular, we require exact matches for model elements and use bigram similarity only for inner nodes (regions). The reason is that process model elements have often relatively short names, and the names can be very similar, although representing completely different functions (e.g., ApproveCredit, ApproveContract, CreditAccount). The resulting criteria are as follows:

Matching criterion for model elements

$$match_e(n,m) \triangleq (type(n) = type(m)) \wedge (name(n) = name(m))$$

Matching criterion for regions

$$match_r(r,s) \triangleq (\frac{common(r,s)}{max(r,s)} \geq l) \wedge (sim_{2g}(value(r), value(s)) \geq f)$$

where

type returns the type of the model element as a numeric code, such as 0 for start event, 1 for task, 2 for exclusive gateway, etc.

name returns the name of the model element, for example: *Get Balance*, *Debit Savings*, etc.

sim_{2g} calculates the bigram-based similarity of two strings [6]; it returns a numeric value between *0* and *1*, where *1* means that the strings are equal.

value returns the string formed by the concatenation of the names and types of all model elements of a region. Thus, similarity of names is emphasized, since types are short numeric codes and names are typically complete words.

common returns the number of pairs of model elements of the two regions that match exactly (i.e. $match_e$ is *true*).

max returns the maximum number of distinct pairs that could be matched (i.e., the number of all model elements in the smaller region).

f and ***l*** are thresholds controling the algorithm. We obtained the best results in our evaluation with *0.6* and *0.4*, respectively.

3.2 Attribute Matching

Let us explain the first phase by applying it to the PSTs in Fig. 2 obtained from the models in Fig. 1.

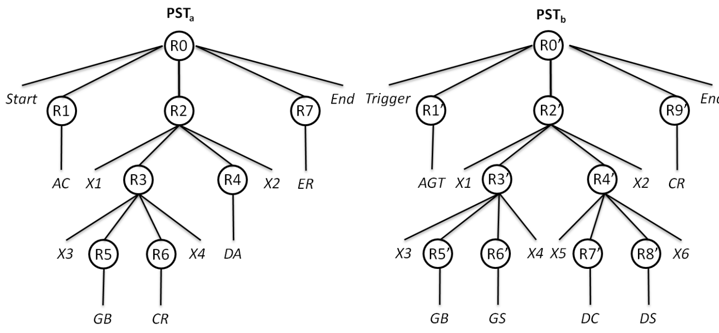


Fig. 2. PSTs representation of the business process models

First, the algorithm assumes that the roots of both PSTs correspond to each other. Then, the algorithm performs a depth-first traversal in one of the PSTs in order to establish correspondence links with the second PST. Starting with region *R1* in PST_a, it tries to find a corresponding region in PST_b. According to the matching criterion for regions (cf. Sect. 3.1), a necessary condition for a match is to satisfy the formula $\frac{common(R1,X)}{max(R1,X)} \geq l$ with any region X in PST_b. Since $R1$ has only one child (a model element), satisfying the formula requires finding a region in PST_b containing a model element with exactly the same name and type (matching criterion for model elements) as the activity *Approve Card*. Since there is none, the algorithm proceeds to region $R2$.

For $R2$, the algorithm finds $R2'$ in PST_b to satisfy the above formula ($\frac{5}{6} \geq 0.4$). The algorithm also checks that $sim_{2g}(value(R2), value(R2')) \geq f$ is satisfied. Assuming abbreviations, $value(R2)$ returns *X12X32GB1CR1X42DA1X22*; $value(R2')$ returns *X12X32GB1GS1X42X52DC1DS1X62X22*. Both strings have a similarity of around 0.65 (assuming full names). The algorithm then keeps on searching more matches for $R2$ in PST_b. The formula $\frac{common(R2,R3')}{max(R2,R3')} \geq l$ is also satisfied, returning $\frac{3}{4}$; however, the value obtained from the string comparison, 0.51, is smaller that f, so $R3$ is discarded as a match (see left figure in Fig. 3, where the top link is selected and the bottom one is discarded). No other region in PST_b satisfies the matching criterion with $R2$; however, if there were several matching regions in PST_b, the correspondence link would be established with the region with the highest string similarity to $R2$. If there are more than one region with the same highest string similarity to $R2$ (unlikely though, because copies are uncommon in process modeling), one of them is chosen arbitrarily.

The algorithm keeps traversing PST_a and establishes a correspondence link between $R3$ and $R3'$, since the string similarity value is 0.79 (right figure in

Fig. 3). $R5'$ in PST_b corresponds to $R5$, since the string similarity is 1. The same applies to $R6$ and $R9'$. There are no correspondence links for $R4$ and $R7$. Finally, the algorithm establishes correspondence links among model elements. In our example, correspondence links from $X1, X2, X3, X4, GB, CR$ and End in PST_a to the model elements with the same name in PST_b are created.

Figure 4 shows the complete set of correspondence links based on attribute matching, also indicating their model element cardinality. To avoid clutter, the links among model elements with the same name are not shown.

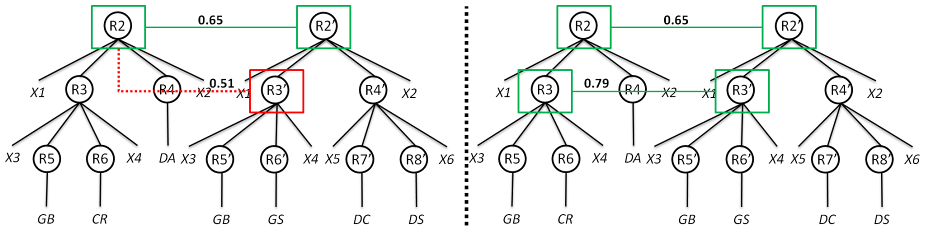


Fig. 3. Attribute matching phase step by step for $R2$ and $R3$

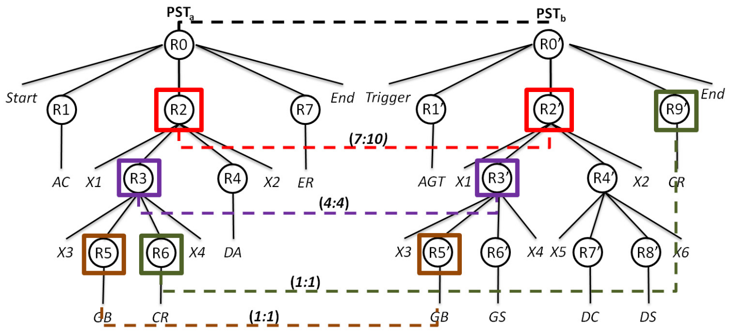


Fig. 4. Correspondence Links for the Attribute Matching Phase

3.3 Structure Matching

The second phase of the algorithm aims to match nodes that have not been matched in the first phase due to their different content. It does so by considering the location of the unmatched nodes in the PSTs and the correspondence links established so far. For example, consider regions $R4$ and $R4'$ in Fig. 4. Although they are dissimilar, it is likely, given the correspondence links so far, that they should be linked. The task of this procedure is to find such pairs of nodes and link them. The rule for finding node pairs to link is as follows. Let $\mathrm{n_a}$ and $\mathrm{n_b}$ be a pair of unmatched nodes. If the parents of $\mathrm{n_a}$ and $\mathrm{n_b}$ are linked, and if at least one sibling (the left or right one) of n$_a$ and n$_b$ are linked, n$_a$ and n$_b$ should be linked, too. If none of the siblings are linked (possibly because they do not exist), we will also link the nodes if both n$_a$ and n$_b$ are the last or first node in the child list. According to these rules, the aforementioned regions $R4$ and $R4'$ should be matched, as their parents ($R2$ and $R2'$) and their left and right siblings ($R3$ with $R3'$ and $X2$ with $X2$) match. The same happens with $R1$ and $R1'$ since $R0$ matches with $R0'$ and $R2$ with $R2'$. This newly created correspondence link allows us to link the *Start* and *Trigger* events, too. All correspondence links established by both phases of the algorithm are shown in Fig. 5. As previously, correspondence links between model elements with the same name are not shown.

Complexity. Assume $n = max(|PST_a|, |PST_b|)$, where $|PST|$ is the number of regions. The cost of comparing the attributes of two regions is denoted by c and the cost of checking their structure similarity is s. The matching of all regions is in $O(n^2(c + s))$, that is, $O(n^2)$, since the algorithm compares each possible region pair.

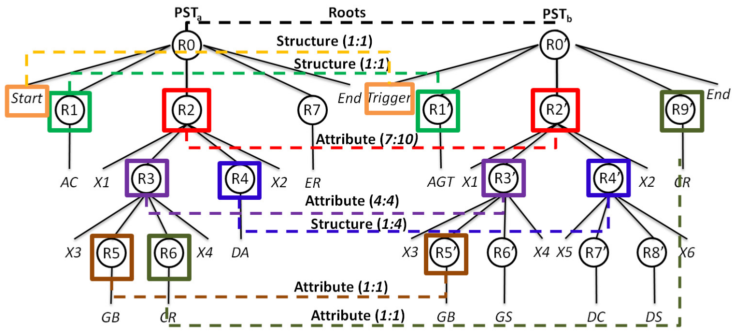


Fig. 5. Correspondence Links from Both Phases

4 Evaluation

We are interested in knowing the *precision* and *recall* of the presented algorithm when establishing correspondences among pairs of real-world process models across different levels of abstraction. Precision tells us whether the recovered correspondence links are correct and recall tells how large of a portion of the links the algorithm can recover. The following subsections present the methodology we have followed and the results.

4.1 Methodology

Objective. We want to evaluate the precision and recall of our algorithm. We define these measures for model correspondences (sets of correspondence links) between the PSTs. We refer to a model correspondence established by the domain experts as a *reference correspondence* (RC) and to a model correspondence established by our algorithm as a *computed correspondence* (CC). Given these sets, precision (P) and recall (R) are defined, respectively, as $P = \frac{|RC \bigcap CC|}{|CC|}$ and $R = \frac{|RC \bigcap CC|}{|RC|}$.

Subject Data. We used business process models taken from the Bank of Northeast of Brazil (BNB), a major financial institution in Brazil that is controlled by the federal government and oriented towards regional development. BNB has been using Business Process Modeling since 2007 in a development process based on the *Rational Unified Process.* The development process entails iterative and multi-staged model refinement, resulting in three types of process models (from higher to lower level of abstraction): business specifications, technical specifications, and executable processes. We had access to several projects developed as a result of this process and used them for evaluating the algorithm.

Table 1. BPM Projects

		Number of Models		
Project	Domain	Business	Technical	Implementation
P1	Customer Registration	2	2	2
P2	Credit Backoffice	6	6	6
P3	Credit Risk Assessment	2	2	2
P4	Procurement	3	3	3

We obtained four real BPM projects, containing 39 models in total. Table 1 shows, for each project, the number of models defined in each stage. Our target is to determine the correspondences between each corresponding pair of business and technical specifications and between the latter and executable implementations. Table 2 gives the total number of model elements for each level of abstraction.

Reference Correspondences. As reference correspondences, we use the correspondence links established manually by the domain experts (the bank's employees) who created and maintain the models. The reference correspondences in one of the projects was already established for auditing and regulatory compliance purposes, and reused here. The correspondences for the other projects were established as part of this research.

Table 2. Model Sizes

		Total Numbers		
		Tasks	Gateways	Events
P1	Business Spec.	59	38	25
	Technical Spec.	78	46	36
	Implementation	123	56	43
P2	Business Spec.	47	46	18
	Technical Spec.	95	48	23
	Implementation	107	52	31
P3	Business Spec.	17	8	6
	Technical Spec.	19	10	8
	Implementation	22	6	9
P4	Business Spec.	13	10	11
	Technical Spec.	18	12	15
	Implementation	25	14	17

Algorithm Implementation. We have implemented the algorithm in Java as an Eclipse feature, on top of the SOA Tools Platform BPMN Modeler [13]. Since the original models from BNB were created using IBM's WebSphere Process Modeler, we needed to recreate them to run our tool.

4.2 Results

Table 3 shows the results of our evaluation. We matched pairs of models at different levels of abstraction from each project. Concretely, we compared business and technical models, and the latter and IT implementation models. Column "Pair Type" indicates the type of models compared in each row. Column "Corresp - RC" gives the total number of correspondence links identified by the domain experts. Column "Corresp Type" shows the numbers obtained in each phase of the algorithm."Total" represents the net result of the two phases. Notice that the correspondence links do not overlap between the phases. Column "Correct" specifies the number and the cardinality of correspondence links that our algorithm was able to identify, in each phase, from those in the reference correspondence, including their cardinalities. Columns "FP" and "FN" give the number of false positives and negatives, respectively. False positives are those correspondence links that our algorithm finds but do not belong to the set of reference correspondence links. False negatives are those correspondence links included in the reference correspondence that our algorithm is unable to detect. In each phase, "FP" and "FN" are computed with respect to the complete reference. Finally, "Prec" gives precision, followed by column "Recall".

If we consider the correspondence links all together—as if they had been extracted from only one pair of models—we have 622 reference links found manually by the domain experts. Out of these 622, our algorithm was able to correctly identify 438, with 32 false positives and 184 false negatives, yielding overall recall of 70% and precision of 93% Among the reference links, 117 had cardinality type

Table 3. Correspondences among Models across Different Abstraction Levels. B: Business; T: Technical; IT: Information Technology; Corresp - RC: Reference Correspondence; FP: False Positives; FN: False Negatives; Prec: Precision.

Project	Pair Type	Corresp - RC	Corresp Type	Correct (*1:1*; *1:n*; *m:n*)	FP	FN	Prec	Recall
1	B–T	30	Attribute	16 (15;0;1)	0	14	100%	53%
			Structure	4 (1;2;1)	2	26	67%	13%
			Total	20 (16;2;2)	2	10	91%	67%
1	T–IT	42	Attribute	28 (26;0;2)	0	14	100%	67%
			Structure	3 (2;1;0)	2	39	60%	7%
			Total	31 (28;1;2)	2	11	94%	74%
2	B–T	138	Attribute	95 (90;0;5)	0	43	100%	69%
			Structure	8 (6;2;0)	4	130	67%	6%
			Total	103 (96;2;5)	4	35	96%	75%
2	T–IT	240	Attribute	136 (127;0;9)	0	104	100%	57%
			Structure	18 (10;5;3)	12	222	60%	8%
			Total	154 (137;5;12)	12	86	93%	64%
3	B–T	32	Attribute	22 (21;0;1)	0	10	100%	69%
			Structure	4 (4;0;0)	2	28	67%	13%
			Total	26 (25;0;1)	2	6	93%	81%
3	T–IT	44	Attribute	32 (32;0;0)	0	12	100%	72%
			Structure	2 (2;0;0)	5	42	29%	5%
			Total	34 (34;0;0)	5	10	87%	77%
4	B–T	42	Attribute	24 (23;0;1)	0	18	100%	57%
			Structure	6 (3;3;0)	3	36	67%	14%
			Total	30 (26;3;1)	3	12	91%	71%
4	T–IT	54	Attribute	36 (36;0;0)	0	18	100%	67%
			Structure	4 (2;1;1)	2	50	67%	7%
			Total	40 (38;1;1)	2	14	95%	74%

1:n and 89 had the cardinality type *m:n*. From these, the algorithm identified correctly 14 (12%) and 24 (27%), respectively.

The overall precision ranges between 87%-96%. None of the false positives is obtained in the attribute matching phase. This is very positive since a large portion of the reference correspondence links is recovered in this phase. The number of false positives in the structure matching phase is quite large compared to the number of reference correspondence links of purely structural nature. This would be a serious problem in a situation where models have many such purely structural correspondences. We identified two causes for having so many false positives. The principal cause is the presence of non-hierarchical refinement patterns [1]. For example, in one B-T pair of the Project 2, there is an activity in the business specification that corresponds to 3 activities in the technical specification. Each of the 3 activities belongs to a different region in the technical specification. The algorithm cannot identify such kind of correspondence; the second phase matched the region containing the business activity to an incorrect region in the technical specification. Another cause is matching nodes that are the last or first node in the child list. Although this is reasonable in many cases, it also leads to incorrect matches. For example, this rule produced a false positive in one T-IT pair of the Project 3. Unfortunately, without extra information (e.g., IDs or annotations) it is likely not possible to decide whether or not to match the regions in many of such cases.

The relatively high number of false negatives —20%-40%—is caused mainly also by the presence of non-hierarchical refinements, which occurred in all the projects. We believe that these numbers can be reduced by applying a pattern matching technique for describing and finding instances of well-known or organization-specific non-hierarchical refinements patterns, which we leave for future work.

4.3 Threats to Validity

This section summarizes the potential threats that may impact the internal and external validity [4] of the empirical results.

Threats to External Validity. A potential threat to external validity is that the models used in the evaluation may not be representative of those occurring in other realistic settings. While the models used here come from real-world projects, the algorithm should be tested additionally on models from other organizations and domains.

Threats to Internal Validity. The main threat is the re-modeling of the BNB's business process models to be processed by our tool. BNB applies IBM tools that use an extension of BPMN. Some features of the BNB models that are not covered in BPMN had to be omitted during the translation. This threat was minimized by checking with the domain experts that the BMPN models obtained after the simplification were largely equivalent to the original models.

5 Related Work

Matching of models is a standard topic in MDD. For example, UMLDiff is a prominent approach for matching UML models [19]. However, effective matching requires heuristics that are usually notation and application specific. Our work focuses on finding such heuristics for matching business process models across levels of abstraction. Discovery of effective heuristics usually requires studying the differences among such models. In this context, Dijkman [2] presents a classification of frequently occurring differences between similar business processes in general. Our previous study [1] provided an in-depth analysis of such differences between models targeting different levels of abstraction.

As in our approach, the work by Dijkman et al. [3] aims to realize business process models alignment based on lexical matching (similar to our attribute matching) and structural matching. They report recall of 60% and precision of 89% for their approach. However, their algorithm only captures 1:1 correspondences between model elements. Our algorithm also identifies correspondences between SESE regions, which is necessary for matching models at different levels of abstraction.

Weidlich et al. present ICOP in [17], a framework based on matchers to identify correspondences between process models. They represent the models using

Refined Process Structure Trees (RPSTs) [14,12] rather than PSTs. In RPSTs, regions can have more than one entry and more than one exit. The approach by Weidlich et al. deals both with 1:1 and 1:n matches. Our approach additionally relates regions to regions, which are examples of m:n matches. They report recall of 60% and precision of 80%.

Ehrig et al. [5] propose a set of similarity measures for process models, for example, in order to discover existing related process models in repositories. However, the approach does not establish fine grained correspondence links like in our approach. The authors do not discuss recall and precision of the approach.

Several related works deal with comparing process models (e.g., [7,8]), checking their consistency (e.g., [16]), and their synchronization (e.g., [18]). All these works assume that model correspondences have been previously established.

Table 4 summarizes our contribution in the light of the related works.

Table 4. Related BPM Matching Approaches. + : Feature Provided; – : Feature not Provided; NA : Not Available.

	Approach			
Feature	Weidlich et al. [17]	Dijkman et al. [3]	Ehrig et al. [5]	Our Approach
Match Activity Attributes	+	+	+	+
Match Model Structure	+	+	–	+
Match Activity-Activity (1:1)	+	+	+	+
Match Activity-SESE (1:n)	+	–	–	+
Match SESE-SESE (m:n)	–	–	–	+
Do not Require Model Element IDs	+	+	+	+
Support Activity Inserts and Deletes	+	+	+	+
Support Activity Moves	+	+	–	+
Support Activity Renaming	+	+	–	+
Support Activity Copies	+	+	–	–
Overall Precision	80%	89%	NA	93%
Overall Recall	60%	60%	NA	70%

6 Conclusions

We have presented an algorithm to automatically detect correspondences between BMPN process models across levels of abstraction. The algorithm operates over the PSTs of the input models in two phases. The first phase matches the PST nodes using region and model element matching criteria adapted from previous work on matching ASTs. The second phase establishes additional correspondences based on the position of the nodes in the PSTs.

We evaluated our algorithm on 26 pairs of business-technical and technical-IT level models from four real BPM projects, achieving overall precision of 93% and recall of 70%. Given the substantial recall and the high precision, the algorithm helps automating significant part of the correspondence recovery for such models.

The evaluation revealed that the algorithm is not able to detect certain kinds of complex correspondences. We believe that this limitation could be addressed in future work by extending the algorithm with an additional phase based on general and project-specific refinement patterns.

Acknowledgment. This work was partially supported by an IBM PhD CAS Fellow Scholarship and by Spanish Research Projects TIN2008-03107 and TIN2011-23795. We would like to thank the Bank of the Northeast of Brazil (Banco do Nordeste – BNB) for providing the case study as well as valuable requirements and feedback on the design of the algorithm.

References

1. Castelo Branco, M., Xiong, Y., Czarnecki, K., Küster, J., Völzer, H.: An Empirical Study on Consistency Management of Business and IT Process Models. Technical Report GSDLAB-TR 2012-03-22, Generative Software Development Laboratory, University of Waterloo, Waterloo (2012), `http://gsd.uwaterloo.ca/reportstudybpm`
2. Dijkman, R.: A Classification of Differences between Similar Business Processes. In: EDOC 2007, pp. 37–47. IEEE Computer Society, Washington, DC (2007)
3. Dijkman, R., Dumas, M., Garcia-Banuelos, L., Kaarik, R.: Aligning Business Process Models. In: EDOC 2009, pp. 45–53. IEEE (September 2009)
4. Easterbrook, S.M., Singer, J., Storey, M., Damian, D.: Selecting Empirical Methods for Software Engineering Research. In: Guide to Advanced Empirical Software Engineering, pp. 285–311. Springer (2007)
5. Ehrig, M., Koschmider, A., Oberweis, A.: Measuring similarity between semantic business process models. In: APCCM 2007, pp. 71–80. Australian Computer Society, Inc., Darlinghurst (2007)
6. Fluri, B., Wursch, M., Pinzger, M., Gall, H.: Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction. IEEE Transactions on Software Engineering 33(11), 725–743 (2007)
7. Gerth, C., Luckey, M., Küster, J.M., Engels, G.: Detection of Semantically Equivalent Fragments for Business Process Model Change Management. In: SCC 2010, pp. 57–64. IEEE Computer Society, Washington, DC (2010)
8. Gerth, C., Luckey, M., Küster, J.M., Engels, G.: Detection of Semantically Equivalent Fragments for Business Process Model Change Management. Tech. Rep. IBM Research Report RZ 3767, IBM Research, Zurich, Switzerland (2010), `http://www.cs.uni-paderborn.de/uploads/tx_sibibtex/rz3767.pdf`
9. Johnson, R., Pearson, D., Pingali, K.: The Program Structure Tree: Computing Control Regions in Linear Time. In: SIGPLAN Conference on Programming Language Design and Implementation (1994)
10. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions and reversals. Soviet Physics Doklady 10(8), 707–710 (1966)
11. Object Management Group: Business Process Model and Notation (BPMN) Version 2.0, `http://www.omg.org/spec/BPMN/2.0/`
12. Polyvyanyy, A., Vanhatalo, J., Völzer, H.: Simplified computation and generalization of the refined process structure tree. In: Proceedings of the 7th International Conference on Web Services and Formal Methods, WS-FM 2010, pp. 25–41. Springer, Heidelberg (2011)
13. SOA Tools Platform: Eclipse BPMN Modeler, `http://eclipse.org/projects/project.php?id=soa.bpmnmodeler`
14. Vanhatalo, J., Völzer, H., Koehler, J.: The Refined Process Structure Tree. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 100–115. Springer, Heidelberg (2008)

15. Vanhatalo, J., Völzer, H., Leymann, F.: Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 43–55. Springer, Heidelberg (2007)
16. Weidlich, M., Dijkman, R., Weske, M.: Deciding Behaviour Compatibility of Complex Correspondences between Process Models. In: Hull, R., Mendling, J., Tai, S. (eds.) BPM 2010. LNCS, vol. 6336, pp. 78–94. Springer, Heidelberg (2010)
17. Weidlich, M., Dijkman, R.M., Mendling, J.: The ICoP Framework: Identification of Correspondences between Process Models. In: Pernici, B. (ed.) CAiSE 2010. LNCS, vol. 6051, pp. 483–498. Springer, Heidelberg (2010)
18. Weidmann, M., Alvi, M., Koetter, F., Leymann, F., Renner, T., Schumm, D.: Business Process Change Management based on Process Model Synchronization of Multiple Abstraction Levels. In: Proceedings of SOCA 2011. IEEE Computer Society (2011)
19. Xing, Z., Stroulia, E.: Umldiff: an algorithm for object-oriented design differencing. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE 2005, pp. 54–65. ACM, New York (2005), http://doi.acm.org/10.1145/1101908.1101919

Experiences of Applying UML/MARTE on Three Industrial Projects

Muhammad Zohaib Iqbal[1,2], Shaukat Ali[1], Tao Yue[1], and Lionel Briand[1,3]

[1] Certus Center for V & V, Simula Research Laboratory, P.O. Box 134, Lysaker, Norway
[2] Department of Informatics, University of Oslo, Norway
[3] SnT Center, University of Luxembourg, Luxembourg
{zohaib,shaukat,tao}@simula.no,
lionel.briand@uni.lu

Abstract. MARTE (Modeling and Analysis of Real-Time and Embedded Systems) is a UML profile, which has been developed to model concepts specific to Real-Time and Embedded Systems (RTES). In previous years, we have applied UML/MARTE to three distinct industrial problems in various industry sectors: architecture modeling and configuration of large-scale and highly configurable integrated control systems, model-based robustness testing of communication-intensive systems, and model-based environment simulator generation of large-scale RTES for testing. In this paper, we report on our experiences of solving these problems by applying UML/MARTE on four industrial case studies. Based on our common experiences, we derive a framework to help practitioners for future applications of UML/MARTE. The framework provides a set of detailed guidelines on how to apply MARTE in industrial contexts and will help reduce the gap between the modeling standards and industrial needs.

Keywords: UML, MARTE, Real-time Embedded Systems, Architecture Modeling, Model-based Testing.

1 Introduction

Model Based Engineering (MBE) consists in using models as the primary artifacts in various development phases of software systems, including, for example, configuration and software testing. The Unified Modeling Language (UML) [1] and its extensions (via its profiling mechanism) are the most widely used modeling notations for software systems in diverse domains.

Real-time embedded systems (RTES) are widely used in many different domains, as for example from integrated control systems to consumer electronics. Already 98% of computing devices are embedded in nature and it is estimated that, by the year 2020, there will be over 40 billion embedded computing devices worldwide [2]. Modeling for such systems requires constructs that deal with characteristics specific to RTES (such as resource modeling, timeliness, schedulability). The recent UML profile for Modeling and Analysis of Real-Time Embedded Systems (MARTE) [3] is an effort to address the growing modeling needs of RTES.

In software engineering, like any engineering discipline, the usefulness of a new concept must ultimately be evaluated by applying it in real-life scenarios. To successfully apply MBE in practice, selecting a modeling language is not sufficient; rather we need to provide a detailed methodology on how to use the selected notations, which is a piece of information usually missing from language specifications and varies from problem to problem. This paper reports the experiences of four such applications on industrial RTES and based on the experiences layouts the guidelines that can be used for future successful application of UML/MARTE for RTES.

There are very few works discussing the experiences of using UML/MARTE. Demathieu *et al* [4] discuss their experiences of applying UML and MARTE on an academic case study for software resource modeling, hardware resource modeling, and modeling for logical system decomposition. Briand *et al* [5] discuss their experiences of applying MBE to three industrial cases belonging to maritime and energy domains using UML and MARTE. The work focuses on providing guidelines to improve collaboration between industry and researchers. Yue *et al* [6] discuss their experience of conducting a systematic and industrial domain analysis and the feasibility of applying model-based product line engineering methods for architecture modeling and configuration of large-scale integrated control systems. Espinoza *et al* [7] evaluate MARTE after applying it to a project in the automobile domain. Middleton *et al* [8] present their experiences about applying UML and MARTE for stochastic modeling of two interactive applications.

Our work discusses experiences of applying UML/MARTE on four industrial RTES belonging to different domains. We report our experiences of solving three industrial problems over the span of four years. The first problem was about architecture modeling and configuration of large-scale and highly configurable integrated control systems for FMC [9] Subsea Production Systems. The second problem was of model-based robustness testing of a video conferencing system at Cisco Systems [10]. The third problem was of environment model-based testing for a marine seismic acquisition system at WesternGeco [11] and an automated bottle recycling system at Tomra [12]. Based on our common experiences in the projects, we derived a comprehensive framework to successfully use MARTE in future industrial applications. The framework, which is the first of its kind, aims at providing detailed guidelines and steps on how to apply and extend UML/MARTE in industrial contexts.

The rest of the paper is organized as follows. Section 2 provides the background, while Section 3 discusses the contexts, modeling solutions and key results for the four selected industrial problems. Section 4 discusses the proposed framework based on our experiences from these four cases. Finally, Section 5 concludes the paper.

2 Background

The MARTE profile was defined to provide a number of concepts that modelers can use to express relevant properties of RTES, for example related to performance and schedulability. MARTE is meant to replace the previously defined UML profile for Schedulability, Performance, and Time specification (SPT) [13]. At the highest level, MARTE contains three packages. The core package is MARTE Foundations that contains the sub-packages for modeling non-functional properties (NFP package), time properties (Time package), generic resource modeling of an execution platform for

RTES (GRM package), and resource allocation (Alloc package). The MARTE Foundations package contains the core elements that are reused by the other two packages of the profile: MARTE design model and RealTime&Embedded Analysing (RTEA). The MARTE design model package contains various sub-packages required for modeling the design of RTES. This includes the packages to support modeling of component-based RTES with the Generic Component Model package (GCM), high-level features for RTES with the High-Level Application Modeling package (HLAM), and for detailed modeling of software and hardware resources with the Detailed Resource Modeling package (DRM). The RTEA package contains further concepts related primarily to modeling for analysis. This includes the Generic Quantitative Analysis Modeling package (GQAM) which provides generic concepts for resource modeling. These concepts are further specialized by the Schedulability Analysis Modeling (SAM) package for modeling properties useful for Schedulability and the Performance Analysis Modeling package (PAM) for modeling properties useful for performance analysis.

3 Industrial Applications of UML/MARTE

This section discusses three UML/MARTE applications in different industrial contexts. For each of the three applications, we provide the case study description, the problem description, the modeling solution, the modeling tool, and the key results of the application. This information will subsequently be used to propose a framework meant to provide guidance to future users of UML/MARTE.

3.1 Architectural Modeling and Configuration with UML/MARTE

Case Study Description. Integrated Control Systems (ICSs) are heterogeneous systems-of-systems, where software and hardware components are integrated to control and monitor physical devices and processes, such as process plants or oil and gas production platforms. FMC Technologies, Inc is a leading global provider of technology solutions for the energy industry. FMC's Subsea Production Systems (SPSs) are large-scale, highly-hierarchical, and highly-configurable ICSs for managing exploitation of oil and gas production fields. One of its key technologies is subsea production systems, used to develop new energy reserves and for managing and improving producing fields. They are composed of hundreds of mechanical, hydraulic, and electrical components and configured software to support various field layouts ranging from single satellite wells to large multiple-well sites (more than 50 wells). The main components of the system are subsea control modules, which contain software, electronics, instrumentation, and hydraulics for safe and efficient operation of subsea tree valves, chokes, and downhole valves.

Problem Description. The research question of this project is to devise a product line architecture modeling methodology, including modeling notations, guidelines, and tool support, for the purpose of facilitating the systematic and automated product configuration of ICSs such as FMC's SPSs. The ultimate goal is to improve the overall quality and productivity of the product development lifecycle of ICSs. Specifically, selected/tailored modeling notations of such a methodology should have the following characteristics: (i) It should contain both hardware and software modeling notations that should be expressive enough to model the required hardware and software concepts; (ii) The relations between software and hardware components should be captured, such as the deployment of a software component to its hardware computing resources; (iii) The consistency between hardware and software components should be maintained in the context of supporting configuration; (iv) The variability modeling notation should enable automated configuration and configuration reuse. We have proposed such a produce line architecture modeling methodology, named SimPL [14], to facilitate automated configuration of families of ICSs.

Modeling Solution. In addition to satisfy the modeling requirements described above, there are a number of practical requirements that affect the selection of existing modeling languages: 1) the modeling notation should be easy to learn and apply for industrial partners; 2) the modeling notation should have available tool support. Therefore our modeling solution is based on UML/MARTE, with a minimum extension through the UML profiling extension mechanism.

To facilitate automated configuration, the modeling notation we proposed for modeling the product line architecture uses UML classes, properties, and relationships (i.e., generalization relationships, and several types of association relationships) resulting in base models of hardware and software. In the SimPL methodology we use the following four stereotypes from MARTE to create hardware models and to model software to hardware bindings/allocations. To distinguish between hardware and software classes, any class in the hardware sub-view should be stereotyped by one of the following four MARTE stereotypes: 1) «HwComputingResource» is used to distinguish those electrical hardware components on which software is deployed; 2) «HwDevice» is used to distinguish those hardware devices that are controlled by, or in general interact with, software; 3) «HwComponent» characterizes hardware classes representing hardware components that physically contain other devices and execution platforms; 4) «Assign» models the deployment, allocation, or binding of a structure (e.g., software class) in the software sub-view to a resource (e.g., a hardware component) in the hardware sub-view. UML templates and packages, along with six stereotypes from our newly proposed profile, named SimPL, are used to model the product line architecture.

Modeling Tool. IBM Rational Software Architect (RSA) [15] was used to model the architecture.

Key results. The resulting product-line model contained a total of five views and sub-views and is visualized using 17 class diagrams. The model contains a total of 71 classes, including 46 software classes, 24 classes belonging to the hardware sub-view, and a class representing the topmost element, FMCSystem.

The software sub-view contains configurable software classes related to the selected components of the FMC family, their attributes, their relationships, and supporting containment and taxonomic hierarchies. The hardware sub-view captures a subset of devices (i.e., only those devices that are controlled by software classes captured in the software sub-view), their attributes, and the supporting containment and taxonomic hierarchies. The result is a hardware sub-view with 24 hardware components and devices, including 11 computing resources. Two types of relationships between the software and hardware classes (i.e., allocation of software to hardware and software controlling hardware) are captured in the allocation view.

The variability view contains 22 configuration units, corresponding to 22 configurable classes in software and hardware sub-views. A total of 109 variability points are organized using these configuration units. In addition, a total of 34 dependencies stereotyped with the SimPL profile were created to complete the variability model. A total of 16 OCL constraints are captured in the variability view modeling the dependencies between variability points, mainly the dependencies between variability points introduced by software and those introduced by hardware.

3.2 Model-Based Robustness Testing with UML/MARTE

We applied UML/MARTE to support automated, model-based robustness testing of a core subsystem of a video conferencing system developed by Cisco Systems, Norway.

Case Study Description. Our case study is a commercial Video Conferencing System (VCS) called Saturn developed by Cisco Systems Inc, Norway. The core functionality of Saturn manages the sending and receiving of multimedia streams. Audio and video signals are sent through separate channels and there is also a possibility of transmitting presentations in parallel with audio and video. Presentations can be sent by only one conference participant at a time and all others receive it. In total, Saturn consists of 20 subsystems such as audio and video subsystems. Each subsystem can run in parallel to the subsystem implementing the core functionality dealing with establishing videoconferences.

Problem Description. Our case study is part of a project aiming at supporting automated, model-based robustness testing of Saturn. A system should be robust enough to handle the possible abnormal situations that can occur in its operating environment and invalid inputs. For example, Saturn should be robust against hostile environment conditions (regarding the network and other communicating VCSs), such as high percentage of packet loss and high percentage of corrupt packets. Saturn should not crash, halt, or restart in the presence of, for instance, a high percentage of packet loss. Furthermore, Saturn should continue to work in a degraded mode, such as continuing the videoconference with lower audio and video quality. In the worst case, Saturn should return to the most recent safe state instead of bluntly stopping execution. Such behavior is very important for a commercial VCS and must be tested systematically and automatically to be scalable.

Modeling Solution. Following, we discuss our modeling solution to support automated robustness testing.

To model the functional behavior, for each subsystem, we modeled a class diagram to capture APIs and state variables. In addition, we modeled one or more state machines to capture the behavior of each subsystem. Due to confidentiality restrictions, we do not provide details of the subsystems. However, on average each subsystem has five states and 11 transitions, with the biggest subsystem having 22 states and 63 transitions. It is important to note that, though the complexity of an individual subsystem may not look high in terms of number of states and transitions, all subsystems run in parallel to each other and therefore the spaces of system states and possible execution interleavings are very large. Saturn's implementation consists of more than three million lines of C code.

Table 1. Summary of features of MARTE and other profiles applied

Robustness Behavior	Stereotypes			Existing MARTE NFPs	Newly introduced NFPs
	NFP	GRM	RobustProfile		
Media Quality	2	1	19	19	2
Network Communication	4	1	13	21	3
Illegal Inputs	-	-	1	2	-

Robustness behavior is typically crosscutting many parts of the system functional model and, as a result, modeling such behavior directly within the functional models is not practical since it leads to many redundancies and hence results in large, cluttered models. To cope with this issue, we decided to adopt Aspect-Oriented Modeling (AOM) [16] and more specifically a UML profile for AOM called AspectSM [17]. With it, we model each aspect as a UML state machine with stereotypes (aspect state machine). The modeling of aspect state machines is systematically derived from a fault taxonomy [17] categorizing different types of faults (faults in the environment such as communication medium and media streams that lead to faulty situations in the environment). Each aspect state machine has a corresponding aspect class diagram modeling different properties of the environment using the MARTE profile, whose violations lead to faulty situations in the environment. More specifically, we used the NFPs package to model properties of the operating environment of Saturn.

Saturn's non-functional behaviors consist of five aspect class diagrams and five aspect state machines modeling various robustness behaviors. The largest aspect state machine specifying robustness behavior has three states and ten transitions, which would translate into 1604 transitions in standard UML state machines if AspectSM was not used.

Modeling Tool. IBM RSA was used for modeling class diagrams, UML state machines, aspect state machines, and defining the AspectSM profile.

Key Results. Table 1 summarizes the features of the MARTE profile and other profiles, which we used in conjunction with MARTE in our case study. The first column shows various robustness behaviors we modeled in this case study. The first one is related to modeling faulty situations in media, i.e., audio and video, the second behavior is about constraining parameters of events on transitions, which is used to generate test cases exercising the system robustness with illegal inputs, and the third robustness behavior models the behavior of a system in the presence of various network faults. Columns two and three show that we used stereotypes from MARTE NFP and GRM packages. For instance, to model network communication we used four stereotypes from the NFP package (e.g., *NfpType*), whereas we used one stereotype from the GRM package, *CommunicationMedia*. The fourth column shows the stereotypes from other profiles used in conjunction with MARTE. In our case study, we used stereotypes from RobustProfile [17], which allows modeling various properties of faults (e.g., severity) to assist in defining robustness test strategies. For example, for modeling media quality we used in total 19 stereotypes such as *AudioFault* and *VideoFault* from RobustProfile. The fifth column shows the number of existing NFPs we used that are already defined in MARTE for each of the robustness behaviors. For media quality, we used 19 existing NFPs, e.g., *NPF_Percentage*. The last column shows the number of new NFPs we defined in our case study. For instance, in case of media quality, we defined two new NFPs based on the existing NFPs defined in MARTE, e.g., *PacketLoss* in the case of modeling network communication.

3.3 Testing RTES Using UML/MARTE Environment Models

We applied our approach for model-based testing of RTES to two industrial case studies, involving WesternGeco AS and Tomra AS, both in Norway.

Case Study Description. The case study at WesternGeco is of a very large and complex control system for marine seismic acquisition. The system controls tens of thousands of sensors and actuators in its environment. The timing deadlines on the environment are in the order of hundreds of milliseconds. WesternGeco is a market leader in the field of such seismic systems. The system was developed using Java.

The second case study is an automated bottle-recycling machine developed by Tomra AS. The system under test (SUT) was an embedded device 'Sorter', which was responsible to sort the bottles into their appropriate destinations. The system communicated with a number of components to guide recycled items through the recycling machine to their appropriate destinations. It is possible to cascade multiple sorters with one another, which results in a complex recycling machine. The SUT was developed using C.

Both the RTES were running in environments that enforce time deadlines in the order of hundreds of milliseconds with acceptable jitters of a few milliseconds in response time.

Problem Description. RTES typically work in environments comprising large numbers of interacting components. The interactions with the environment can be bound by time constraints. Violating such time constraints, or violating them too often for soft real-time systems, can lead to serious failures leading to threats to human life or the environment. For effective testing of industrial scale RTES, systematic automated testing strategies that have high fault revealing power are essential. The system testing of RTES requires interactions with the actual environment. Since, the cost of testing in real conditions tends to be high, environment simulators are typically used for this purpose. For the industrial systems of WesternGeco and Tomra, we applied one such approach for black-box system level testing based on the environment models of the systems. These models were used to generate an environment simulator [18], test cases, and obtain test oracles [19]. For test case generation, we applied various testing strategies, including search-based testing using search-based testing [20], adaptive random testing [21], and a hybrid approach combining these two strategies [22].

Modeling Solution. The environment models were developed using our proposed UML & MARTE Real-time Embedded systems Modeling Profile (REMP) [23]. REMP provided extension to the standard UML class diagram and state machine notations and used the MARTE Time package and GQAM package for modeling timing details and non-deterministic events, respectively. One of the major aims while developing REMP was to keep it as simple as possible. We only used those notations and concepts from UML/MARTE that were essential to model the two industrial case studies. Even though the notation subset was minimal, the goal was to keep REMP generic and applicable to the testing of soft RTES belonging to various domains. This was the motivation to apply the methodology to two case studies that belonged to entirely different domains.

The structural details of a RTES environment were modeled as an environment domain model, which captures the information of various environment components, their properties, and their relationships. For the domain model, we used the UML class diagram notation and annotated class diagram elements with REMP. The behavioral details of the environment were modeled using the state machine notation annotated with REMP. Each environment component has one associated state machine. Such state machines contain information of the nominal behavior of the components, their robustness behavior (e.g., break down of a sensor), and "error states" that should never be reached (e.g., hazardous situations). If any of these error states is reached, then it implies a faulty RTES. Error states act as the oracle of the test cases, i.e., a test case is successful in triggering a fault in the RTES if an error state of the environment is reached during testing.

Modeling Tool. For initial interactive sessions with experts, we used a sketching tool to model the domain. Later on when we had sufficient details of the system, we used Enterprise Architect for modeling Tomra's case study (because that was the tool they already used) and IBM RSA for modeling WesternGeco. Later on due to various limitations of Enterprise Architect, we migrated the models to IBM RSA.

Key Results. For Tomra's case, we had a total of 55 environment components, out of which 43 have a corresponding state machine. For testing, we only focused on a subset of the SUT, for which we only use four of the environment components with a total of 23 states and 38 transitions. For the subset of environment models for WesternGeco's case, a total of three environment components have a state machine. In total for these components, we modeled 27 states and 46 transitions. In both cases, environment components have a large number of instances during test case execution.

From MARTE, we mostly used the concepts of *TimedEvent* and *TimedProcessing* from the Time Package. The MARTE *TimedEvent* concept is used to model timeout transitions, so that it is possible for the time events to explicitly specify a clock (if needed). Each environment component may have its own clock or multiple components may share the same clock for absolute timing. Clocks are modeled using the MARTE's concept of clocks. According to REMP, if no clock is specified, then by default the notion of time is considered to be according to the physical time. Specifying a threshold time for an action execution or for a component to remain in a state is done using the MARTE *TimedProcessing* concept. This is also a useful concept and can be used, for example, to model the behavior of an environment component when the RTES expects a response from it within a time threshold.

From the GQAM package of MARTE, we used the concept of *GaStep* to model non-determinism. Whenever a timeout transition is labeled with «*gaStep*» and a non-zero value for the *prob* property, this is interpreted as the probability of taking the transition over the time of the test case execution. This stereotype was used to model scenarios where the modeler wants to specify exact probabilities of an event occurrence. For non-determinism, REMP provides other stereotypes too that give more control to the testing engine to specify the probability of event occurrences.

In our methodology, we chose Java as the action language for writing actions. The decision to choose Java as the action language at the model level is due to the lack of tool support for the UML action language (ALF) [24] at the time our tool was developed. Testers of the SUT are also expected to be more familiar with Java (consistent with our experience of applying the approach in two industrial contexts), rather than with a newly approved, standard language. Moreover, ALF does not provide support for specifying time related actions (e.g., corresponding to the MARTE's concept of an *RTAction* to specify an atomic action). It was also not possible to specify time delays with ALF. Both these concepts were used repeatedly while modeling the environment of both industrial cases.

4 Framework for Applying UML/MARTE in Industry

In this section, we present a framework we devised by combining our experiences in applying UML/MARTE on the industrial problems described above. This framework can help practitioners in future application of UML/MARTE in industrial contexts. At a high level, the framework is presented as a UML activity diagram shown in Fig. 1. Following, we briefly discuss each of these activities.

4.1 Perform Domain Analysis (A1)

Each of our industrial applications started from performing a domain analysis. Domain analysis is defined as "*the process by which information used in developing software systems within the domain is identified, captured, and organized with the purpose of making it reusable (to create assets) when building new products*" [25]. Typically, the domain analysis results in a domain model [26] that captures domain concepts and the relationships among them. A domain model can be described using different notations, UML being a frequently used one. For all the three applications, we used the UML class diagram notation for domain modeling.

The objective of the domain analysis that we performed was different from what is typically presented in OOAD methods [27]. More specifically, our domain analysis is not the start of the software analysis phase but its usage depends on the problem at hand. For architectural modeling, the domain model was later used as a basis to derive the product line architecture modeling methodology, including a UML profile and modeling guidelines. For both the model-based testing projects, the domain analyses resulted in the definition of either environment or system static structure models (as class diagrams), which were used, later on together with state machines, to facilitate automated test-case generation.

To derive the domain models, we followed an iterative process during which we had multiple sessions with our industry partners. In some cases, we initially used sketching tools and simple drawings on white boards for ease of understanding. We started by just capturing the concepts first and later introduced associations and attributes in the domain models. Last, we also added OCL constraints on the domain model concepts. Detailed discussions on how the domain analysis was performed in each of the applications are provided in [6] for architectural modeling, in [17] for robustness testing, and in [23] for model-based testing of RTES.

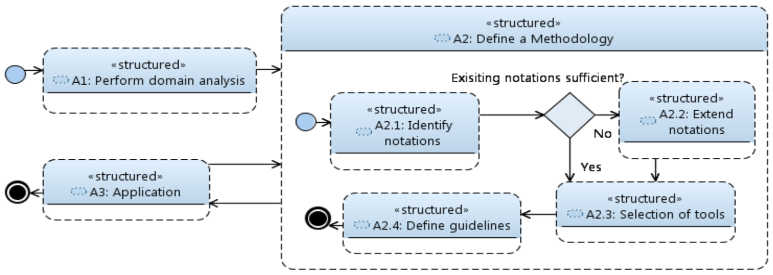


Fig. 1. Framework for UML/MARTE applied to industrial applications

In all the three cases, the domain analysis was useful in the following ways: (i) It helped us in understanding and specifying (as a domain model) the complexities of large-scale systems having characteristics of multiple disciplines (e.g., electrical, mechanical, and software) and involving multiple stakeholders; (ii) It was instrumental in understanding the needs of industry partners and served as a communication medium with them; (iii) It formed a basis for other activities that we carried out at later stages of the projects, such as defining the modeling methodology and identifying the language and notations for the modeling solution.

4.2 Define a Modeling Methodology (A2)

After performing the domain analysis, we defined a specific modeling methodology to tackle each problem, keeping in mind the requirements of the domain. To apply UML/MARTE in practice, just identifying a set of notations is not sufficient. We need to define a proper process and guidelines, select proper modeling tools, and train the industry partners regarding all these aspects. Following, we discuss the various sub-activities of defining a methodology.

Identify Notations (A2.1). The first activity for each of the applications was to identify the modeling notations. In all our industrial applications, we carefully selected a subset of UML and MARTE for modeling. The reasons for using UML are as follows: (i) it is a modeling standard; (ii) it has industrial strength tool support ranging from open source (e.g., Papyrus) to commercial (e.g., IBM RSA); (iii) it has sufficient training material available to help train industry partners; (iv) it provides a rich set of notations to model a system from different perspectives; (v) it is extensible for various application domains. Though MARTE is a relatively new profile, we have observed significant progress in tool support and training material available over the last couple of years. Plus it has a rich set of concepts, which can be selected and used for various modeling purposes in the context of real-time, embedded, and concurrent systems.

Despite the above-mentioned advantages, UML is still a challenge to apply in industrial settings without clear objectives and a well-defined methodology. UML is a general purpose, standard modeling language that is meant to cater for different application domain and problems, and is as a result quite large. The entire language is not meant to be used to solve a particular problem in a particular domain. Therefore one of the key requirements to make UML successful in industry is to select a proper subset of the language matching the needs. In our projects, we systematically aimed to identify such a minimal subset. Fig. 2 shows the packages of UML that were used for our applications. We used UML class diagrams for modeling the domains for all the industrial case studies. Other notations were selected based on individual needs of the target industrial problem and domain. For architectural modeling we used UML package and class diagram, and for both model-based testing applications, we used UML state machines to model system behavior. In total, we only used four out of fourteen UML diagrams (including the UML profile diagram that we used to create profiles as part of activity A2.2).

MARTE is a comprehensive UML profile covering different aspects for modeling RTES (Section 2). Similar to UML, the set of concepts provided by MARTE are fairly large to cater to a wide variety of analysis needs and it is also important to clearly identify the required subset of MARTE for a specific problem and domain. Fig. 3 shows the six MARTE packages we used (highlighted in grey), a selected subset of the concepts which were used to model our four industrial case studies. In our experience, using UML/MARTE showed to be an adequate combination considering our industrial application domains.

Extend Notations (A2.2). After we identified the subset of UML and MARTE, the next step was to find out whether the identified notations were sufficient to address our problems. Various steps that we performed in this activity are summarized as an activity diagram in Fig. 4. First we evaluated whether the identified MARTE subset was sufficient. If this was not the case, we tried to extend MARTE using the defined constructs (e.g., by adding a new NFP). When required, we further defined guidelines on how to extend MARTE (for example, see [17]) in the future. We then evaluated whether the identified subsets of UML, MARTE, and its extensions were sufficient for our modeling purposes. If this was not the case, we extended UML by creating UML profiles. One of the important decisions was to decide whether to go for a profile or a domain specific language (DSL). In all our cases, we decided to opt for UML profiles over DSL since, in our applications as in many others, minimizing extensions to UML is expected to ease practical adoption and technology transfer. In [28], two main approaches for profile creation are discussed. The first approach directly implements a profile by defining key concepts of a target domain, such as what was done to define SysML [29]. The second approach first creates a conceptual model outlining the key concepts of a target domain followed by creating a profile for the identified concepts, such as what was done to define SPT [13] and MARTE. We used the second approach to define profiles in our context, since it is more systematic as it clearly separates the profile creation process into two distinct stages.

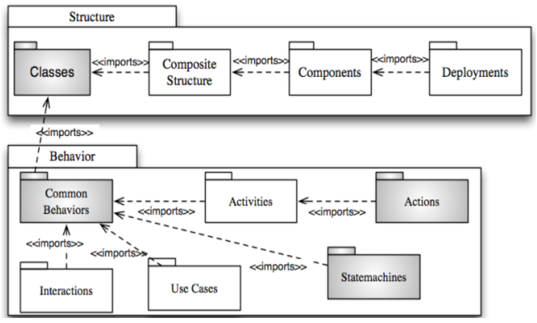


Fig. 2. UML packages used in industrial case studies (highlighted in grey)

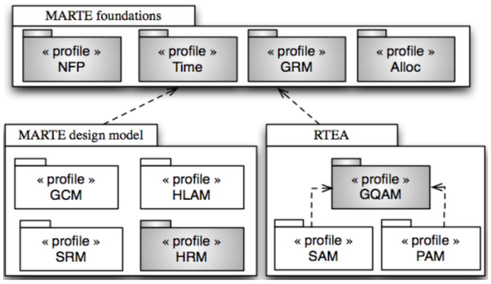


Fig. 3. MARTE packages used in industrial case studies (highlighted in grey)

We found the MARTE NFP package and the extension mechanism sufficient for our industrial application of model-based robustness testing. The NFP package provides different data types such as *NFP_Percentage* and *NFP_DataTxRate*, which are helpful to model properties of the environment, for instance jitter and packet loss in networks. When the built-in data types of MARTE are not sufficient, the open modeling framework of MARTE can be used to define new NFP types by either extending the existing NFPs or by defining completely new NFPs. For example, we extended MARTE's NFPs and define several properties of the environment when modeling echo in audio streams and synchronization mismatch between audio and video streams coming to a video conferencing system. From our experience of using MARTE, in addition to the advantages of using a standard, we can conclude that the MARTE profile and its open modeling framework were sufficient to model relevant properties of the Saturn operating environment (Section 3.2). However, for our specific problem of robustness testing, we defined a UML profile called RobustProfile [17] to model faults and their properties. The profile supports the modeling of recovery mechanisms when a fault has occurred and the modeling of states that a system can transition to when it has recovered. Since these features were not part of MARTE, a profile was required.

For architecture modeling, we proposed the use of 6 new concepts as stereotypes to extend UML. For model-based robustness testing, we proposed 30 new stereotypes to extend UML and MARTE, and for the environment model-based testing profile, we proposed 8 new stereotypes to extend UML concepts. Overall, we can see that a limited number of stereotypes were required to extend UML for all the three projects. For robustness testing, most of the new stereotypes were based on a fault model and were extending MARTE NFPs.

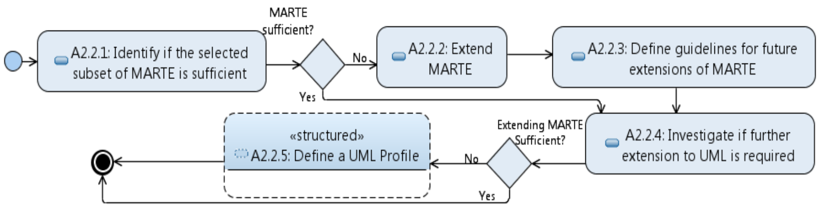

MARTE sufficient?
A2.2.1: Identify if the selected subset of MARTE is sufficient
No
Yes
A2.2.2: Extend MARTE
A2.2.3: Define guidelines for future extensions of MARTE
A2.2.4: Investigate if further extension to UML is required
Extending MARTE Sufficient?
No
Yes
«structured» A2.2.5: Define a UML Profile

Fig. 4. Sub-activities under the activity A2.2 Extend Notations

Tool Selection (A2.3). An important consideration for the practical adoption of our proposed methodologies in industrial settings is the selection of an adequate modeling tool. This is important since the models developed are meant to support automation (e.g., software test automation). The modeling tool should provide support to export the models in a standard format which can later be processed by other MDE tools (e.g., for model transformations and OCL parsers). According to the MARTE official website [30], the MARTE profile is available in four tools: IBM RSA [15], IBM Rhapsody [31], Papyrus UML [32], and MagicDraw [33].

Among these modeling tools, only IBM RSA and Papyrus UML are EMF-based and hence can be used with other EMF based MBE tools (e.g., Kermeta for model transformations). For Papyrus UML, we faced serious usability problems when modeling state machines, since most of the interface of the tool is based on the assumption that the modeler is aware of the underlying UML metamodel. IBM RSA comes with a high price tag to be used in small to medium sized companies. Even IBM RSA has usability issues, for example, it is not possible to directly link action on a modeling element, such as sending of a signal, in the action code written as part of effects in the state machines. Similarly, the MARTE profile is only compatible with RSA version 7.0 and if used with later versions, it does not support the Value Specification Language (VSL) editor. Due to this reason and that a complete parser of VSL was not available at the time we worked on the industrial projects, we used OCL to specify values for NFPs and other MARTE types.

For one of the projects, we also worked with Enterprise Architect [34]. Though Enterprise Architect is cheap and affordable for smaller companies, migrating its models to a form compatible to EMF-based tools is not trivial. Initially for the domain model we also used a sketching tool, which was easier to use for industry partners, because it did not enforce any constraint on the modeling.

Overall, we found IBM RSA as the most viable modeling tool in terms of usability and its interoperability with third party MBE tools.

Define Guidelines (A2.4). The next step was to define modeling guidelines for each of the methodologies. As discussed earlier, only specifying a set of notations is not sufficient and we need a proper methodology to help modelers determine what to model, in which order, and at what level of detail. The guidelines are not generic and are, to some extent, specific to each domain and application. For example, for the environment model-based testing approach, we defined guidelines to help modelers in identifying test-relevant environment concepts and their relationships in the context of embedded systems [23]. According to our experience, such guidelines are crucial for modelers to correctly and effectively apply our modeling notations.

4.3 Application of Methodology

Once the methodology was defined, numerous training sessions took place, which ranged from acquiring basic UML modeling skills to more advanced methodology specific training. Training was conducted in an interactive manner, where the attendees were given exercises based on their own domain and systems. This last point is very important as people more easily understand and adopt technologies that have shown to apply to their environment.

Training must be complemented by workshops where we model the solution to a representative (sub)problem with them, thus reducing the initial learning curve with respect to the modeling tool and notations. Later, when the first modeling activities are undertaken, mentoring is also required, at least in the initial stages, until a certain level of comfort is attained. A natural tendency is for people to revert to previous practices when faced with a seemingly intractable problem.

4.4 Summary and Discussion

For the three industrial projects, we used the UML class, package and state machine diagrams for modeling the different aspects of software systems. From MARTE, we used concepts from the MARTE Time, NFP, GQAM, Alloc, GRM and HRM packages. Over the years, a number of researchers and industry practitioners have raised the issue that UML is too large [35] [36]. Recently, the same has been written about MARTE [7]. In our opinion and based on our practice, UML and MARTE are meant to provide an encompassing set of modeling notations catering diverse needs. To successfully apply these standards to industrial projects, we need a complete methodology that identifies the subset of UML and MARTE to be used to address specific problems in specific contexts and guidelines to help people apply such standards in a systematic and consistent manner.

A complete methodology based on UML/MARTE should be derived for a specific purpose, to address a particular problem in a particular domain. To do so, we found that a thorough domain analysis is an important step, which, as we discussed in Section 4.1, is a necessary basis not just for the analysis but also to make decisions during other activities. Depending on the complexity of the domain under analysis and the nature of the problem, the domain analysis activities and effort required vary significantly from case to case. The next steps are to carefully select a minimal subset of UML and MARTE notations and if needed, extend MARTE, for example by defining new NFPs, and extend UML by defining a profile. Though the selection of a modeling tool might seem to be a trivial process, in our experience, this can have large impact on adoption by the industry partners. If needed, the modeling tool should be customized based on the modeling notations selected, so that concepts of UML and MARTE that are not relevant are also not visible to the end user. Along with the notations, we found it an essential step to provide a set of modeling guidelines for the end user, which will help her to properly use these notations.

Integrating UML and MARTE can be challenging too, especially when it comes to bridging the semantic gap between the two. For example, when «HwComponent» was used on a class in a class diagram to represent a hardware component, the meaning of its association with another class not carrying any stereotype becomes ambiguous. This is because UML is typically used to model software. Without having any stereotype applied, a class by default implies that it is a software class. Then the association between the hardware component class and the software class should be given a specific meaning, like the deployment of the software to its hardware platform. In our cases, we address such semantic gaps in our modeling guidelines.

In our experience, there is limited action language support for MARTE concepts, such as time delays between actions and the concept of *RTAction* (e.g., required to model atomic actions). Even in the recently released Action Language for Foundational UML (ALF) [24], such concepts are not supported. We used Java as an action language, which provided the concepts of real-time actions that we required.

For model-based robustness testing of RTES, we defined a profile for modeling faults and their properties, recovery mechanisms, and faulty states. These are based on well-defined fault models in the literature and are applicable to RTES in general. These concepts can be a good addition as they align with the goals of the MARTE profile, though this requires further investigation.

5 Conclusion

Applying Model-based Engineering (MBE) notations and methodologies to real-life industrial problems is a challenging task and very few articles in the research literature report on such experiences. For successful MBE application, a comprehensive methodology for modeling should be adopted that is specific to the problem being solved and adequate for the application domain. This paper discusses our experiences of applying Unified Modeling Language (UML) and the UML profile for Modeling and Analysis of Real-Time Embedded Systems (MARTE) to solve three distinct industrial problems related to the use of real-time embedded systems (RTES) in four different industry sectors. The industrial problems that we tackled were related to architectural modeling and configuration, model-based robustness testing, and environment model-based testing of RTES. Based on these experiences, we derived a framework to guide practitioners in their application of UML/MARTE in industrial contexts. This will help practitioners bridge the gap between modeling standards and the modeling needs of industrial RTES.

Acknowledgements. We are thankful to all our industrial partners for their support. M. Iqbal was funded by Norwegian Research Council as part of the ITEA 2 VERDE project. L. Briand was supported by a FNR PEARL grant.

References

1. OMG: Unified Modeling Language Superstructure, Version 2.3 (2010), `http://www.omg.org/spec/UML/2.3/`
2. Artemis Joint Undertaking - The public private partnership for R & D Embedded Systems, `http://artemis-ju.eu/embedded_systems`
3. OMG: Modeling and Analysis of Real-time and Embedded systems (MARTE), Version 1.0 (2009), `http://www.omg.org/spec/MARTE/1.0/`
4. Demathieu, S., Thomas, A.F., Andre, A.C., Gerard, S., Terrier, F.: First Experiments Using the UML Profile for MARTE. In: Proceedings of the 2008 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing, pp. 50–57. IEEE (2008)
5. Briand, L., Falessi, D., Nejati, S., Sabetzadeh, M., Yue, T.: Research-Based Innovation: A Tale of Three Projects in Model-Driven Engineering. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, pp. 800–816. Springer, Heidelberg (2012)
6. Yue, T., Briand, L., Selic, B., Gan, Q.: Experiences with Model-based Product Line Engineering for Developing a Family of Integrated Control Systems: an Industrial Case Study. Simula Research Laboratory, Technical Report (2012-06) (2012)
7. Espinoza, H., Richter, K., Gérard, S.: Evaluating MARTE in an Industry-Driven Environment: TIMMO's Challenges for AUTOSAR Timing Modeling. In: Proceedings of Design Automation and Test in Europe (DATE), MARTE (2008)
8. Middleton, S.E., Servin, A., Zlatev, Z., Nasser, B., Papay, J., Boniface, M.: Experiences using the UML profile for MARTE to stochastically model post-production interactive applications. In: eChallenges 2010, pp. 1–8 (2010)
9. FMC Technologies, `http://www.fmctechnologies.com`
10. Cisco Inc., `http://www.cisco.com`
11. WesternGeco, `http://www.slb.com/services/westerngeco.aspx`
12. Tomra AS, `http://www.tomra.no`
13. UML Profile for Schedulability, Performance and Time (SPT), `http://www.omg.org/technology/documents/profile_catalog.htm`
14. Behjati, R., Yue, T., Briand, L., Selic, B.: SimPL: A Product-Line Modeling Methodology for Families of Integrated Control Systems Technical Report 2011-14 (ver.2), Simula Research Laboratory (2012)
15. IBM RSA, `http://www.ibm.com/software/awdtools/architect/swarchitect/`
16. Yedduladoddi, R.: Aspect Oriented Software Development: An Approach to Composing UML Design Models. VDM Verlag, Dr. Müller (2009)
17. Ali, S., Briand, L.C., Hemmati, H.: Modeling Robustness Behavior Using Aspect-Oriented Modeling to Support Robustness Testing of Industrial Systems. Simula Research Laboratory, Technical Report (2010-03) (2010)
18. Iqbal, M.Z., Arcuri, A., Briand, L.: Code Generation from UML/MARTE/OCL Environment Models to Support Automated System Testing of Real-Time Embedded Software. Simula Research Laboratory, Technical Report (2011-04) (2011)
19. Arcuri, A., Iqbal, M.Z., Briand, L.: Black-Box System Testing of Real-Time Embedded Systems Using Random and Search-Based Testing. In: Petrenko, A., Simão, A., Maldonado, J.C. (eds.) ICTSS 2010. LNCS, vol. 6435, pp. 95–110. Springer, Heidelberg (2010)
20. Iqbal, M.Z., Arcuri, A., Briand, L.: Empirical Investigation of Search Algorithms for Environment Model-Based Testing of Real-Time Embedded Software. In: International Symposium on Software Testing and Analysis (ISSTA). ACM (2012)

21. Iqbal, M.Z., Arcuri, A., Briand, L.: Automated System Testing of Real-Time Embedded Systems Based on Environment Models. Simula Research Laboratory. Technical Report (2011-19) (2011)
22. Iqbal, M.Z., Arcuri, A., Briand, L.: Combining Search-based and Adaptive Random Testing Strategies for Environment Model-based Testing of Real-time Embedded Systems. In: Symposium on Search-based Software Engineering. Springer (2012)
23. Iqbal, M.Z., Arcuri, A., Briand, L.: Environment Modeling with UML/MARTE to Support Black-Box System Testing for Real-Time Embedded Systems: Methodology and Industrial Case Studies. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 286–300. Springer, Heidelberg (2010)
24. OMG: Concrete Syntax for UML Action Language (Action Language for Foundational UML - ALF), Version 1.0 - Beta 1 (2010), `http://www.omg.org/spec/ALF/`
25. America, P., Thiel, S., Ferber, S., Mergel, M.: Introduction to Domain Analysis (2001), `http://www.esi.es/esaps/public-pdf/CWD121-20-06-01.pdf`
26. Conceptual Model: `http://en.wikipedia.org/wiki/Conceptual_model_computer_science`
27. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. Prentice Hall PTR, Upper Saddle River (2001)
28. Lagarde, F., Espinoza, H., Terrier, F., André, C., Gérard, S.: Leveraging Patterns on Domain Models to Improve UML Profile Definition. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 116–130. Springer, Heidelberg (2008)
29. Weilkiens, T.: Systems Engineering with SysML/UML: Modeling, Analysis, Design. Tim Weilkiens (2008)
30. MARTE Tools, `http://www.omgmarte.org/node/31`
31. IBM Rational Rhapsody, `http://www.ibm.com/software/awdtools/rhapsody/`
32. Papyrus UML, `http://www.papyrusuml.org`
33. MagicDraw, `http://www.magicdraw.com/`
34. Enterprise Architect, `http://www.sparxsystems.com/`
35. Grossman, M., Aronson, J.E., McCarthy, R.V.: Does UML make the grade? Insights from the software development community. Information and Software Technology 47, 383–397 (2005)
36. Suess, J.G., Fritzson, P., Pop, A.: The Impreciseness of UML and Implications for ModelicaML. In: Proceedings of the 2nd International Workshop on Equation-Based Object-Oriented Languages and Tools (EOOLT 2008), pp. 17–26. Linköping University (2008)

Cost Estimation for Model-Driven Engineering

Sagar Sunkle and Vinay Kulkarni

Tata Research Development and Design Center
Tata Consultancy Services
54B, Industrial Estate, Hadapsar
Pune, 411013 India
{sagar.sunkle,vinay.vkulkarni}@tcs.com

Abstract. Cost estimation studies in model-driven engineering (MDE) are scarce; first, due to difficulty in quantifying qualitative characteristics of MDE that supposedly influence software development effort and second, due to the complexity of measuring varied artifacts that are generated and used in an end-to-end MDE toolset. A cost estimation approach is therefore needed that can incorporate characteristics of MDE that affect economies of scale and effort in application development with the size computation of various artifacts in MDE. We plan to use the constructive cost model (COCOMO) II to obtain baseline cost estimation of MDE applications. Our main contributions are a method to capture the qualitative characteristics of MDE in terms of cost drivers in COCOMO II and a method for computation of various artifacts generated by an MDE toolset. Our initial exploration of these ideas suggests that it is possible to automate cost estimation for MDE.

Keywords: Model-driven Engineering, Cost Estimation, COCOMO II.

1 Introduction

We began developing our model-driven engineering (MDE) toolset 15 years ago with a need to deliver a banking application on a short notice [1,2]. Since then we have successfully delivered 60+ large business-critical enterprise applications worldwide on a wide variety of technology platforms and differing domains using our MDE toolset [3,4]. Yet, even now, as we commence negotiations with prospective customers, we are asked if we can prove that our flavor of MDE or MDE in general is economically more beneficial to them than code-centric approaches.

The consideration of economic issues of MDE is particularly relevant now more than ever. A number of IT companies are using MDE toolsets of their own to deliver large applications spanning variety of domains [5]. MDE has been thought by many to provide a slew of advantages over code-centric development such as faster development, quality improvement, meaningful validation with architecture enforcing, low skill demand for developers and empowering of domain experts, and portability, interoperability, and reusability [6]. However, there is very little evidence as to whether these advantages can be validated to be economically beneficial [7].

We believe that there are two reasons for the scarcity of cost estimation studies in MDE. Firstly, MDE differs from code-centric development with regards to persuasive use of models which are more abstract than code, and enable automation of all software development life cycle (SDLC) activities to various extents. The raised level of abstraction and automation, and the rest of the advantages, although perceivably reduce the software development effort, are difficult to translate to quantitative units of economic profit. Secondly, if cost estimation techniques are to be applied to MDE, a number of artifacts must be taken into consideration for sizing software, starting with models and encompassing code, documentation, tests, configuration and deployment scripts, and so on. There are differing opinions about how to do this leading to further complexity of cost estimation in MDE.

We take a step in this direction by presenting an approach that attempts to obtain quantitative measure of MDE characteristics responsible for reduced development effort and compute size of various MDE artifacts. For this purpose we propose to use constructive cost model (COCOMO) II [8]. COCOMO II takes *size* of software application as input to its equations for estimating cost and schedule. The equations also use a set of cost drivers, namely *scale factors* and *effort multipliers* [9]. As the names suggest, these cost drivers include a subjective assessment of development practices of an organization that are likely to influence economies of scale and effort required for application development respectively. Our main contribution are an explanation of how COCOMO II cost drivers are likely to be influenced by development practices characteristic of MDE and a way to compute size of variety of MDE artifacts. We do not yet provide calibration and validation of our approach with historical data. While our ongoing work concentrates on this, in this paper we provide only a brief description of how we intend to approach it.

The paper is organized as follows. Section 2 describes the motivation behind our work and presents an outline of our approach. In Section 3, we show how qualitative characteristics of MDE can be quantified. Section 4 presents the nature of various MDE artifacts and how the measurement of size can be automated for them. In Section 5, we discuss how we plan to integrate these and calibrate and validate this integration in COCOMO II. Related work is reviewed in Section 6 and Section 7 concludes the paper.

2 Motivation and Outline

Given the fact that MDE has been promoted as the paradigm to tackle growing complexity of business applications, one would presume that there are many studies providing practical evidence of economic benefits of MDE. But economic studies, particularly cost estimation studies, in the context of MDE are conspicuous only by their absence. A company like ours considers many product pricing strategies like *pricing-to-win* and *pricing by analogy* and so on, and cost estimation would only give a baseline price of the product, upon the basis of which further negotiations commence with a customer. However, cost estimation is generally required by business units. Even for marketing purposes, it is necessary that we are able to show that MDE in fact does increase the productivity in quantitative terms like person-months and calendar time while further easing maintenance and reducing overall cost.

What makes cost estimation in MDE more complex than cost estimation in code-centric applications is the inherent difficulty of capturing beneficial effects of MDE on productivity, including those obtained by automation and raised level of abstraction, in a *measurable* format. Furthermore, an end-to-end MDE toolset such as ours uses models and generates a *variety of artifacts* which must be measured for size and which take different amount of effort to generate and maintain. Currently there are a number of opinions regarding how programming languages and technology platforms affect the sizing of a product [10] and what to do when code is auto-generated as against when it is manually written, as well as how to measure a number of non-code artifacts [11].

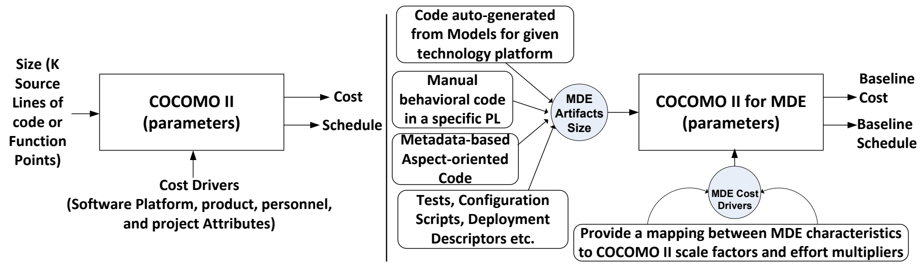

Size (K Source Lines of code or Function Points)
COCOMO II (parameters)
Cost
Schedule
Cost Drivers (Software Platform, product, personnel, and project Attributes)
Code auto-generated from Models for given technology platform
Manual behavioral code in a specific PL
Metadata-based Aspect-oriented Code
Tests, Configuration Scripts, Deployment Descriptors etc.
MDE Artifacts Size
COCOMO II for MDE (parameters)
Baseline Cost
Baseline Schedule
MDE Cost Drivers
Provide a mapping between MDE characteristics to COCOMO II scale factors and effort multipliers

Fig. 1. COCOMO II for Code-centric (left) and MDE (right) Applications

To estimate cost of MDE applications therefore, we need three components; one, a way to quantify MDE characteristics like automation and raised level of abstraction; two, a way to measure size of various MDE artifacts; and three, a cost model that is amenable for including quantified MDE characteristics and adjusted sizes of various MDE artifacts. The cost model we are investigating for this purpose is COCOMO II as it provides these facilities apart from being a widely used cost estimation model. Figure 1(left) shows COCOMO II used for code-centric applications with size and cost drivers as input and cost and schedule as outputs. Instead of modifying existing variables, adding extra variables, or post-processing results of COCOMO (as in many COCOMO derivatives [12]), at this early stage we use COCOMO II itself without making any changes; reinterpreting various cost drivers in the context of MDE practices and taking into account sizes of various MDE artifacts as shown in Figure 1(right).

In the following sections, first we show how quantification of MDE characteristics can be achieved by mapping peculiarities of a MDE toolset to COCOMO II cost drivers and then show size measurements for various common MDE artifacts in the context of our MDE toolset.

3 Measuring Qualitative Characteristics of MDE

As shown in Figure 1, we intend to map characteristic of our MDE toolset with COCOMO II cost drivers. We show how this is done by first presenting a brief description of the COCOMO II model itself.

3.1 COCOMO II

COCOMO II scale factors and effort multipliers are cost drivers that capture characteristics of software development that affect the effort to complete a project. All COCOMO II cost drivers are assigned qualitative rating levels that express the impact of the driver on development effort [9]. Each rating level has a value that translates a cost driver's qualitative rating into a quantitative one for use in the model. These rating levels range from the lowest as *very low* and to the highest as *extra high*. This is shown in Table 1.

Table 1. COCOMO II Cost Drivers Features and Rating to Value Conversion

Feature	Very Low	Low	Nominal	High	Very High	Extra High
Feature X for given Cost Driver	↑	↑	↑	↑	↑	↑
	Value ranges for each rating level					

Each cost driver has varying number of features with values ranges for all rating levels as shown in Table 1. Our approach for quantifying MDE characteristics is to explore the possible effects of practices specific to our MDE toolset on the rating levels of COCOMO II cost drivers and determine *general* rating levels. The actual rating levels are determined by data collection and analysis methods such as questionnaires and interviews and then projected onto a quantitative scale by using translation of COCOMO II ratings to values. Before elaborating how characteristics of our MDE toolset relate to various COCOMO II cost drivers, we briefly review practices and capabilities of our toolset.

3.2 Our MDE Toolset

Our basic MDE approach was sufficient for addressing general functionality concern of large business-critical database-centric enterprise applications [13]. However, having developed applications for a number of domains, we found that applications for even the same domain differed in *design strategies, architectural specifics*, and *technology platforms* ⟨d, a, t⟩ concerns [3]. We developed *aspect-oriented modeling* techniques centered around the concept of *building blocks* which are specifications of ⟨d, a, t⟩ concerns in terms of concern-specific metamodels. The aspect-oriented modeling techniques enabled us to provide manageable variations in these concerns [14]. Finally, the business process concern was addressed by supporting families of business processes by extending business process modeling notation with adaptation operators [15].

These model-driven aspect-oriented code generation capabilities were amended with *multi-user repositories* for storing models to provide a unified view while developing with models at different stages of SDLC [16]. This was followed by a *component abstraction* to support notions of private and public *workspaces* and role-based MDE artifact access which made synchronization between offshore and onsite developers possible [13,16]. We also adopted *well-defined baseline* for each phase in the product lifecycle as large software development engagements were required to be delivered in phases [17].

Together with our model and metamodel architecture centered around separation of functionality, design strategies, architectural specifics, technology platforms, and business process concerns, model-aware specification and transformation languages (OMGen and Q++ [18]), multi-user repositories, component abstraction, and versioning and configuration management, we were able to deliver 60+ quality products worldwide.

In the next section, the peculiar characteristics of our MDE toolset stated above are used in finding how features of scale factors are affected with the definitions of features in COCOMO II model definition manual [9].

3.3 COCOMO II Scale Factors and Our MDE Toolset

COCOMO II scale factors apply to a product as a whole. In the following we explain how development practices using our MDE toolset possibly affect COCOMO II scale factors.

Precedentedness. (PREC) This scale factor attempts to capture similarity in products developed by an organization. If the products are largely similar then precedentedness is high. From our experience in delivering 60+ enterprise business applications, we found that a considerable amount of organizational understanding of product objective is required initially. Once this *knowledge is encoded in the models* though, the unified view provided by our model repositories [16] can always be used to arrive at a big picture. Furthermore, our MDE toolset provides end-to-end model to code capability which means that piecemeal frameworks targeting various operational procedures and data processing architectures and algorithms need not be integrated individually. Therefore irrespective of specific domains or specialized requirements, we can maintain *an above nominal degree of precedentedness* in our products.

Development Flexibility. (FLEX) This scale factor attempts to capture flexibility in terms of conformance of software with pre-established requirements and external interface specifications. Better *separation of five concerns* in our MDE toolset [2] means that whatever level of conformance is required; we are generally able to deliver *within accepted trade-off* between early completion and deployment of a product and complete conformance with requirements and specification.

Architecture/Risk Resolution. (RESL) This scale factor indicates the extent to which an organization implements a risk management plan by development and verification of application specifications with scheduled product design reviews. When developing enterprise applications with our toolset, we create a *reference implementation* that spans all architectural layers. It is validated by operational characteristics and this information is shared with eventual stakeholders and finally, code generators are customized as required. Manual reviews are scheduled based on project time line. In case of drastically different architectures, about 25% of development schedule is dedicated to establishing the architectures. With mostly similar requirements differing only in GUI, only 5% development schedule is required to be assigned to architecture building. RESL also captures percent of top architects required to be available at different stages of development. In our case, a high percent of solution architects are required in defining and encoding the architecture. Once the architecture is encoded though, this percentage can be reduced during actual development of the application. In other words, our MDE toolset can achieve *high to very high RESL* ratings in general.

Process Maturity. (PMAT) The process maturity scale factor is determined based on Capability Maturity Model (CMM) from Software Engineering Institute. When CMM rating is available for an organization, it may be used as it is. Since our organization has overall CMM level 5, it means that the estimated process maturity level (EPML) for us would be close to 5[1]. Yet, we believe that a review of each of the key process areas (KPAs) from the standpoint of our MDE toolset is required to obtain a more realistic EPML level for development units in our organization working with our MDE toolset, which is outside the scope of this paper.

Team Cohesion. (TEAM) The team cohesion scale factor as described in COCOMO II considers synchronization in the objectives and cultures of stakeholders and their experience is operating as a team. We believe that the TEAM factor as described in COCOMO II manual is more or less independent of the development paradigm used such as MDE. The coordinated development of a project with team members located in different sites using our MDE toolset is covered under SITE effort multiplier.

In the following, we similarly indicate how development practices using our MDE toolset affect features of COCOMO II effort multipliers.

3.4 COCOMO II Effort Multipliers and Our MDE Toolset

The 17 effort multipliers in the post-architecture model are used to adjust nominal effort required for a software product/solution under development. These multipliers are classified into product factors, platform factors, personnel factors, and project factors reflecting their rationale.

[1] See Quality Framework section in TCS Corporate Facts available on Web at `http://www.tcs.com/about/corp_facts/Pages/default.aspx`.

Product Factors. The rationale behind product factors is that a product that is complex, has high reliability requirements, and requires large test dataset will require more efforts to complete.

Required Reliability. (RELY) indicates extent to which software must perform its intended function and the effect of software failure. Since our toolset provides extensive *automated testing and validation support* with manual reviews, we believe that we are able to restrict financial losses due to software failure to *easily recoverable* ones instead of letting them escalate into high losses.

Database Size. (DATA) multiplier indicates effort in generating and maintaining test data. Since tests are generated automatically for business logic and GUI in our toolset, we consider this effort to be *low* in general.

Product Complexity. (CPLX) captures complexity of control, computational, device-dependent, data management, and user interface management operations in a specific product. As described earlier in Section 3.2, our experience over a number of products suggests that domains we handled presented with high product complexity in general. Clear separation of concerns in our MDE toolset with application development facilitated along GUI, data, and business logic layers, has enabled us to *reduce product complexity to nominal (and in some cases low) levels.*

Developed for Reuse. (RUSE) multiplier takes into account additional effort required to construct reusable components. It is assumed that creation of reusable components requires more generic design of software, elaborate documentation, and extensive testing. The *component abstraction* in our MDE toolset along with automated documentation and test generation implicitly supports this. We have consequently found that it has enabled us to generate different applications in the same vertical for different organizations. This multiplier is between *across program* to *across product line* in general.

Documentation Match for Life Cycle Needs. (DOCU) multiplier indicates the level of required documentation. The ability of our MDE toolset to *automatically generate documentation* places this effort to *nominal* in general.

Platform Factors. These multipliers target hardware-software complex such as **execution time constraint** (TIME), **storage constraint** (STOR), and **platform volatility** (changes in compilers/assemblers supporting development) (PVOL). Out of these, we consider TIME and STOR to be more or less independent of any MDE practices and therefore would map to their nominal values. PVOL, on the other hand, is adequately addressed by our MDE toolset as we have experience in deploying application onto multiple and varied platforms enabling us to *reflect major changes within pre-established timeline.*

Personnel Factors. These multipliers capture the development teams' capability and expertise. When interpreting these in the context of our MDE toolset, we attempt to indicate how the development team is helped by various facilities provided by our MDE toolset.

Analyst Capability. (ACAP) multiplier considers analysis and design ability of an analyst, his efficiency and thoroughness, and his ability to communicate and cooperate. We consider that analysts using our MDE toolset are helped in terms of facilities for designing models and the model repositories enable sharing knowledge with other stakeholders. Helped by component abstraction, high internal cohesiveness, explicit and dependent components and their composition, and ability to expose required and provided interfaces and implementation [16], *analysts in the nominal to high percentile can perform to very high levels* in general.

Programmer Capability. (PCAP) multiplier considers similar abilities as in ACAP, on behalf of programmers. In our MDE toolset, programmer's job is restricted to providing business logic. Since business logic itself is written in Q++, a high level language, code generators for project-specific technologies take care of intricate details of technology platforms. Because documentation and test are automatically generated, programmers' job is made *easier to a large extent.* As our code generators are themselves written in another model-aware language called OMGen [18], developers of the code generators are similarly helped.

Personnel Continuity. (PCON) This multiplier considers annual personnel turnover. In the context of our MDE toolset, we interpret this multiplier as *the degree to which MDE makes it easier for the new personnel to pick up where earlier personnel left* and *how much of the personnel must continue over a longer period of time.* Using our MDE toolset, we have observed that solution architects are not needed after the architecture is defined and encoded. Domain experts though need to be present over most of the duration of the project. Various graphical views and mild learning curve in learning high level model aware language with much less number of operations and functionality than a general purpose language enables a quick turnover of programmers as well. This enables us to maintain *above nominal to high continuity* in general.

We believe that the effort multipliers **application experience** (APEX), **platform experience** (PLEX), and **language and tool experience** (LTEX) would be largely influenced by *the maturity of our toolset.* We have observed that high level of abstraction and code generation along with other development practices delineated earlier indicate that otherwise nominal level experience over application, platform, and language/tools translates into *roughly between high and very high* when using our MDE toolset.

Project Factors. These effort multipliers account for the influence of such factors as use of modern software tools, location of development team, and effect of compression of project schedule on estimated efforts.

We have indicated earlier that our MDE toolset encompasses all SDLC stages. Furthermore, the use of multi-user repositories for models and business logic specifications along with versioning and configuration management support means that **use of software tools** (TOOL) multiplier evaluates in general to *support for strong, mature, proactive life-cycle tools, well integrated with processes, methods, and reuse.* **Multisite development** (SITE) multiplier captures collocation of team members and communication support. In the context of our MDE toolset, multisite development is *made easier* with workspace realization and role-based secure access of various MDE artifacts, indicating a very high connectedness. **Required development schedule** (SCED) effort multiplier maps to 1 for most projects as changes in schedules can be more or less effectively managed with buffer times provided over the baseline analogy estimates.

3.5 MDE Characteristics Influencing COCOMO II Cost Drivers

To determine the ratings for our MDE toolset, we use questionnaires and assisted interviews in a manner similar to [19] and [20]. The respondents are chosen from the personnel of our company who have been involved in the development and use of our MDE toolset for a period 5-15 years.

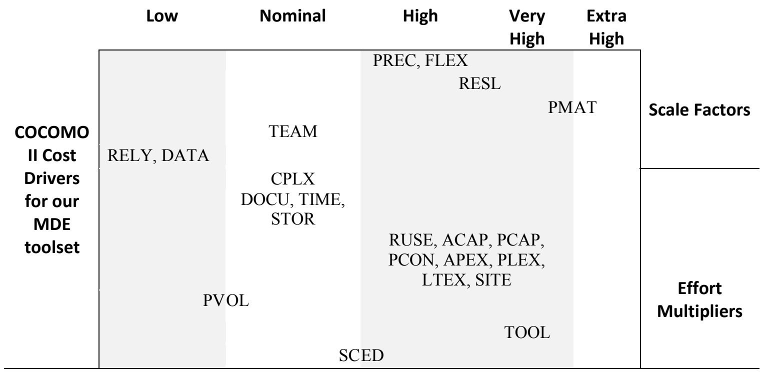


Fig. 2. Distribution of COCOMO II Cost Drivers over Rating Levels for MDE Applications; Response of 16 Team Leaders to 45 Multiple Choice Questions

Initial questionnaire responses and interviews with experienced team leaders revealed the distribution of COCOMO II cost drivers over rating levels shown in Figure 2. It indicates that a number of cost drivers fall into rating levels of high and very high (drivers for which higher ratings indicate less efforts) and low (drivers for which lower ratings indicate less efforts such as RELY and DATA) resulting in economies of scale and reduced overall effort.

We have shown only the ranges of rating levels that each driver falls into in Figure 2 rather than actual values for two reasons; first, these values are not yet calibrated and validated, and second, we have considered responses only of experienced team leaders and not other types of roles. Yet, the distribution seen here could be considered representative of effort reducing benefits of our MDE toolset with key contributing characteristics illustrated in Table 2.

As elaborated in the previous section, certain MDE characteristics affect both scale factors and effort multipliers in COCOMO II. These are shown in Table 2. We have also cited the reference(s) where a particular MDE characteristic was explained in the context of our MDE toolset.

Table 2. Relation between MDE Characteristics and COCOMO II Cost Drivers; TIME and STOR are Independent of any Characteristic

	Raised Level of Abstraction			**Automation**			**Tooling**
	Separation of Functionality, Business process, Design Strategies, Architectural Specifics, and Technology Platforms concerns	Component abstraction	Model-aware language for code generation and automated testing	Automated consistency validation and periodic manual code reviews	Automated generation and maintenance of documentation	Multi-user repositories of models + versioning and configuration management support	Mature tooling for MDE SDLC
	[2]	[16]	[14, 17]	[3, 4, 13, 16]	[16]	[16]	[16]
COCOMO II Scale Factors	**FLEX**			**RESL**	**DOCU**		**PREC, PMAT**
COCOMO II Effort Multipliers	**CPLX**	**RUSE, ACAP**	**PCAP, PVOL**	**RELY, DATA**		**SITE, PCON**	**APEX, PLEX, LTEX, TOOL, SCED**

We have roughly divided the characteristics of our MDE toolset between two core MDE characteristics, namely raised level of abstraction and automation. Mature tooling, enabled on the basis of these two, results in reduced effort as indicated by many COCOMO II cost drivers that are influenced by it. Although the MDE characteristics noted in Table 2 and distribution of cost drivers over rating levels shown in Figure 2 are specific to our toolset, we believe that mapping provided by us between these can be taken as a starting point by others. For instance, capabilities similar to our MDE toolset such as separation of concerns with component-like abstraction, model-aware Q++-like language(s) for automated code generation, automated consistency validation, tests, and document generation, model persistence and query with repositories and versioning support, role-based MDE artifact access, and mature tooling for MDE SDLC can be similarly mapped to COCOMO II cost drivers in case of other MDE toolsets as elaborated in the previous section.

Having obtained quantitative measures of MDE characteristics, we explain in the next section, how to obtain size measure of various MDE artifacts which is another input to COCOMO II.

4 Measuring MDE Artifacts

Our MDE toolset uses four different kinds of artifacts, namely auto-generated code (skeletal class code, queries, GUI code) [14], manually added code (code written in Q++ for method bodies, services, etc.) [16], several non-code artifacts such as tests, deployment descriptors, user documentation, and configuration scripts [17] and finally, metadata-based aspect-oriented code for $\langle$d, a, t$\rangle$ concerns which is application-specific [18].

Table 3 gives an idea about general size and technology platforms of some of the applications we delivered using our MDE toolset [3]. At the time when we measured the source lines of code (SLOC) of these applications, we did not

Table 3. Final SLOC of Enterprise Applications [3]

Product	# classes/screens	Final SLOC (K)	Technology Platforms
Straight-through Processing	334/0	3271	IBM S/390, Sun Solaris, Win NT, C++, Java, ICS, MQ Series, WebSphere, DB2
Negotiated Dealing	303/0	627	IBM S/390, Sun Solaris, Win NT, C++, Java, CICS, MQ Series, COM+, DB2
Distributed Management	250/213	2670	HP-UX, Java, JSP, WebLogic, Oracle, EJB
Insurance	105/0	2700	IBM S/390, Sun Solaris, C++, Java, CICS, DB2, CORBA

make the distinction between auto-generated code, manual code, non-code artifacts and application-specific code. Instead, we computed the SLOC of complete application codebase. We have known that effort required for each of above mentioned artifacts is slightly different and it should be accounted for. We delineate the separate SLOC computation for each of these artifacts next.

4.1 Separate Size Computation of MDE Artifacts

COCOMO II model definition module contains an SLOC checklist which *excludes* code generated with source code generators [9], but will be probably included in future [21]. However a number of researchers state that it is necessary to make the distinction between different kinds of code [10, 11]. Additionally, COCOMO II considers logical SLOC rather than physical SLOC in order to avoid the calibration and validation data to become language-specific. In the following, we indicate different categories of code in our MDE toolset and its SLOC computation.

Table 4. Auto-generated Code Adjustment Factors [11]; GL- Generation Language

Auto-generate Code in	To obtain Logical SLOC Multiply Auto-generated Code SLOC by
2GL	1
3GL (C, Cobol)	0.25
4GL (SQL, Perl)	0.06
Object-oriented (C++, Java, Python)	0.09

Auto-generated Code. McDonald et al. observe that auto-generated code needs to be measured properly lest its measurement is inflated [10]. Lum et al. argue similarly stating that auto-generated code is not free and takes some effort and therefore needs to be considered along with the manually added code [11]. They suggest that since productivity level of developing auto-generated code differs from other code, it must be converted so that it becomes comparable to SLOC of non-auto-generated code. This is indicated in Table 4.

Manually Added Code. Our model-aware Q++ language is used to write bodies of methods generated from the class models. The final code is generated depending upon the target technology platform language[2]. Table 5 shows that based on target programming language the logical SLOC differs as well. Depending upon whether code was written in a programming language or Q++, SLOC can be counted directly by using SLOC counter for that language, or using a general purpose statement counter and adjusting it by multiplying by 0.06 as Q++ is a fourth generation language.

Table 5. Language Adjustment Factors [11]

Technology Platform Language	To obtain Logical SLOC reduce physical SLOC by
3GL	25%
4GL	40%
Object-oriented	30%

Non-code Artifacts. Adjustments have also been suggested for counting of non-code artifacts like database scripts, configuration and deployment scripts, test cases, and so on, which also take effort to create and must be accounted for [11]. The exact SLOC of these artifacts can be obtained using a general purpose counter and then multiplied by 0.06 because these are auto-generated.

Metadata-Based Aspect-Oriented Code. Depending on domain requirements, it is possible that substantial amount of code is generated using metadata-based aspect-oriented code generation techniques [14]. This is the code for ⟨d, a, t⟩ concerns mentioned earlier in Section 3.2. Code that is generated for ⟨d, a, t⟩ concerns is conditional in nature, in the sense that model-to-text transformation takes place differently based on choices along these concerns in specific applications. We have already provided an automation method and tool for counting SLOC of ⟨d, a, t⟩ concerns [22] using software product line concepts.

These adjustments are incorporated in counting the complete SLOC of application developed using our MDE toolset as described next.

4.2 Calculating Complete SLOC for MDE Applications

Starting with models, steps enlisted below can be followed to automate counting and adjusting logical SLOC sizes of various MDE artifacts:

[2] It is also possible to write method bodies directly in the target programming language.

- Auto-generated code is computed by using method similar to [23]. Since we use only class models, other UML artifacts need not be considered in our case as in [23]. Furthermore, we measure only the skeletal code rather than simulating average SLOC of method bodies, since method bodies are added manually which we measure separately. The SLOC is adjusted first by 0.09 and then by language adjustment factor for the target language as indicated in Table 5.
- Manually added code is measured using a language-based SLOC counter and adjusted according to language adjustment factor as shown in Table 5. If this code is written in Q++, then both auto-generation adjustment factor of 0.06 and language adjustment factor is required to be used.
- Non-code artifacts are measured using general purpose SLOC counter. Since they are auto-generated, the SLOC is adjusted by 0.06.
- Application-specific code due to ⟨d, a, t⟩ concerns is calculated by extending code generators with SLOC counting statements. We account for code generator extension effort along with the auto-generated code adjustment factor when computing the contribution of this code.

With the MDE characteristics captured in terms of scale factors and effort multipliers and the MDE artifacts sizes computed in terms of SLOC, we have everything needed for the estimation of cost and duration using COCOMO II as explained next.

5 Proposed Calibration and Validation of COCOMO II for MDE

The data required for calibration and validation of COCOMO II is values of COCOMO II cost drivers, size of products in KSLOC, duration of the project in calendar time, duration of person-months i.e., actual time spent by staff working on the project, how much of the staff worked on the given project and their salaries [9,24]. A calibration and validation method such as k-fold cross-validation can be used for calibrating this data [25]. Generally, our clients are charged by hours spent on their project and timekeeping record is maintained over the duration of development. This is how we plan to obtain the rest of the data needed for calibration and validation. Note that there are two reasons why we do not yet present calibration and validation of proposed approach as explained in the following sections.

5.1 Relevance of Response

The results presented in Section 3.5 were obtained based on the responses of experienced team leaders who manage, train, and help developers, and were developers working with our toolset previously and whose response we could get in time. In our personnel we also have consultants (who are in contact with clients and determine product scope, specifications of the concerned system, and also partake in sales and support), domain experts (from our largest customers for the products under consideration), solution architects, and researchers (who were involved in conceptualizing various aspects of our MDE toolset and technology transfer). We believe that in order to get more accurate values for COCOMO II cost drivers, we will have to take into consideration as to respondents of which role are in a position to make the best subjective judgments about certain cost drivers as shown in Table 6.

Table 6. COCOMO II Cost Drivers and Associated Roles: 1- Team Leaders, 2- Consultants, 3- Domain Experts, 4- Solution Architects, 5- Researchers

PREC	FLEX	RESL	PMAT	TEAM	RELY	DATA	CPLX	RUSE	DOCU	TIME	STOR	PVOL	ACAP	PCAP	PCON	APEX	PLEX	LTEX	SITE	TOOL	SCED
1,4	1,3,4	1	2	-	2,3,4	1	1,2,4	2,5	1	-	-	1,4	1,3,4	1	1	1	1	1	1,4	1-5	2,4

Although each respondent answers questions related to all cost drivers, the value of each cost driver will be calculated by giving more weightage to the answers by the respondents of role type indicated in Table 6. This will give us values of COCOMO II cost drivers that are closer to reality in the context of our MDE toolset.

5.2 Data Availability

Our MDE toolset has evolved over time and some of our products have spanned 5-15 years of development, maintenance, and enhancement cycle and timekeeping records for some artifacts have not been digitally maintained. Peculiarities of our business model and multisite development means that in some cases model repositories and codebases are confidential. Making available sufficient and reliable data in time and to be able to validate our calibrations has proven difficult to us as it has in other cost estimation studies [6]. Nevertheless, as we stated in Section 2, the business relevance of these studies is helping us getting there and in near future we will be able to calibrate and validate data related to sufficient number of complete products.

6 Related Work

Assessing Benefits of MDE. Various social, technical, and organizational factors that influence success of MDE were studied in [7]. It was also found that maturity of toolset capable of carrying out various MDE SDLC tasks is the main determinant of MDE success [6]. The factors that affect effort such as better communication between stakeholders with models, quick response to changes due to automation, and maturity of toolset, etc., expressed in these studies are in line with our findings enlisted in Table 2. On the other hand, these as well as many other qualitative studies in MDE which we do not cite for the lack of space, do not consider economic effect of these perceived MDE advantages as done by us.

Sizing Studies in MDE. There are many proposals to estimate the size of code or function points starting with UML models, for instance use case and class points as in [23]. Other model sizing studies focus on object-oriented constructs and metrics over them [26] or use metrics specification metamodel to generate measurement software [27], disregarding the existence of various MDE artifacts as in an industrial setting. In contrast, we consider code categories that take different effort to generate and maintain and compute their SLOC separately.

Our Estimation Studies in MDE. We have previously presented studies which consisted of total number of screens and effort in person-months per screen [28] and distribution of development effort in our MDE toolset for specific activities [17]. These studies nevertheless were not aimed at cost estimation.

COCOMO II for Large Applications. COCOMO II was applied to a set of 10 industrial projects in [19] where values of cost drivers were determined through interviews. A detailed description of calibration procedure is provided in [19] which is useful to us. In contrast to the projects ranging 5-38 KSLOC considered in this case study, our projects are very large reaching a few thousand KSLOC as shown in Table 3 earlier. Application of COCOMO II in banking and insurance environment by [20] describes a setup for measurement environment for calculating SLOC and workload times. Cost drivers are determined through questionnaire and conversion factors are used for reconciling differences between SLOC of many programming languages. Both these studies are for code-centric development in contrast to our approach.

7 Conclusion

We showed in this paper one way to estimate cost of MDE application by mapping practices of our MDE toolset to COCOMO II cost drivers and automating size measurements of various MDE artifacts. MDE benefits could be roughly grouped among raised level of abstraction, automation, and mature tooling which influence various cost drivers and thus get reflected in the cost calculations. Also these lead to generation of various code and non-code artifacts in different programming languages and technology platforms. The right amount of effort for each kind of artifact is obtained by various adjustment factors. Actual use of COCOMO II for MDE would need calibration of various constants in COCOMO II effort, cost, and duration (schedule) equations and validation which is ongoing. Because core characteristics of MDE are bound to be present in any end-to-end MDE toolset in some or the other form and to varying extent, the method described in the paper can be applied for other industrial MDE toolsets and practices based upon them even though we present it in the context of our MDE toolset.

References

1. Kulkarni, V., Venkatesh, R., Reddy, S.: Generating Enterprise Applications from Models. In: Bruel, J.-M., Bellahsène, Z. (eds.) OOIS 2002. LNCS, vol. 2426, pp. 270–315. Springer, Heidelberg (2002)
2. Kulkarni, V., Reddy, S.: Separation of Concerns in Model-Driven Development. IEEE Software 20(5), 64–69 (2003)
3. Kulkarni, V., Reddy, S.: Model-Driven Development of Enterprise Applications. In: Nunes, N.J., Selic, B., da Silva, A.R., Álvarez, J.A.T. (eds.) UML Satellite Activities 2004. LNCS, vol. 3297, pp. 118–128. Springer, Heidelberg (2005)
4. Kulkarni, V., Reddy, S.: Introducing MDA In a Large IT Consultancy Organization. In: APSEC, pp. 419–426. IEEE Computer Society (2006)
5. Mellor, S., Rioux, L., Bézivin, J.: MDA-Components: Is There a Need and a Market? Technical Report OMG document ad/03-06-01, OFTA/MDA users SIG/MDA working Group (June 2003)
6. Mohagheghi, P., Dehlen, V.: Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 432–443. Springer, Heidelberg (2008)
7. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical Assessment of MDE in Industry. In: Taylor, R.N., Gall, H., Medvidovic, N. (eds.) ICSE, pp. 471–480. ACM (2011)
8. Boehm, B., Clark, H., Brown, R., Chulani, M., Ray, S.: Bert: Software Cost Estimation with COCOMO II with CDRom, 1st edn. Prentice Hall, Upper Saddle River (2000)
9. Boehm, B., Abts, C., Horowitz, E., Brown, A.W., Madachy, R., Chulani, S., Reifer, D., Clark, B., Steece, B.: COCOMO II Model Definition Manual. Technical report, Center of Software Engineering at USC (2000)
10. McDonald, P., Strickland, D., Wildman, C.: Estimating the Effective Size of Auto-Generated Code in a Large Software Project. In: 17th International Forum on COCOMO® and Software Cost Modeling (October 2002)
11. Lum, K., Bramble, M., Hihn, J., Hackney, J., Khorrami, M., Monson, E.: Handbook of Software Cost Estimation. Technical Report JPL D-26303, Rev. 0, Jet Propulsion Laboratory (May 2003)
12. Boehm, B., Valerdi, R., Lane, J.A., Brown, A.W.: COCOMO Suite Methodology And Evolution. CrossTalk - The Journal of Defense Software Engineering (April 2005)
13. Kulkarni, V., Reddy, S.: A Model-Driven Approach for Developing Business Applications: Experience, Lessons Learnt And A Way Forward. In: Shroff, G., Jalote, P., Rajamani, S.K. (eds.) ISEC, pp. 21–28. ACM (2008)
14. Kulkarni, V., Reddy, S.: An Abstraction for Reusable MDD Components: Model-based Generation of Model-based Code Generators. In: Smaragdakis, Y., Siek, J.G. (eds.) GPCE, pp. 181–184. ACM (2008)
15. Kulkarni, V., Barat, S.: Business Process Families Using Model-Driven Techniques. In: Zur Muehlen, M., Su, J. (eds.) BPM 2010 Workshops. LNBIP, vol. 66, pp. 314–325. Springer, Heidelberg (2011)
16. Kulkarni, V., Reddy, S., Rajbhoj, A.: Scaling Up Model Driven Engineering – Experience and Lessons Learnt. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part II. LNCS, vol. 6395, pp. 331–345. Springer, Heidelberg (2010)
17. Kulkarni, V., Barat, S., Ramteerthkar, U.: Early Experience with Agile Methodology in a Model-Driven Approach. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 578–590. Springer, Heidelberg (2011)

18. Barat, S., Kulkarni, V.: Developing Configurable Extensible Code Generators for Model-Driven Development Approach. In: SEKE, Knowledge Systems Institute Graduate School, pp. 577–582 (2010)
19. Dillibabu, R., Krishnaiah, K.: Cost Estimation of a Software Product Using COCOMO II.2000 Model - A Case Study. International Journal of Project Management 23(4), 297–307 (2005)
20. De Rore, L., Snoeck, M., Dedene, G.: COCOMO II Applied In a Banking and Insurance Environment: Experience Report. Open Access publications from Katholieke Universiteit Leuven urn:hdl:123456789/119117, Katholieke Universiteit Leuven (2006)
21. Boehm, B.W., Valerdi, R.: Achievements and Challenges in COCOMO-based Software Resource Estimation. IEEE Softw. 25, 74–83 (2008)
22. Sunkle, S., Kulkarni, V., Roychoudhury, S.: Measuring Metadata-Based Aspect-Oriented Code In Model-Driven Engineering. In: 3rd Workshop on Emerging Trends in Software Metrics (WETSOM) at International Conference on Software Engineering, ICSE (accepted, June 2012)
23. Kim, S., Lively, W., Simmons, D.: An Effort Estimation by UML Points in Early Stage of Software Development. In: Arabnia, H.R., Reza, H. (eds.) Software Engineering Research and Practice, pp. 415–421. CSREA Press (2006)
24. Clark, B., Devnani-Chulani, S., Boehm, B.W.: Calibrating the COCOMO II Post-Architecture Model. In: ICSE, pp. 477–480 (1998)
25. Menzies, T., Port, D., Chen, Z., Hihn, J., Stukes, S.: Validation Methods For Calibrating Software Effort Models. In: Roman, G.C., Griswold, W.G., Nuseibeh, B. (eds.) ICSE, pp. 587–595. ACM (2005)
26. Reissing, R.: Towards A Model For Object-Oriented Design Measurement. In: Procedings of the 5th ECOOP Workshop on Quantitative Approaches in Object Oriented Software Engineering, pp. 71–84. Springer (2001)
27. Monperrus, M., Jézéquel, J.M., Champeau, J., Hoeltzener, B.: A Model-Driven Measurement Approach. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 505–519. Springer, Heidelberg (2008)
28. Mohan, R., Kulkarni, V.: Model Driven Development of Graphical User Interfaces for Enterprise Business Applications – Experience, Lessons Learnt and a Way Forward. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 307–321. Springer, Heidelberg (2009)

Evaluating the Effort of Composing Design Models: A Controlled Experiment

Kleinner Farias[1], Alessandro Garcia[1], Jon Whittle[2], Christina Chavez[3], and Carlos Lucena[1]

[1] OPUS Research Group/LES, Informatics Department, PUC-Rio, Brazil
`{kfarias,afgarcia,lucena}@inf.puc-rio.br`

[2] School of Computing and Communications, Lancaster University, UK
`whittle@comp.lancs.ac.uk`

[3] Department of Computer Science, Federal University of Bahia, Brazil
`flach@dcc.ufba.br`

Abstract. The lack of empirical knowledge about the effects of model composition techniques on developers' effort is the key impairment for their widespread adoption in practice. This problem applies to both existing categories of model composition techniques, i.e. specification-based (e.g. Epsilon) and heuristic-based (e.g. IBM RSA) techniques. This paper reports on a controlled experiment that investigates the effort to: (1) apply both categories of model composition techniques, and (2) detect and resolve inconsistencies in the output composed models. The techniques are investigated in 144 evolution scenarios, where 2304 compositions of elements of class diagrams were produced. The results suggest that: (1) the employed heuristic-based techniques require less effort to produce the intended model than the chosen specification-based technique, (2) the correctness of the output composed models generated by the techniques is not significantly different, and (3) the use of manual heuristics for model composition outperforms their automated counterparts.

Keywords: Model composition effort, empirical studies, effort measurement.

1 Introduction

Model composition plays a central role in many software engineering activities, including the evolution of design models [5,8]. Developers may spend some considerable effort applying model composition techniques to compose M_A and M_B. As a consequence, both academia and industry are increasingly concerned with developing effective techniques for composing design models (e.g. [5,10][14-19]). Model composition can be defined as a set of tasks that should be performed over two (or more) input models, M_A and M_B, in order to produce an output intended model, M_{AB}.

Existing techniques that support model composition can be classified as specification-based techniques (e.g. Epsilon [15]), and heuristic-based techniques (e.g. the heuristics supported by the IBM Rational Software Architect (RSA) [16]). In the first case, developers explicitly specify the correspondence and composition relations between the elements of the input models (M_A and M_B) to give rise to M_{AB}. In the second case, developers use a set of predefined heuristics, which "guess" the relations between the elements of M_A and M_B before producing M_{AB}.

However, instead of producing the output intended model, M_{AB}, as would be expected, the technoiques may produce an output composed model, M_{CM}, with inconsistencies. These inconsistencies often result from the incorrect resolution of conflicting changes between the model element from M_A and M_B. If M_{CM} and M_{AB} do not match ($M_{CM} \neq M_{AB}$) due to inconsistencies in M_{CM}, developers will need to invest some extra effort to detect and resolve the inconsistencies in M_{CM} so that it can be transformed into M_{AB}. Note that the key motivation for applying composition techniques is to reduce the effort of the developers to produce the output intended model [17, 18]. The proponents of specification-based techniques claim that explicit composition specifications entail a more systematic way to compose M_A and M_B [8, 17]; hence, developers expect to save effort by using them. That is, the conventional wisdom [5, 8, 17, 18] is that a precise composition specification favors the production of correctly composed models (i.e. where $M_{CM} = M_{AB}$), thereby minimizing the developers' effort.

To date, however, there is little evidence to confirm (or not) this expectation. As a result, developers use model composition techniques without any support of empirical knowledge regarding their effects on the effort to apply them as well as to detect and resolve inconsistencies in M_{CM}. If a particular composition technique reduces effort, but has a detrimental effect on the model correctness (or vice-versa), it is quite arguable whether developers may use it in mainstream software projects, where time and cost are tight. Having empirical knowledge at hand, developers can choose and adopt composition techniques in a rational way. Today, the adoption of the techniques is based on evangelists (often divergent) opinions.

This paper reports empirical findings on the use of specification-based and heuristic-based composition techniques (Section 2.3) to evolve design models. We have conducted a controlled experiment to evaluate and compare such techniques with respect to the developer's effort and model correctness in the context of evolving design models (Section 3). A total of 24 subjects carried out 144 compositions of UML class diagrams with the support of such techniques. The comparative analysis (Section 4) embodied the effort of applying alternative composition techniques, detecting inconsistencies and resolving them in the output composed model. The main surprising results, supported by statistical analysis, suggest that: (1) the specification-based technique required more effort to produce the intended model than the selected heuristic-based techniques; and (2) there was no significant difference in the correctness of the output composed models generated by the assessed techniques.

The contributions of this paper are (a) empirical findings on the impact of heuristic and specification-based composition techniques on developers' effort to apply techniques, detect inconsistencies and resolve inconsistencies; (b) insights about how to evaluate the developers' effort, reduce error proneness in model composition, and minimize side effects of composition techniques in practice; and finally, (c) to serve as an in-depth example of how controlled experiments can be conducted to evaluate and compare model composition techniques. We also discuss the threats to validity (Section 5), the limitations of related work (Section 6), and concluding remarks (Section 7). Although we cannot generalize our empirical findings to other model composition techniques, our exploratory experiment stands for a trailblazing contribution to improve understanding of the potential effects of model composition techniques on developers' effort.

2 Background

2.1 Model Composition Effort

In this study, model composition effort is described by an effort equation (Fig. 1). Developers often invest effort to realize three activities to compose the base model, M_A, (the model to-be changed) and the delta model, M_B (i.e. the changes), to produce M_{CM}. The first activity, the application of the model composition technique, is represented by $f(M_A,M_B)$ in the equation. The additional effort is usually invested to detect inconsistencies in M_{CM} – represented by, $diff(M_{CM},M_{AB})$ – and to resolve these inconsistencies – represented by $g(M_{CM})$. If M_{CM} perfectly matches the intended model, M_{AB}, then $diff(M_{CM}, M_{AB}) = 0$ and $g(M_{CM}) = 0$. Otherwise, additional effort is required to deal with inconsistencies, meaning that $diff(M_{CM},M_{AB}) > 0$ and $g(M_{CM}) > 0$.

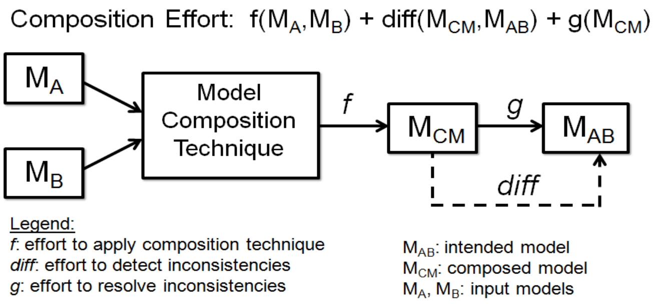
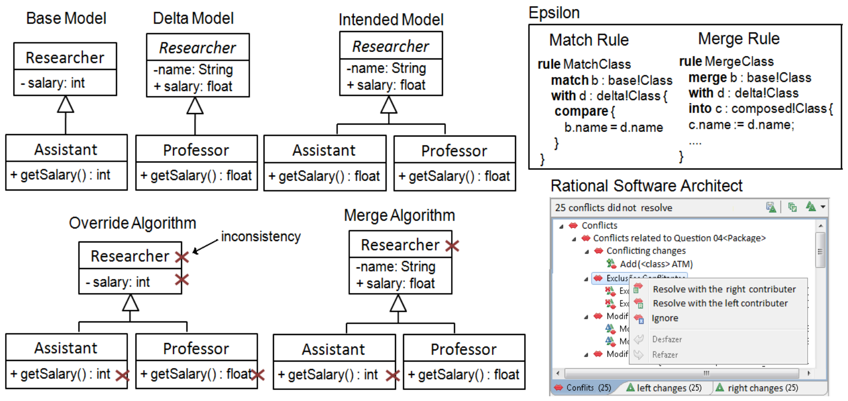

Composition Effort: f(MA,MB) + diff(MCM,MAB) + g(MCM)
MA
MB
Model Composition Technique
f
MCM
g
MAB
diff
Legend:
f: effort to apply composition technique
diff: effort to detect inconsistencies
g: effort to resolve inconsistencies
MAB: intended model
MCM: composed model
MA, MB: input models

Fig. 1. Overview of model composition effort: an equation.

2.2 Composition Conflicts and Inconsistencies

It is well known that the properties of the model elements of M_A and M_B may conflict with each other. Fig. 2 shows a simple example of composition conflict. In the base model, the *Researcher* is defined as a concrete UML class (i.e. *Researcher.isAbstract* = false) whereas in the delta model, *Researcher* is an abstract class (i.e. *Researcher.isAbstract* = true). Before composing, the developers need to properly answer the question: should class *Researcher* be an abstract class (or not)? In this particular case, the correct answer is that the *Researcher* must be abstract (see the intended model in Fig. 2). However, conflicts may be converted into inconsistencies in M_{CM} when unexpected values are set to the properties of the model elements. Fig. 2 shows that the class *Researcher* produced by the override and merge algorithms (Section 2.3) is a concrete class (*isAbstract* = false) instead of an abstract one (*isAbstract* = true), as would be expected. Because of this inconsistency, the output composed model is *not compliant* with the intended model. Two categories of inconsistencies can emerge, including:

- *Syntactic inconsistency* emerges when a composed model element does not conform to the rules defined in the modeling language's metamodel. For example, a class must have attributes with different names.
- *Semantic inconsistency* arises when the meaning of the elements of the composed model does not match with the meaning of the intended model elements. For instance, a class in M_{CM} has an unexpected method, or it requires functionality from other classes that no longer exist after the composition.



```
rule MatchClass
  match b : base!Class
  with d : delta!Class {
   compare {
     b.name = d.name
   }
}
```

```
rule MergeClass
  merge b : base!Class
  with d : delta!Class
  into c : composed!Class {
  c.name := d.name;
  ....
}
```

```
25 conflicts did not resolve
Conflicts
  Conflicts related to Question 04<Package>
    Conflicting changes
      Add(<class> ATM)
```

Fig. 2. Illustrative example.

In our study, we focus on semantic inconsistencies because they cannot be automatically identified using model composition techniques. They often require some intervention from software developers. In addition, they are mainly responsible for non-trivial composition problems during the model evolution [8]. As a consequence, they often require more effort and are more detrimental to the correctness of the output model than syntactic inconsistencies [9]. The categories of semantic inconsistencies considered are: (1) a model element in M_{CM} is not compliant with the corresponding one in M_{AB}; (2) model elements are missing from M_{CM}, or should nor be defined in M_{CM}; (3) model elements are unexpectedly duplicated according to M_{AB}; and (4) there are dangling relationships between classes, i.e. a model element makes reference to other model elements that do not exist. These categories are the most common types of problems faced by developers dealing with model inconsistencies [4,8]. In our study, we explicitly discriminate each contradicting change (i.e. the conflict) from its consequence in the output model (i.e. the inconsistency).

2.3 Model Composition Techniques

The composition techniques used in our study were Epsilon [15], the representative of specification-based techniques, and two representatives of heuristic-based techniques, namely the IBM RSA [16] and traditional composition algorithms (TRA) [9]. These three techniques were selected as they provide different degrees of automation support. The selected heuristic-based techniques include both an automated technique

(RSA) and a manual technique (TCA) to support model composition. Specification-based techniques cannot be applied manually. Epsilon and IBM RSA are supported by robust, usable tools, an essential prerequisite for a controlled experiment like ours. IBM RSA is an industry-leading tool and it is the most widely used tool in the industry [16]. Epsilon is stable, easy-to-use tool for specification-based composition that was available for our study. Traditional algorithms, such as merge and override, are well explored in the academic literature and have been used to support and guide manual model composition [10, 19]. These techniques are described as follows.

Epsilon (EPS). It provides a hybrid, rule-based language for merging design models [15]. Developers invest effort to edit a set of match and merge rules before producing M_{AB}. Fig. 2 shows an example of these rules. The merge rule specifies that all classes to be composed will have the names of classes from the delta model (i.e., *c.name* := *d.name*). Based on these specifications, developers define how composition relations should be identified.

IBM RSA (RSA). It is one of the most robust modeling tools used in industry [16]. IBM RSA is characterized as a semi-automated model composition technique. Like the Epsilon technique, its use does not ensure that M_{AB} will be always produced. By using the IBM RSA developers should interactively resolve conflicts before producing M_{AB}. Fig. 2 depicts an example of a conflict report. When conflicting changes emerge, developers should decide which changes will be inserted into the output composed model — from the base model (*Researcher.isAbstract* = false) or from the delta model (*Researcher.isAbstract* = true).

Traditional Algorithms (TRA). These algorithms fall in the category of manual, heuristic-based composition techniques. In particular, we focus on three well-established composition algorithms: override, merge and union [9]. These algorithms were chosen for several reasons. First, model evolution scenarios can be decomposed into one or more operations supported by a combination of these algorithms. Second, these algorithms are often used as guidelines for the developers composing OO models manually [10, 19]. Third, we wanted to investigate to what extent the aforementioned automated techniques outperform the use of a classical manual technique for model composition. In the following, we provide a brief definition for override and merge algorithms to be applied to two hypothetical input models, M_A and M_B. We say that two elements from M_A and M_B are corresponding if they have been identified as equivalent in the matching process. Matching can be achieved using any number of standard heuristics, such as match-by-name.

1. Override (direction: M_A to M_B). For all pairs of corresponding elements in M_A and M_B, M_A's elements should override M_B's similar elements. Elements not involved in the correspondence remain unchanged and are inserted into the output model.

2. Merge. For all corresponding elements in M_A and M_B, the elements should be combined. The combination depends on the element type. In this paper, we only consider classes and interfaces — in this case, the combination adds the operations of M_A's elements to those of M_B. Elements in M_A and M_B that are not involved in a correspondence matching remain unchanged and are directly copied to the output model. In Fig. 2, the override and merge algorithms are applied and two composed models are produced with inconsistencies.

3 Experiment Planning

3.1 Experiment Definition

The objective of this study is stated based on the GQM template 2 as follows:

Analyze model composition techniques *for the purpose of* investigating their effects *with respect to* the effort and correctness *from the perspective of* developers *in the context of* evolving design models.

Based on this, we focus on the two research questions:

RQ1: What is the relative effort of composing two input models by using specification-based composition techniques with respect to heuristic-based composition techniques?

RQ2: Is the number of correctly composed models higher when using specification-based techniques than heuristic-based techniques?

3.2 Hypothesis Formulation

Hypothesis 1. We conjecture that although specification-based composition techniques provide a more systematic way to compose the input models, they do not reduce the overall composition effort in practice. We suspect that developers have to invest too much effort to specify the compositions; but, this additional effort is not converted into a higher number of correctly composed models than that produced with heuristic techniques. However, it is by no means obvious that this hypothesis holds. It may be, for example, that specification-based techniques help developers to match and then compose the input models more quickly.

Null Hypothesis 1, $H_{1\text{-}0}$: The specification-based composition technique requires less (or equal) effort than the heuristic-based ones to produce M_{AB} from M_A and M_B.

$H_{1\text{-}0}$: $\text{Effort}(M_A,M_B)_{\text{Specification}} \leq \text{Effort}(M_A,M_B)_{\text{Heuristic}}$

Alternative Hypothesis 1, $H_{1\text{-}1}$: The specification-based technique requires more effort than the heuristic-based ones to produce M_{AB} from M_A and M_B.

$H_{1\text{-}1}$: $\text{Effort}(M_A,M_B)_{\text{Specification}} > \text{Effort}(M_A,M_B)_{\text{Heuristic}}$

We refine this hypothesis in other three *subhypotheses* ($H1_2$, $H1_3$, and $H1_4$). A formulation for these hypotheses is presented in Table 1.

Hypothesis 2. The specification-based technique is expected to produce a higher number of correctly composed models as developers can precisely express the composition relations between the input models. However, it is not clear whether this composition technique can, in fact, help developers to improve the correctness (Cor) of the output model when compared to the use of heuristic approaches. These hypotheses are presented as follows:

Null Hypothesis 2, $H_{2\text{-}0}$: The specification-based technique produces a lower (or equal) number of correctly composed models than the heuristic-based techniques.

$H_{2\text{-}0}$: $\text{Cor}(M_{CM})_{\text{Specification}} \leq \text{Cor}(M_{CM})_{\text{Heuristic}}$

Alternative Hypothesis 2, $H_{2\text{-}1}$: The specification-based technique produces a higher number of correctly composed models than the heuristic-based technique.

$H_{2\text{-}1}$: $Cor(M_{CM})_{Specification} > Cor(M_{CM})_{Heuristics}$

The composition correctness is influenced by the presence (or not) of inconsistencies in the output composed model. Thus, we investigate if the specification-based technique entails (or not) a lower inconsistency rate than the use of the heuristic-based techniques. This new elaborated hypothesis is stated in Table 1.

Table 1. Tested hypotheses

Null Hypothesis	Alternative Hypothesis
$H1_{1\text{-}0}$: $Effort(M_A,M_B)_S \leq Effort(M_A,M_B)_H$	$H1_{1\text{-}1}$: $Effort(M_A,M_B)_S > Effort(M_A,M_B)_H$
$H1_{2\text{-}0}$: $f(M_A,M_B)_S \leq f(M_A,M_B)_H$	$H1_{2\text{-}1}$: $f(M_A,M_B)_S > f(M_A,M_B)_H$
$H1_{3\text{-}0}$: $diff(M_{CM},M_{AB})_S \leq diff(M_{CM},M_{AB})_H$	$H1_{3\text{-}1}$: $diff(M_{CM},M_{AB})_S > diff(M_{CM},M_{AB})_H$
$H1_{4\text{-}0}$: $g(M_{CM})_S \leq g(M_{CM})_H$	$H1_{4\text{-}1}$: $g(M_{CM})_S > g(M_{CM})_H$
$H2_{1\text{-}0}$: $Cor(M_{CM})_S \leq Cor(M_{CM})_H$	$H2_{1\text{-}1}$: $Cor(M_{CM})_S > Cor(M_{CM})_H$
$H2_{2\text{-}0}$: $Rate(M_{CM})_S \geq Rate(M_{CM})_H$	$H2_{2\text{-}1}$: $Rate(M_{CM})_S < Rate(M_{CM})_H$

Effort: Effort to compose the input models (RQ1), S: Specification-based composition technique.

f: Effort to apply the composition techniques (RQ1), H: Heuristic-based.

diff: Effort to detect inconsistencies (RQ1), g: Effort to resolve the inconsistencies (RQ1).

Cor: Correctness of the composition (RQ2), Rate: Inconsistency rate of the composed model (RQ2).

3.3 Context and Subject Selection

The subjects used the Epsilon, IBM RSA and the traditional algorithms to produce model compositions for six software evolution scenarios (Table 2). None of the subjects were familiar beforehand with either the design models or the required changes. The selected evolution scenarios were tasks where developers are not the initial designers of the models. The design models used were fragments of industrial models captured from different application domains, such as financial and simulation of petrol extraction. The experiment was conducted with 16 subjects were professionals from Brazilian companies and 8 subjects were students with professional experience [19]. All professionals held a Master's degree, Bachelor's degree or equivalent, and had a considerable knowledge of software modeling and programming to participate in the experiment [19]. The students were also invited to participate in the experiment, so that we could have subjects with different backgrounds and levels of expertise [1]. They were from two Master and Doctoral programs in Computer Science at two Brazilian universities: Pontifical Catholic University of Rio de Janeiro (PUC-Rio) and the Federal University of Bahia (UFBA). These students attended either a course on "empirical studies in software engineering" at PUC-Rio or a course on "software evolution" at UFBA. The experiments were part of the courses and were performed as practical laboratory exercises. The participant was exposed to the same level of training on the model composition techniques under assessment [19].

Table 2. The tasks of the evolution scenarios

Task	Models	Required Changes to the Base Model
1	Oil Extraction	*Add* one class, one method, and one relationship. *Modify* one class from concrete to abstract.
2	Car System	*Remove* two methods and *modify* the direction of a relationship.
3	ATM	*Add* two classes and *refine* two classes from one. *Remove* this last class.
4	Supply Chain	*Add* two classes and one relationship.
5	Financial	*Remove* one class and *add* two methods to a particular class. *Refine* two classes from one and remove the last one. *Remove* one relationship.
6	Simulation of extraction	*Modify* the direction of five relationships. *Modify* the name of two methods.

3.4 Experimental Design

The experimental design of this study is characterized as a *randomized complete block one* with three treatments, i.e. the use of the three composition techniques. The study had a set of activities that were organized into three phases (see Fig. 3). The subjects were *randomly* assigned and *equally* distributed to the treatments, following a within-subjects design in which all subjects serve in the three treatments [1]. In each treatment, the subjects used a model composition technique to carry out two experimental tasks (Table 2), totaling six tasks performed. Therefore, the experiment design was, by definition, a *balanced design*. Fig. 3 shows through an experimental process how the three phases were organized. The subjects individually performed all activities to avoid any threat to the experimental process. The activities are further described as follows.

Training. All subjects received training to ensure they acquired the needed familiarity with each model composition technique.

Apply the techniques. The participants were encouraged to compose M_A and M_B based upon a description of changes (Table 2) that defines how the model elements of M_A were changed. Note that M_B, the delta model, was pre-prepared. The measure of application effort (time in minutes) was collected during this activity. In addition, the composed model, video and audio records represent the outputs of this activity. Each subject performed this task six times. The video and audio records were later used during the qualitative analyses (Section 4.3). It is important to point out that a participant (subject x) produced M_{CM} in the first phase; in the second phase, other participant (subject n-x) detected and resolved the inconsistencies in M_{CM} in order to produce M_{AB}.

Detect inconsistencies. Subjects reviewed M_{CM} to detect inconsistencies. To this end, they checked if M_{CM} had the changes described in the evolution descriptions and if the contradicting changes between M_A and M_B were correctly addressed. As a result of this activity, we have the measure of detection effort (time in minutes), video and audio records, and a list of inconsistencies identified.

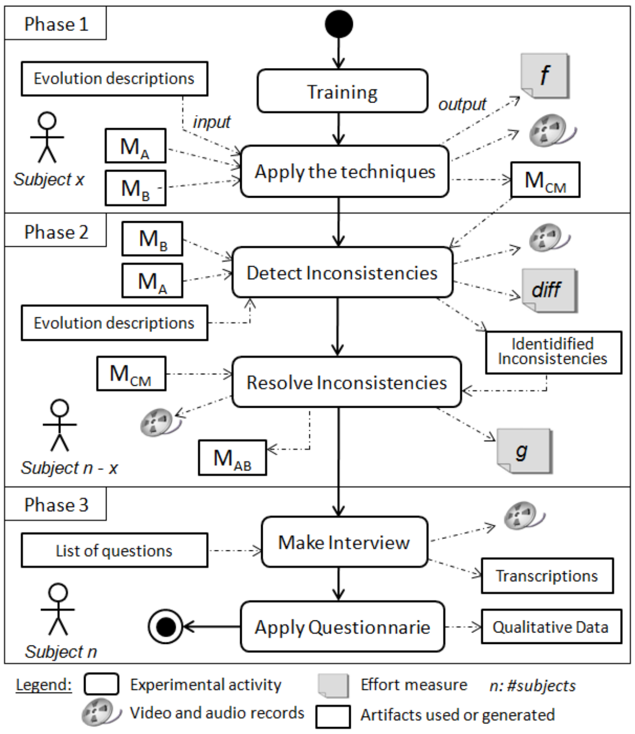


Fig. 3. The experimental process

Resolve inconsistencies. The subjects resolved the inconsistencies previously localized in order to produce M_{AB}. The resolution effort was also measured (time in minutes) and the video and audios were recorded.

Make interview and Answer questionnaire. Subjects reflected on their experience on model composition during the experiment through semi-structured interviews. These interviews helped us to enrich the body of qualitative data collected. The subjects also filled out a questionnaire. This allowed us to collect their academic background and work experience and apply some inquisitive questions.

Material. The models used in our study were UML class diagrams with about 8 classes and 7 relationships. This medium size of the models was essential to perform a controlled study like this and to be in compliance with recommendations from previous work [20]. For example, Asklund et al. [18] recommends that software changes should be as small as possible so that the number of conflicts remains small. In addition, given the time constraints of controlled experiments, the subjects could not be exposed to very large models.

Variables. The independent variable of this study is the choice of composition techniques. We investigate the impact of this independent variable in the following dependent variables:

- Effort. This variable measures the overall time (in minutes) invested by subjects to compose the input models ($H_{1\text{-}1}$). It is elaborated in three other variables: effort to apply model compositions ($H_{1\text{-}2}$), effort to detect inconsistencies ($H_{1\text{-}3}$), and effort to resolve inconsistency ($H_{1\text{-}4}$).
- Correctness. The full correctness of a composition ($H_{2\text{-}1}$) is ensured when the output composed model produced is *correct* with respect to the description of the intended change request (i.e. $M_{CM} = M_{AB}$). We have compared the produced models with the intended models (our 'reference intended models'), produced by the actual developers of those systems from where the input models were extracted. The composed model produced may be rated as either *correct* or *incorrect.* Note that a composed model with one of the previously described inconsistencies (Section 2.2) would be deemed as *incorrect.* We also investigate the *inconsistency rate* of the incorrectly composed model. It represents the ratio of the number *of inco*nsis*tencies* of a composed model divided by its number of model elements ($H_{2\text{-}2}$). The actual developers were consulted when we were unsure about particular inconsistencies in the composed models produced by the subjects.

4 Experimental Results

4.1 RQ1: Effort and Composition Techniques

Descriptive Statistics. The developers invest less effort to produce M_{AB} by using heuristic-based techniques rather than the specification-based technique. In fact, they spent less effort to apply the composition techniques (*f*), detect inconsistencies (*diff*), and resolve inconsistencies (*g*) (Table 3). The traditional algorithms required less effort than the IBM RSA, which in turn required less than the Epsilon. This is a very interesting finding because the common sense would be otherwise i.e., developers would invest less effort by using the Epsilon and IBM RSA. Table 3 shows the descriptive statistics of the collected data. Regarding the median of the general effort, it grew significantly from 11 to 14 and 21 by using RSA and Epsilon, respectively. This superior effort represents an increase by about 27.27 and 90.90 percent. This upward trend was also observed in *f, diff,* and *g*. This evidence, therefore, demonstrates that the developers, in fact, tend to invest less effort with heuristic-based techniques than specification-based one.

Hypothesis Testing. Since the Shapiro-Wilk test [1] indicated deviations from normality, the Wilcoxon signed-rank test and Friedman test were applied. While the Wilcoxon test allowed us to realize a pairwise comparison of the distributions, Friedman test allowed checking if there exist significant differences among the three techniques under investigation. We test H1 (and its subhypotheses) to evaluate the RQ1 in the six experimental tasks (Table 2). Table 4 shows the p-values for the pairwise comparison. Bold p-values highlight statistically significant results (i.e. p-value < 0.05).

They indicate the rejection of the respective null hypothesis. The main feature is that the general composition effort (and *f*, *diff* and *g*) using heuristic-based techniques were significantly lower than using automated techniques in all cases. Still by using the traditional algorithms this significance is higher. Thus, we can reject the H1 null hypotheses (and its $H1_{1\text{-}0}$, $H1_{2\text{-}0}$, $H1_{3\text{-}0}$ e $H1_{4\text{-}0}$). For example, in row 1 of Table 4, for measure *Effort*, between RSA and EPS, the W is negative (-544) and p-value is less than 0.05 (p = 0.001). This means that the composition effort by using the IBM RSA is significantly lower than one using Epsilon. From row 1 it is also possible to notice that only one null hypothesis was not rejected, and in just one case: the effort to detect inconsistencies considering the IBM RSA and Epsilon (p-value = 0.0891). This means that the subjects did not spend substantially different effort to detect inconsistencies in IBM RSA and Epsilon. Therefore, our initial intuition that the specification-based technique would not reduce the composition effort is confirmed.

Table 3. Descriptive statistic for the composition effort

	Effort			**f**			**diff**			**g**		
	[1] RA	RSA	EPS	TRA	RSA	EPS	TRA	RSA	EPS	TRA	RSA	EPS
N	46	46	46	46	46	46	46	46	46	46	46	46
Min	5	5	9	2	3	4	1	1	1	0	0	0
25th	7	11	14	4	6	8.7	2	2	3	0	0	0.5
Med	11	14	21	6	8	12	3	4	4.5	0.5	2	3
75th	18	24	34	9	11	17	5.2	8	8.7	4	7	9
Max	31	66	114	25	22	39	11	22	38	9	22	38
Mean	13.3	18.2	29.1	7.2	9.0	14.8	3.9	5.3	7.7	2.1	3.8	6.6
St.D.	6.9	11.0	23.3	4.4	4.2	8.8	2.4	4.4	8.2	2.9	5.1	9.1

N: #compositions, Min: minimum, Med: median, Max: maximum, StD: Standard Deviation, TRA: traditional, RSA: Rational Software Architect, EPS: Epsilon.

Table 4. Wilcoxon test results for the composition effort

		General Effort			**f(M_A,M_B)**			**diff(M_{CM},M_{AB})**			**g(M_{CM})**		
task	**S**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**	**A**	**B**	**C**
All	p	**0.005**	**0.0001**	**0.001**	**0.02**	**0.0001**	**0.0003**	**0.03**	**0.0003**	0.08	**0.01**	**0.0003**	**0.04**
	W	-420	-900	-544	**-277**	-834	-588	-233	-533	-186	-261	-423	-248
1	p	0.33	0.5	0.5	0.42	0.40	0.3628	0.14	0.5	0.39	0.46	0.39	0.30
	W	6	0	0	-4	5	6	16	-1	4	-2	-4	-7
2	p	**0.01**	**0.003**	0.14	0.23	**0.007**	**0.0342**	**0.01**	0.22	0.23	0.08	0.05	0.22
	W	-32	-36	-16	-12	-34	-27	-21	-8	8	-14	-24	-10
3	p	0.28	**0.01**	0.13	0.37	**0.01**	0.1548	0.27	0.05	0.12	0.23	0.06	0.12
	W	-8	-21	-14	-4	-26	-16	-8	-20	8	-8	-10	12
4	p	0.5	**0.01**	**0.01**	0.29	**0.01**	**0.0171**	0.29	0.06	**0.03**	0.5	**0.01**	**0.04**
	W	-1	-28	-26	-3	-28	-26	3	-19	-22	0	-21	-17
5	p	**0.01**	**0.007**	0.97	0.07	**0.003**	**0.0177**	**0.02**	.0.8	0.19	0.27	0.43	0.5
	W	-26	-36	-20	-18	-36	-31	-11	-25	-11	-8	-3	-1
6	p	**0.04**	**0.03**	0.42	0.21	0.07	0.1094	0.06	**0.01**	0.11	**0.04**	0.12	0.42
	W	-21	-23	3	-9	-18	-13	-12	-28	15	-17	-28	28

W: sum of signed ranks, RSA: IBM rational software architect, EPS: Epsilon, TRA: traditional algorithm, A: TRA vs RSA, B: TRA vs EPS, C:RSA vs EPS*, p: p-value,* S*: statistics.*

4.2 RQ2: Correctness and Composition Techniques

Descriptive Statistics. Fig. 4 shows the correctness of the compositions generated by using the three techniques: traditional algorithms, Epsilon, and IBM RSA in six experimental tasks. The y-axis represents the proportions of the number of M_{AB} achieved by the number of compositions realized in each task using each composition technique, while the x-axis consists of the experiment tasks. Thus, the histogram shows how the correctly composed model happened throughout the experimental tasks.

The main outstanding feature is the lack of a distribution pattern of the proportions of correctly composed models in the tasks. For example, in task 1, TRA produced a lower proportion of correctly composed models than RSA and EPS. That is, the intended model was generated in 42.86 percent of the cases in TRA, whereas 57.14 percent of the cases in RSA and EPS. On the other hand, in task 2, TRA outnumbers RSA and EPS. It produced the intended model in 71.43 percent of the cases, while EPS and RSA produced 28.57 and 57.14 percent of the cases, respectively.

Although TRA has obtained low measures in task 3 in comparison to task 2 (a decrease from 71.43 to 42.86 percent), it still got a superior value compared to EPS and RSA, i.e. value by about three times higher than the measure of EPS and RSA, comparing 42.86 and 14.29 percent. On the other hand, in task 6, this superiority was reversed. RSA got double the value than TRA and EPS, comparing 28.57 and 57.14 percent. Still subjects obtained the intended model by using TRA and RSA in all composition cases, while less than half of the cases in EPS. We have observed that TRA got a higher number of intended models than RSA and EPS. The subjects produced the intended model in 61.90 percent of the compositions using TRA against 59.52 and 42.86 percent using the RSA and Epsilon technique, respectively.

Table 4 shows the descriptive statistics of the inconsistency rate of the composed models. Our initial expectation was that the specification-based technique would minimize the inconsistence rate whereas also get lower measures than the heuristic-based techniques. However, this expectation was not confirmed. We have observed that the inconsistency rate was similar in specification-based and heuristic-based technique in most cases. This means that developers will not produce correctly composed model by using a technique based on composition specifications. Rather, the output models will have equal (or even more) inconsistency rate.

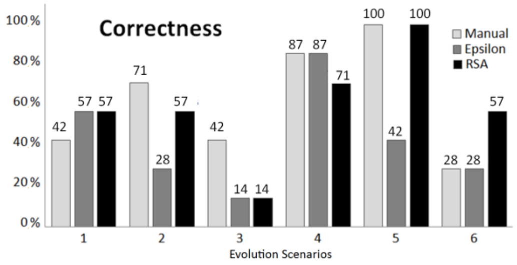


Fig. 4. The correctness of the output composed model

Table 5. The descriptive statistics for the inconsistancy rate

	N	Med	75th	Max	Mean	St D.
TRA	46	0	0.31	1.63	0.26	0.45
RSA	46	0	0.425	1.22	0.21	0.29
EPS	46	0.47	0.78	5.22	0.58	0.88

For example, on average, EPS produced a higher inconsistency rate than TRA and RSA. In general, the mean of the inconsistency rate in Epsilon is two times higher than one TRA and RSA, increasing by about 123 and 176 percent, respectively. Still note that the inconsistence rate in RSA is also higher than in TRA. In short, the inconsistency rate in EPS is higher than RSA, which outnumber TRA. This suggests that the inconsistency rate have favored TRA in comparison with RSA and EPS in most cases. This implies that, to some extent, the number of inconsistencies is decreased whenever the composed model is produced by TRA and RSA.

Hypothesis Testing. We apply the McNemar test to test $H2_1$. Table 6 shows the chi-square statistic and p-values for the pairwise comparisons. In all cases, the p-value was large ($p > 0.05$), so the null hypothesis of $H2_{1\text{-}0}$ cannot be rejected. Although the p-value to the six tasks is not shown in the table, the p-value took values greater than 0.05 in the six tasks. This implies that there is no significant difference between the proportions of the correctly composed models of the composition techniques.

We test $H2_2$ by applying the Wilcoxon test. Table 7 depicts the pairwise p-values for each measure. Bold p-values point out statistically significant results. They also indicate the rejection of the null hypothesis. Note that the sum of signed ranks (W) shows the direction in which the result is significant. For example, in row 2, W is negative (-250) and the p-value is lower than 0.05 (p = 0.0301) for the measure between TRA vs EPS. This means that the inconsistency rate for TRA is significantly lower than in EPS. RSA also obtained an inconsistence rate significantly lower (p = 0.001) than EPS. For instance, in row 1, the W is negative (-5) and p-value is higher than 0.05 for the inconsistency rate between TRA vs RSA. This means that the inconsistency rate for TRA is lower, but no significantly lower than RSA.

Table 6. The descriptive statistic for the inconsistancy rate

Task	Comparison	χ^2	p-value
all	TRA vs RSA	0.27	0.606
	TRA vs EPS	0.75	0.387
	RSA vs EPS	0	1

Table 7. The descriptive statistic for the inconsistancy rate

Tasks	*Statistic*	**Inconsistency Rate**		
		tra vs rsa	***tra vs eps***	***eps vs rsa***
All	p-value	0.4851	**0.0301**	**0.0011**
	W	-5	**250**	**344**

W: sum of signed ranks.

4.3 Additional Observations

We have analyzed the qualitative data (i.e. interviews, video and audio records) to try explaining the results previously mentioned. First, the subjects mentioned that they often had some additional difficulties to match and compose the input model elements by using the specification-based composition techniques. Since they had difficulties to express the semantics of the changes required in each evolution scenario, given the problem at hand. This problem was observed in compositions dominated by relations between the input model elements of the type one-to-many (1:N) or many-to-many (N:N). The following extract from the interview also illustrates, for example, the difficulty related to the understanding of the scope of elements involved to specify a composition: "*…express the changes in match and merge rules is boring…because all overlapping parts of the two input models should be analyzed...this is not a trivial task.*" Second, the IBM RSA tool shows the commonalities and differences between the input models in multiple, partial views. This strategy jeopardizes the creation of a "big picture view" of the output intended model. The following extract confirms this observation: "*I have to check more than three views to complete something…it is very complicated when more complex changes happen... because I have to mentally "infer" a complete, unique view. On the other hand, the "strict" uses of the traditional algorithms are much more intuitive and allow me to freely work closer to the manner that I think that about model composition is.*"

Finally, we have observed that: (1) the model composition techniques should be more intuitive and flexible to express different types of changes such as addition, removal, modification, and refinement of the model elements; (2) the techniques should represent the conflicts between the input models in more innovative views; and (3) new composition techniques should be a mixture of specification-based and heuristic-based techniques. As a possible follow-up work, we would suggest to design intelligent recommendation systems that help developers to indicate what the best model composition strategy to-be applied, or even recommending how the input models should be restructured to save effort, whereas preventing inconsistencies. Moreover, the future techniques might provide "richer" visualization means to help developers to prevent inconsistencies before model compositions happen. Instead of merely reporting conflicting changes and inconsistencies, the techniques might provide layers and visualization filters of both conflicting changes and inconsistencies. Thus, developers could intuitively identify how the input model elements conflict with each other and how the inconsistencies propagate through the elements of the output composed model.

5 Threats to Validity

Statistical Conclusion Validity. Experimental guidelines were followed to eliminate this threat [2]: (1) the assumptions of the statistical tests (paired t-test and Wilcoxon) were not violated; (2) collected datasets were normally distributed; (3) the homogeneity of the subjects' background was assured; (4) the method of quantification was properly applied; and (5) statistical methods were used. The Kolmogorov-Smirnov and Shapiro-Wilk tests [2] were used to check how likely the collected sample was normally distributed. *Construct Validity*. It concerns the degree to which inferences are warranted from the observed cause and effect operations included in our study to the constructs that these instances might represent. That is, it answers the question: "Are we actually measuring what we think we are measuring?" All variables of this study were quantified based on a previous study [4]. Thus, they were defined and independently validated. Moreover, the concept of effort used in our study is well known in the literature [10]. Therefore, we are sure that the quantification method used is correct, and the quantification was accurately done. *External Validity*. We analyzed whether the causal relationships investigated during this study could be held over variations in people, treatments, composition techniques and the design models. There are reasons to believe the results generalize beyond the three techniques used, but leave it to further work to fully test this.

6 Related Work

Model composition is a very active research field in many research areas such as merging of state charts [7], composition of software product lines [11], aspect-oriented models [12] and mainly UML models. Research initiatives tend to focus on proposing model composition techniques or even creating innovative modeling languages. However, the evaluation of the developers' effort on composing design models using the proposed techniques is still incipient. The lack of quantitative and qualitative indicators on composition effort hinders mainly the understanding of side effects peculiar to certain composition techniques.

Current work has notably aimed at evaluating modeling languages such as UML in terms of some quality attributes such as comprehensibility [14], completeness. Although UML has been adopted, in fact, as the industry standard modeling language, it is just a point of investigation in empirical studies considering model composition. In general, most of the research on the interplay of effort and composition techniques rest on subjective assessment criteria [5]. Even worse, this leads to dependence on experts who have built up an arsenal of mentally-held indicators to analyze the growing complexity of models and then evaluate the effort on composing them [4]. Consequently, the truth is that developers ultimately rely on feedback from experts to determine "how good" the input models and their compositions are. There are many examples in the literature of composition techniques such as MATA [7], Epsilon [15], and IBM RSA [16]. But, they will only be useful if the quality of the output composed models (e.g. correctness) is assured, and the composition effort required is low. Unfortunately, these approaches do not offer any insight or empirical evidence about the effort required to compose design models. As a matter of fact, the current literature about the composition technique points out the absence of empirical studies and does highlight the importance of empirical evidence [5,7,8,12].

According to [5], the state of the practice in assessing model quality provides evidence that modeling is still in the craftsmanship era and when we assess model composition, this problem is accentuated. More specifically, to the best of our knowledge, our results are the first to empirically investigate the topics of the research questions in a controlled way and systematic by using specification-based and heuristic-based techniques.

7 Concluding Remarks and Future Work

This paper can be seen as a first step to systematically assess the trade-off between the specification-based and heuristic-based techniques in terms of effort and correctness. The results of this first controlled experiment suggested that the specification-based techniques neither reduce the developers' effort nor guarantee the higher number of correctly composed models. Even worse, the traditional composition algorithms outnumbered the specification-based technique, to some extent.

However, further empirical studies are still required to investigate if our results can be confirmed (or not) in other contexts, considering other design models, encompassing different evolution scenarios and evaluating other composition techniques. Although the techniques investigated are robust and representative, and there are reasons to believe the results will possibly generalize to other similar scenarios, we do not claim this generalization beyond these techniques, and their use applied to the design models, in particular, class diagrams. Finally, we expect that our findings can be used to motivate other studies.

References

1. Wohlin, et al.: Experimentation in Software Engineering: an Introduction. Kluwer Academic Publishers, Norwell (2000)
2. Devore, J., et al.: Applied Statistics for Engineers and Scientists. Duxbury (1999)
3. Basili, V., Caldiera, G., Rombach, H.: The Goal Question Metric Paradigm. In: Encyclopedia of Software Engineering, vol. 2, pp. 528–532. John Wiley and Sons (1994)
4. Farias, K., Garcia, A., Whittle, J.: Assessing the Impact of Aspects on Model Composition Effort. In: AOSD 2012, Saint Malo, France, pp. 73–84 (2010)
5. France, R., Rumpe, B.: Model-Driven Development of Complex Software: A Research Roadmap. In: Future of Software Engineering at ICSE 2007, pp. 37–54 (2007)
6. Unified Modeling Language: Infrastructure, Object Management Group (February 2010)
7. Whittle, J., Jayaraman, P.: Synthesizing Hierarchical State Machines from Expressive Scenario Descriptions. ACM TOSEM 19(3) (January 2010)
8. Mens, T.: A State-of-the-Art Survey on Software Merging. IEEE Trans. on Soft. Engineering 28(5), 449–462 (2002)
9. Clarke, S.: Composition of Object-Oriented Software Design Models, PhD thesis, Dublin City University (2001)
10. Jørgensen, M.: Practical Guidelines for Expert-Judgment-Based Software Effort Estimation. IEEE Software, 57–63 (May 2005)
11. Thaker, S., Batory, D., Kitchin, D., Cook, W.: Safe Composition of Product Lines. In: 6th GPCE, Salzburg, Austria, pp. 95–104 (2007)
12. Klein, J., Hélouët, L., Jézéquel, J.: Semantic-based Weaving of Scenarios. In: 5th AOSD 2006, Bonn, Germany, pp. 27–38 (March 2006)
13. Dingel, J., Diskin, Z., Zito, A.: Understanding and Improving UML Package Merge. Journal of Soft. and Syst. Modeling 7(4), 443–467 (2008)
14. Lange, C., Chaudron, M.: Effects of Defects in UML Models – An Experimental Investigation. In: ICSE 2006, China, pp. 401–410 (2006)
15. Epsilon (2011), http://www.eclipse.org/gmt/epsilon/
16. IBM RSA (2011), http://www.ibm.com/developerworks/rational/products/rsa/
17. Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design: The Theme Approach. Addison-Wesley, Upper Saddle River (2005)
18. Asklund, U.: Identifying Conflicts during Structural Merge. In: Proc. Nordic Workshop Programming Environment Research, pp. 231–242 (1994)
19. Evaluating the Effort of Composing Design Models: A Controlled Experiment (2012), http://www.les.inf.puc-rio.br/opus/models12-app

Transition to Model-Driven Engineering

What Is Revolutionary, What Remains the Same?

Jorge Aranda, Daniela Damian, and Arber Borici

University of Victoria
{jaranda,danielad,borici}@uvic.ca

Abstract. A considerable amount of research has been dedicated to bring the vision of model-driven engineering (MDE) to fruition. However, the practical experiences of organizations that transition to MDE are underreported. This paper presents a case study of the organizational consequences experienced by one large organization after transitioning to MDE. We present four findings from our case study. First, MDE brings development closer to the domain experts, but software engineers are still necessary for many tasks. Second, though MDE presents an opportunity to achieve incremental improvements in productivity, the organizational challenges of software development remain unchanged. Third, switching to MDE may disrupt the balance of the organizational structure, creating morale and power problems. Fourth, the cultural and institutional infrastructure of MDE is underdeveloped, and until MDE becomes better established, transitioning organizations need to exert additional adoption efforts. We offer several observations of relevance to researchers and practitioners based on these findings.

1 Introduction

Model-driven engineering (MDE)—the proposal to guide the development of software-intensive systems with model-based abstractions, combining models, process, analysis, and architecture [18,5]—shows much promise [11]. As abstractions, models could be more efficiently created and modified than lines of code, driving down costs. If the abstractions are appropriate, models could also be easier to understand than code, which would result in an increase of clarity and quality.

As the proceedings of this conference over the years have demonstrated, there has been a considerable amount of research dedicated to make the MDE vision happen. However, there is a dearth of reports on its practical successes and limitations [4], and specifically, about the effect that MDE has had on the software development process of the organizations that adhere to it and on their resulting socio-technical structures. This is despite the fact that such feedback would be considerably valuable to practitioners exploring the feasibility of MDE in their settings and to researchers looking for accounts of the real-life performance of the tools they help create.

In this paper, we report on an interview-based empirical case study of MDE and software development activities in two teams at General Motors, the well-known car manufacturer, which has invested a significant effort in transitioning to MDE in its development process. Our case study allowed us to explore several benefits and drawbacks of MDE, and to gain insights into the ways in which coordination dynamics are altered by the introduction of MDE into the development process. We present these findings after the following discussions on previous field studies of MDE and on the methodological details of our own fieldwork.

2 Related Work

Although there has been much research into analyzing the formal aspects of MDE proposals, modelling languages, and model transformation techniques, as well as into evaluating the comprehensibility of several model representations, there are few accounts of what happens when projects actually transition into this engineering approach [25]. The social, organizational, and political implications of a technology as potentially disruptive as MDE are large, yet practitioners and researchers have little information to help guide them on this process.

There are, however, several notable industrial field studies that report on the factors that contribute to the success or failure of MDE adoption in large software organizations. Hutchinson *et al.* [16,17] report on an interview-based study in which they explore the experiences using and adapting to MDE. Among other things, they warn against adopting MDE wholesale, recommending instead a progressive and iterative approach. They also warn against transitioning without proper organizational support or without motivation from the developers.

Similarly, Staron [28] reports on different experiences of two companies transitioning to MDE. One backed out, the other continued. Staron argues that the state of the art (in 2006) in MDE technology did not support an efficient transition, and that the problem was exacerbated if the transitioning company had to maintain a large base of legacy code. This issue was previously reported by MacDonald *et al.* [22], who claim that MDE does not lead to an improvement in efficiency, effectiveness, or productivity—at least not in the context of projects with a large amount of legacy code.

Other studies of adoption of MDE in industrial settings include Cheng *et al.* [6] who report on the adoption of automated analysis of object-oriented design models and their effect on software design quality. UML class diagrams from two large industrial models at different developmental stages are employed in a utility analysis carried with DesignAdvisor. The authors find that the quantity of severe errors increases proportionally with design complexity and that the utility of design patters greatly contributed to lower the number of errors in the models wherein the patterns were used.

Kulkarni and colleagues [20] describe strategies for scaling up MDE at Tata Consulting and describe elements of their MDE infrastructure, as well as their experience of using it to deliver large business applications over a period of 15 years.

Closest to our study, the work of Baker *et al.* [2] reveals the impact of MDE adoption at a large organization, Motorola. However, their findings are mostly about tool and language feasibility to support large-scale development. They do mention "team inexperience" as an issue Motorola experienced in their deployment of MDE. By inexperience they mean lack of a well defined process, missing skill sets, and organizational lack of flexibility.

We note that, in parallel to this paper, Kuhn *et al.* [19] drew from the same pool of participants we interviewed to study complementary and non-overlapping research questions: while they focused on human aspects primarily at the individual level, considering cognitive aspects of using MDE technology, we focused on human aspects focused on the organizational consequences of MDE adoption. Kuhn *et al.* found several forces and points of friction with respect to cognitive issues of MDE technology. They discovered that model diffing should be a key feature of MDE tools, that there is a need for problem-specific expressivity, that there continues to be a need for exploration late in the product development cycle, and that point-to-point traceability is a fundamental need that becomes even more acute under MDE.

The majority of the related work, to the best of our knowledge, is concerned with the adoptability of MDE in industry, whereas our report focuses on the change of dynamics of an organization that has determined to adopt MDE. Our approach to studying MDE is different—we focus not on factors of adoption, but on the organizational *consequences* of adoption. We describe our research questions in the next section and the case study design and findings in the remainder of the paper.

3 Research Questions

To complement the existing empirical evidence about MDE in industry, the focus of our investigation was on the issues related to consequences in the development processes and the socio-technical structures enabled or affected by the introduction of MDE in large organizations. We formulated the following three research questions to guide our data collection and analysis:

RQ1: How does MDE adoption look like in practice in large-scale projects? To what extent does MDE alter the development landscape?

RQ2: How do the coordination dynamics and the division of labour change under a transition to MDE?

RQ3: What issues, beyond those reported previously in the literature, are relevant for organizations considering an MDE transition?

4 Case Study Design and Execution

We performed an empirical study of the organizational and coordination consequences of transitioning to MDE. A data-rich qualitative study was the most appropriate for our investigation, given the contextualized and multivariate nature of the phenomena that we wished to study.

We followed Yin's case study methodology [30]. Specifically, we executed an exploratory single-case case study. In an exploratory case study, as in other qualitative empirical methodologies such as Grounded Theory [13], one begins with a set of research questions and no hypotheses or propositions to test. In contrast with Grounded Theory, the goal of an exploratory study is not to produce a new theory based on the data. Instead, the researcher collects data from a previously under-explored domain with the goal of reporting insights that can be tested as hypotheses in future studies.

We collected our data from a single organization: General Motors (GM). As part of a larger research project on MDE in industry, the first author of this paper, Aranda, visited the offices of GM in Michigan, along with two researchers from the University of British Columbia, who were interested in studying issues of cognition and MDE. These three researchers conducted a total of ten interviews together, each lasting about two hours, with control engineers, software engineers, and managers of two teams (and in two campuses) at GM. Table 1 summarizes basic information from the interviewees. The first half of the interviews consisted of questions pertinent to this paper and was directed by Aranda; the second half, directed by the UBC researchers, was concerned with cognitive issues that will not be reported here.

The interviews were semi-structured. The interview guide is not included in this paper for space reasons, but we have made it available online [1]. All the interviews but one were audio recorded. A single researcher (Aranda) annotated and coded them, and analyzed the interviews and notes guided by the research questions stated above. We looked for robust findings: insights supported by the observations of several interviewees, as opposed to single-source reports. All the findings reported here are robust in this sense.

4.1 Details from the Study Site

For this paper, we studied the people and activities relating to software development, testing, management, and process definition in two product development groups at GM. For a reader not acquainted with modern automobile manufacturing, studying software development at GM may appear odd: this is a well-known automobile manufacturing corporation, not a software company. In truth, GM (as other automobile manufacturers) is now a hardware and software development company, and to an outsider it may be difficult to imagine the extent to which software controls its products. GM cars increasingly rely on software to perform their functionality, and correspondingly, GM increasingly depends on its software development groups. Furthermore, GM transitioned to MDE in the two years previous to our data collection, making the organization a prime candidate for the study of the consequences of MDE adoption. Today, most software at GM is developed in model-driven tools (such as MATLAB's Simulink and IBM Rational Rhapsody). One manager and process designer explained it this way:

Table 1. Interviewees for our study

ID	**Team**	**Position**
P1	Core	Control Engineer (Algorithm Development Engineer)
P2	Core	Control Engineer (Algorithm Development Engineer)
P3	Core	Manager (Software Engineering)
P4	Core	Software Engineer
P5	Core	Software Engineer
P6	Aux	Software Design Lead
P7	Aux	Software Engineer
P8	Aux	Software Engineer
P9	Aux	Software Readiness Engineer (Testing)
P10	N/A	Manager (Process Definition)

> *We made the rule that the model is the code; you want to make a change, you change the model, you don't change the code. And then you just regenerate.* —P10, Manager

However, as we will see, the MDE transition is still being negotiated in terms of tool adoption, process agreement, and role definition.

As we stated above, the interviews span two product development groups. One of them develops one type of core driving features[1] (we will henceforth refer to it as the "core functionality group"), the other develops a subset of auxiliary functionality (we refer to it as the "auxiliary functionality group"). Both groups follow the same high-level software process. It is based on a V development model [23], with a workflow that goes down from requirements definition to implementation and then back up to testing, but it is adapted to account for the rest of the hardware demands of product design. Specifically, the organization places a much greater emphasis on what it refers to as "the Physics" (the equations and other engineering considerations required in the design of automobiles), which need to be embodied by code, and in calibrating and testing the software in the particular hardware in which it will be run.

The core functionality group is collocated at the building level, and it is divided in two main teams, which we will call Team A and Team B. Broadly, Team A workers are in charge of designing the equations and their interactions with other features, while Team B workers are in charge of implementation.

[1] We are not more precise purposefully, to obfuscate some internal details of the GM structure and of the teams we studied. By "one type of core driving features" we mean features that help a car perform its essential transportation functionality.

There are other groups and roles as well, but they are more detached from MDE and from development in general, and will not be considered here.

The auxiliary functionality group is globally distributed: it has engineers in one of the main company campuses in Michigan and a team of offshore engineers in Asia. Although the group also has people in charge of designing the equations and others in charge of their implementation, they are not separated by teams in the way that the core functionality group is. This group also has testers playing a more prominent part in feature development.

5 Findings

In this section we present four findings on transitioning to MDE that we believe should be of interest for researchers and practitioners in the area.

First, MDE succeeds in bringing software development closer to the subject matter experts [25], but an important (if at times menial) subset of software development activities still needs to be performed by people other than the subject matter experts—people with significant software development skills. These software engineers still play an important role under an MDE structure.

Second, the basic processes and challenges of organizational structure and interaction remain unchanged with MDE: software development uses largely the same organizational forms [29] and processes as traditional software development, and it is still difficult to coordinate, to clarify requirements, and to get teams of professionals to deliver high quality software. In other words, though MDE can bring important benefits under some situations, by abstracting away some software development obstacles [11], it presents at best an incremental improvement in software development, in the case we studied.

Third, switching to MDE may disrupt the organizational structure and alter its balance, which creates morale and power problems that transitioning groups should consider.

Fourth, MDE represents a migration to an underpopulated cultural and institutional landscape. The tools, training, and expectations of professionals under MDE are not as well developed and established as those under more traditional software development dynamics. We expect MDE transitions to be generally problematic for this reason, now and until the cultural and institutional infrastructure of MDE becomes better established.

The following subsections expand on each of these findings.

5.1 Bringing Development Closer to the Subject Matter Experts

One of the most ambitious visions of MDE is that subject matter experts (or, in lieu of them, requirements engineers or designers) will be able to model the behaviour they wish their software artifacts to exhibit, using an accessible and appropriate abstraction, most probably represented in diagrammatic form, and that just with the push of a button their code will be auto-generated for them and ready to deploy or use. Such a vision would, of course, bring in huge savings to the software development process in terms of, for instance, efficiency, quality, and clarity [24]. Just as today nobody uses assembly language to program their software if they can avoid it, we will, at some point in the near future, look back to lines of code as an antiquated, needlessly detailed, and cumbersome mechanism to capture the behaviour of software. Of course, this MDE vision does not need to be realized in full to start yielding benefits. If developing software using models is beneficial, a partial application might well bring partial benefits, too.

In practice, in GM to date, MDE has certainly brought development closer to the subject matter experts at work. In some groups within GM, control engineers (the mechanical or electrical engineers in charge of facing the hardware and other physical and design constraints, and of describing and supervising the production of automobile features) can now work with their simulations and, with relative ease, auto-generate the code that will be deployed. This is in contrast to their previous dynamic, in which control engineers would specify their requirements, determine the equations that should be implemented if needed, and communicate them to software engineers, who would be tasked with implementing them in full.

The extent to which this new dynamic is established varies across GM, partly due to GM's flexibility in allowing different groups to transition towards MDE according to their contexts. Both team and personal factors seem to affect the dynamic's variation. At one extreme, for some collaborations between control and software engineers, the control engineer now models all the desired functionality, auto-generates the code, and passes it on to the software engineer to do some more menial work—ensuring that the model adheres to standards, that its integration with other code is handled properly, that its functionality satisfies the description of the work item appropriately, and so on:

> *I prefer to just do it all myself. I do all the algorithm design and [the software engineers] do checks and coding standards. I do the work and [the software engineer] just tracks it.* —P1 (Control Engineer)

At the other extreme, there has been no approximation of development to subject matter experts at all: the control engineer continues to specify requirements and functionality in free text, or even verbally, and the software engineer implements them using a modelling tool:

> *I'm the dullest knife in the drawer when it comes to modelling simulation and coding. I know how the physics work but I depend on these young kids to make it manifest in software. [...] I would like a better separation of responsibilities. I would like to work on requirements interfacing on design, and somebody else create software which is the manifestation of the algorithm and then get back to me and show me what they've done. That's when the system works best.* —P2 (Control Engineer)

In between there are other variations. Some control engineers make their model changes in a mock version of the models, and the software engineers use those mock changes as their blueprint to implement the real changes. Other control engineers only model some component of their features of particular interest, and leave the rest to the software engineers.

Beyond the extent to which some control engineers have successfully engaged in developing software with a modelling language, there are several tasks that will be difficult to bring under their scope. In other words, a software development "middle man" might be still needed in an MDE framework. This is because domain experts are unlikely to have the software development training, nor the time, nor the professional inclination, to involve themselves with implementation issues. An organization developing software at this scale requires modelling conventions and standards, quality controls, hardware-software calibrations, integration conflict resolutions, and involvement in necessarily bureaucratic processes such as change management boards. Some of these issues may be resolved with appropriate tooling, and others may be addressed by managerial mandate, but this does not mean that the solutions are simple, painless, or even feasible in the short term, and transitioning organizations need to account for this.

5.2 Persistence of the Traditional Organizational Forms

The increased closeness of domain experts to software development work brought about by MDE raises the question of the extent to which MDE has revolutionized the software development landscape. For a long time, organizational scientists have observed that the various groups that belong to the same industry tend to follow similar patterns of interaction, to structure themselves in similar ways, and to encounter common problems and challenges [26]. In other words, they have the same *organizational form* [29]. Revolutionary technologies can bring about new organizational forms, with different challenges and strategies [14].

At the outset, it is unclear whether MDE can be one such revolutionary technology. On one hand, MDE could upset the whole communication and coordination structure, bringing many roles into obsolescence, and eliminating the need for time-consuming and inefficient structures. On the other hand, one could construe MDE more like a change in representation (that is, a change from traditional coding to modelling), and it is irrational to expect it to tackle the fundamental problems of software development [11].

We found that the latter is indeed the case at GM: while MDE *does* bring benefits, it cannot be considered a *revolutionary* solution with respect to the organizational challenges, processes, and structures of software development work. As we mention above, MDE brings development closer to subject matter experts, in abstractions that are closer to their domain, but these experts still must overcome the difficulties of defining, negotiating, and clarifying their activities [7,8], and of coordinating with other professionals when their work outputs diverge from expectations in the real world. In fact, coordination, which has been increasingly recognized as a central problem in software development [15], seems to be as challenging for GM under MDE as for other large organizations using traditional development approaches. It is still hard to coordinate, especially with remote sites that may be missing contextual information, and with which communication is necessarily slower, as reported in previous studies of requirements engineering in global software teams [9]. Engineers still need to clarify requirements with each other. Parallel streams of work mean that several people tweaking the same pieces of code brings out conflicts.

In other words, what we found striking was how little difference there was between GM and other organizations' forms, despite the fact that GM now adheres to an approach that is in some respects radically different from the traditional one. The tools and the language are different, but the organizational structure and challenges remain largely the same. While MDE may provide productivity gains, judging from this case it does not seem to lead to a radically different work arrangement for the software industry.

We found two important caveats to this observation, however. We will deal with them in the coming sections.

5.3 Coordination and Division of Labour

For GM, the switch to MDE caused an interesting organizational disruption. As stated above, before the introduction of MDE, the core functionality group that we studied had settled into a bipartite division of labour, organized by domain: one team (Team A) tackled "the Physics" involved in designing automobile features, and a second team (Team B) addressed the software implementation. One person from each team coupled with each other to work on a feature together. The Team A engineers (the Control Engineers) worked on the equations necessary for the appropriate functioning of their features, as well as on the hardware constraints and the interdependencies with other features. The Team B engineers (the Software Engineers), in turn, focused on "hand-coding" the equations and constraints from their partners into software. They also unit tested their own code, ensured it adhered to their conventions, and were responsible for any changes made to it. In conventional terms, and simplifying the collaboration, the Team A engineer in each couple worked on the analysis and design of a feature, while the Team B engineer worked on its development.

Eventually, MDE tools and processes were introduced to the group. The tools allowed engineers to model the behaviour of their systems graphically, and to auto-generate code that implemented the desired behaviour. The auto-generated code rarely needed to be rewritten. Furthermore, the models could be initially tested in a simulated environment, relaxing GM's dependence on physical tests with real hardware.

This introduction of MDE tools, however, brought a disruption in the work arrangement we described above. The Team A engineer now prepared "the Physics" as a computer model, and auto-generated the code that implements it, but the Team B engineer was still necessary: there were aspects of software development that Team A engineers did not have the training, the time, nor the inclination to tackle: issues such as coding conventions, unit testing, versioning, and code dependencies. The fact that these were largely clerical issues did not escape members of either team. For instance, according to one Team A engineer:

> *There's a software engineer who manages that [module]. We're supposed to be working with them to design stuff, but they really are acting like*

> *bookkeepers and code checkers rather than designers, and they don't seem to be interested in what we're doing. (...) probably because their job is boring.* —P1 (Control Engineer)

In effect, the balance in the partnership was lost: Team A increased its conceptual power and dismissed the work of Team B, whereas Team B exercised its remaining structural and technical power [21] as a gateway through which all changes must flow. The collaboration, according to parties on both sides, was at an all-time low.

The group attempted to solve this problem. Its main strategy was to loosen the division of labour between both teams, so that Team A engineers would become more involved with the implementation issues of their models, and Team B engineers would become competent about the mechanical domain and could contribute to the analysis and design of their features. In partnerships with the right motivation and mentorship efforts, this approach worked. For instance, according to one Team B engineer:

> *Sometimes I do development, that's one of the things we tried to do when we switched to models, to have [Team A] and [Team B] to take on development. Every now and then we help [Team A], others we do the changes ourselves. I've done a couple of those.* —P4 (Software Engineer)

In other partnerships, however, the pattern we observed (which we named the *Modeller-Clerk* pattern) remains. At the time of our interviews, GM continued to negotiate the transition to its new normality.

5.4 Cultural and Institutional Problems

Cultural and institutional forces exert a powerful influence over the activities and decisions of a software organization, though our community tends to overlook them. Many problems of adoption, adaptation, morale, and process redefinition in software development can be traced to institutionalization issues. In order to make our point, however, we first cover the basics of institutionalization theory, as we find it is not well known in our field.

Briefly, and simplifying institutionalization theory [10], in our day-to-day activities we are faced with numerous decisions to make and data to ponder. In general, an efficient strategy to deal with this overwhelming abundance of decisions and data is to repeat the behaviours that worked in the past to deal with similar situations [27]. We choose once (which text editor to use, how to respond to a bug report, when to hold team meetings), and, as long as our choice was at least somewhat successful, every time life throws a similar scenario at us, we respond in the same way.

An essential point of institutionalization theory is that this habit-forming tendency we have extends naturally to our peer interactions. Once a team has satisfactorily dealt with a situation, it is likely to replicate the patterns of interaction that led to its resolution, and the more like situations it tackles, the more the pattern is reinforced: each team member learns what to expect from her colleagues, and would be surprised, or perhaps angered, if a teammate deviates from the established behaviour [3,12]. The team pattern is recognized as *the* way to deal with the situation—it is institutionalized.

Furthermore, these patterns extend beyond single teams and into the wider society. A team requires its recruits to have certain skills in order to perform its patterns appropriately, and it pushes educational institutions to train their students to meet its criteria. (Alternatively, educational institutions can push other organizations to accept *their* definition of what counts as valuable skills to have, and the other organizations shift their patterns accordingly.) In time, most aspects of the domain become institutionalized: what counts as knowledge and accepted wisdom, what are proper career paths, what are the tools of the trade, what are the roles people can take, and which ways do they interact with their peers. As long as a kind of situation arises with frequency, the community will coalesce into a set of institutional forms to deal with it.

One can easily see, after these considerations, that any kind of significant organizational change is difficult. It entails the breakdown of many small negotiated successes, it casts the organization's shared understanding into disarray, and it can even cause strong emotional reactions from people forced to reexamine and rebuild many aspects of their professional lives.

We found that MDE adoption falls under the kind of significant organizational change that causes these kinds of reactions at GM. In many respects, MDE is still a pioneering strategy, and the mainstream of the software development field is not familiar with the tools and the practices required for it.

To begin with, tooling capabilities and modelling conventions are still not comparable to their traditional-coding counterparts:

> *One complaint was that changes took very long. Some people would say, if it takes me one hour to make a change in C code, it will take me four hours in MATLAB.* —P4 (Software Engineer)

But the organizational toll may be a greater problem. For GM, hiring software engineers from the mainstream would be difficult; it chooses to hire mostly electrical or computer engineers, which not only have a better understanding of the domain, but may be more familiar with modelling tools such as Simulink. However, these engineers may be less attuned to software development practices and habits. Even then, many (if not most) of GM's engineers have had to learn MDE on the job.

Other engineers have been building automotive software for years, or even decades, and among them resistance to MDE may be greater. They are used to coding in C, and they are effective and efficient with it. The new approach asks them to move towards tools they see as inferior, rightly or not. They find that both small and large code changes take a longer time than they should. They are not used to some of the abstractions embodied by their new tools. In short, despite their wealth of expertise, both in the automotive and in the software domains, they suddenly feel incompetent. The new cultural and institutional infrastructure is not yet set up to exploit this wealth of expertise properly:

> *It was clear that for people who'd been here for a while... it was all mature, things worked, all that. [With] the MATLAB environment, we needed to redesign the process [...]. For some new people, they were OK with that, others are still struggling.* —P3 (Manager)

Incidentally, the new approach also blocks the old one from functioning. Once a module transitions to MDE, the auto-generated C code, by all accounts, is terrible to work with directly. Furthermore, since the organization has committed to a model *driven* strategy, its tools increasingly enforce a process that requires model-based activity to move items towards their resolution.

It is unclear what an organization at this stage can do other than mitigation. GM is aware that rolling back to traditional coding appears to be an expensive solution—it would require the translation of large swaths of models into human-*readable* C code, and the re-training of new engineers who are now used to working primarily with models. A more likely scenario is to carry forward, absorbing the costs of institutionalization gradually, in the hope that the new approach will yield greater benefits than the old.

This is, of course, a consequence of being at "the bleeding edge" of innovation. These are not intractable problems. If MDE catches on, most of these issues (tooling, process, conventions, education) will dissipate. Our study merely points out that, to date, these cultural and institutional issues still stand forcefully in the way of a smooth transition towards MDE.

6 Discussion

Any technological approach that touches so many aspects of a socio-technical structure as MDE does for software development deserves a close scrutiny of its consequences. In the case of GM's adoption of MDE, we found a set of organizational benefits and problems that we think are of relevance for other organizations considering an MDE transition and to the research community.

At this point, it is useful to revisit our original research questions and provide some answers based on our exploratory study.

RQ1: *How does MDE adoption look like in practice in large-scale projects? To what extent does MDE alter the development landscape?*

Though technically MDE presents several interesting innovations and challenges, judging from our data, large-scale projects that adopt MDE look mostly similar to more traditional large-scale projects. The mainstream structure of software development teams and processes remains in place. That is, MDE brought a shift in emphasis to the activities of GM professionals, a shift that allows Control Engineers to design automotive software features in a tool and in a manner convergent with the product that will eventually be released, but it did not eliminate the need for the software development structure (of software engineers, testers, maintainers, and the like) that supports traditional software organizations.

Furthermore, we found that the organization we studied, under MDE, still struggles with the same issues that most traditional software organizations struggle with. Under the right circumstances, MDE may help relieve some problems in software development, but it leaves its basic organizational form unchanged.

RQ2: *How do the coordination dynamics and the division of labour change under a transition to MDE?*

Some inter-team coordination dynamics change after a transition to MDE. Models are more used for communication and coordination, although there is still significant reliance on face-to-face communication to clear potential misunderstandings, and on natural language to explain several aspects of architecture and functionality.

MDE, however, may be expected to alter the current division of labour in a transitioning organization, and the resulting imbalance may be hard to negotiate, at least temporarily.

RQ3: *What issues, beyond those reported previously in the literature, are relevant for organizations considering an MDE transition?*

The division of labour imbalance issue should be relevant for organizations considering transitioning to MDE. Another issue we discovered is the extent to which the transitioning organization will find frictions as it migrates and attempts to re-establish itself in an underdeveloped cultural and institutional setting. Previous studies [2] have pointed out that one of the most important challenges for organizations adopting MDE is that change is difficult. Our results go a step forward, by helping to explain in what sense is organizational change difficult, and for which reasons (an underdeveloped infrastructure for the new institutions, and a disruption with the old and well-established institutions).

6.1 Threats to Validity

As with any empirical results, one should exercise caution in the interpretation of our findings. We discuss three threats to their validity that we were concerned with as we performed our data collection and analysis: their generalizability, the number of interviews we performed, and the fact that we only performed interviews after the transition to MDE was well underway.

First, there is the natural question of whether one can generalize from a single case study (or indeed from a single study of any kind) of a single organization to the whole MDE field. The short answer is that one cannot. In a setting as complex as that of organizational and technological change, there are too many confounding factors, qualifications, and provisions. The experiences of other transitioning organizations may be quite different from those we observed at GM. Nevertheless, we believe there is much value in reports such as these, for two reasons: first, because they provide rich information when there was little or none, and second, because they begin the scientist's task of uncovering the *reasons* for which the probable consequences of MDE adoption will be experienced. For instance, to say that "organizational change is hard" is trivial, but to uncover *in what ways* it is hard for MDE, what are the likely *causes* for the hardships, and *when* are they likely to dissipate, is valuable. Our report does not provide certain answers to these questions, but we believe it helps reach them despite its necessarily limited scope.

Second, we performed only ten interviews, and these were constrained to only two teams within the organization. Of course, a greater number of interviews and of teams would be preferable. We were fortunate, however, to be able to interview professionals in a wide variety of positions, which provided us with a multiple number of views on the inner workings of the organization. The interviews themselves, as can be inferred from the findings we reported above, were frank and wide-ranging. Although further studies at this or other organizations should help us improve the confidence on our findings, we think they are appropriate for the current, exploratory stage of our research.

Finally, we were only able to perform interviews after the transition to MDE was a "done thing". We did not interview professionals before the transition began. We had to resort to their recollections of how things were different before MDE, which may be biased by current events. We had to come to terms with this necessary evil, given the otherwise unprecedented opportunity we had in having access to a transitioning MDE organization and in being able to ask its members both wide and sensitive questions on their work life.

6.2 Implications

For Researchers: Our findings point towards several research directions that we believe are worth pursuing. First, they show that studying the practical consequences of MDE is an important endeavour, and they call for further studies to build a richer experience bank and improve our generalization power.

Second, the disruption of organizational balance is worth exploring. The extent to which it is unique to GM, or to which it can be avoided with careful organizational planning and design is unclear. The direct and indirect costs that the transitioning organization needs to absorb to deal with it are similarly unclear.

Third, in the case that improvements brought about by MDE are not radical (as our GM data suggests), there is the question of the conditions under which a transition to MDE is advisable, and the realistic advantages that the transitioning organization can be expected to reap from its efforts. We note that these viability questions are severely under explored in the MDE literature. Further research should help provide answers to these questions.

For Practitioners: Naturally, a transition to MDE will have costs and benefits—the questions, as usual, are what costs and benefits are there, and under which conditions the latter overcome the former. We did not delve into the technical costs and benefits of the model representation (though our colleagues from UBC have), but organizationally speaking, we found that introducing MDE does, in effect, bring development closer to subject matter experts, but does not eliminate the need for work tasks that domain experts are not qualified or expected to perform. If the GM experience is indicative of more general cases, it is unlikely that there will be savings from staff reductions or from a leaner organizational structure.

Indeed, we found that the software development process remains largely indistinguishable from that of traditional software development, with its same challenges and issues, though organizationally speaking the introduction of MDE may alter the balance in undesirable ways.

Finally, while MDE has matured over the years to the point where it can sustain the development of products of critical importance and of high quality, as GM's automotive software needs to be, its institutional infrastructure is still underdeveloped, and transitioning practitioners will find that, both technically and organizationally, many things they took for granted need to be built again.

7 Conclusion

As the viability of MDE continues to increase, we will need more field studies that report on the consequences of their adoption by leading practitioners. Only by identifying the current strengths and weaknesses of MDE can researchers expect to refine the MDE approach to fulfill its vision.

By pointing out several positive and negative organizational consequences of MDE adoption, this paper is a step in that direction. MDE brings development closer to subject-matter experts, and although it does not alter the organizational form of software development groups, it shifts the balance of power and the division of labour within them in ways that may be conflictive, at least temporarily. Furthermore, adoption leads to institutional and cultural frictions that will not be resolved in the short term. We argue that these organizational consequences are far from negligible, and that transitioning organizations and researchers need to consider them in their efforts to advance this domain.

Acknowledgements. We are grateful to our interviewees for their time and information, and to Joe D'Ambrosio and Shige Wang at General Motors for their dedicated coordination to help us execute this study. We thank Adrian Kuhn and C. Albert Thompson for their collaboration during our joint visit to GM. Funding for this study was provided by the NECSIS Network.

References

1. Aranda, J., Damian, D.: Interview Guide for GM October - November 2011 Visit (2012), `http://home.segal.uvic.ca/~jorge/materials/guide.pdf`
2. Baker, P., Loh, S.C., Weil, F.: Model-Driven Engineering in a Large Industrial Context — Motorola Case Study. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 476–491. Springer, Heidelberg (2005)

3. Berger, P.L., Luckmann, T.: The Social Construction of Reality. Doubleday Anchor (1967)
4. Beydeda, S., Book, M., Gruhn, V.: Model-Driven Software Development. Springer (2005)
5. Bézivin, J.: Model Driven Engineering: An Emerging Technical Space. In: Lämmel, R., Saraiva, J., Visser, J. (eds.) GTTSE 2005. LNCS, vol. 4143, pp. 36–64. Springer, Heidelberg (2006)
6. Cheng, B.H.C., Stephenson, R., Berenbach, B.: Lessons Learned from Automated Analysis of Industrial UML Class Models (An Experience Report) . In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 324–338. Springer, Heidelberg (2005)
7. Cheng, B.H.C., Atlee, J.M.: Research directions in requirements engineering. In: FOSE 2007: Future of Software Engineering (2007)
8. Curtis, B., Krasner, H., Iscoe, N.: A field study of the software design process for large systems. Communications of the ACM 31(11), 1268–1287 (1988)
9. Damian, D., Zowghi, D.: Requirements engineering challenges in multi-site software development organizations. Requirements Engineering Journal 8(3), 149–160 (2003)
10. DiMaggio, P.J., Powell, W.W.: Introduction. In: Powell, W.W., DiMaggio, P.J. (eds.) The New Institutionalism in Organizational Analysis. ch. 1, pp. 1–38. University of Chicago Press (1991)
11. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: FOSE 2007: Future of Software Engineering (2007)
12. Giddens, A.: The Constitution of Society. Polity Press (1984)
13. Glaser, B.: Basics of Grounded Theory Analysis. Sociology Press (1992)
14. Hannan, M.T., Freeman, J.: Organizational Ecology. Harvard (1989)
15. Herbsleb, J.D.: Global software engineering: The future of socio-technical coordination. In: FOSE 2007: Future of Software Engineering (2007)
16. Hutchinson, J., Rouncefield, M., Whittle, J.: Model-driven engineering practices in industry. In: ICSE 2011: Proceedings of the 33rd International Conference on Software Engineering, Honolulu, HI, USA (2011)
17. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of MDE in industry. In: ICSE 2011: Proceedings of the 33rd International Conference on Software Engineering, Honolulu, HI, USA (2011)
18. Caskurlu, B.: Model Driven Engineering. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335, pp. 286–298. Springer, Heidelberg (2002)
19. Kuhn, A., Murphy, G.C., Thompson, C.A.: An Exploratory Study of Forces and Frictions Affecting Large-Scale Model-Driven Development. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, pp. 352–367. Springer, Heidelberg (2012)
20. Kulkarni, V., Reddy, S., Rajbhoj, A.: Scaling Up Model Driven Engineering – Experience and Lessons Learnt. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part II. LNCS, vol. 6395, pp. 331–345. Springer, Heidelberg (2010)
21. Macaulay, L.A.: Requirements Engineering. Springer (1996)
22. MacDonald, A., Russell, D., Atchison, B.: Model-driven development within a legacy system: an industrial experience report. In: Proceedings of the 2005 Australian Software Engineering Conference (2005)
23. Pressman, R.: Software Engineering: A Practitioner's Approach. McGraw-Hill (2004)
24. Rech, J., Bunse, C.: Model-Driven Software Development: Integrating Quality Assurance. Idea Group Inc. (2009)

25. Schmidt, D.C.: Model driven engineering. IEEE Computer 39(2) (2006)
26. Scott, R., Davis, G.F.: Organizations and Organizing: Rational, Natural, and Open System Perspectives. Prentice-Hall (2007)
27. Scott, W.R.: Institutions and Organizations: Ideas and Interests. Sage (2008)
28. Staron, M.: Adopting Model Driven Software Development in Industry – A Case Study at Two Companies. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 57–72. Springer, Heidelberg (2006)
29. Stinchcombe, A.L.: Social structure and organizations. In: March, J.G. (ed.) Handbook of Organizations, pp. 142–193. Rand McNally (1965)
30. Yin, R.K.: Case Study Research: Design and Methods, 3rd edn. Sage (2003)

Towards an Automatic Service Discovery for UML-Based Rich Service Descriptions

Zille Huma[1], Christian Gerth[1], Gregor Engels[1], and Oliver Juwig[2]

[1] Department of Computer Science, University of Paderborn, Germany[⋆]
{zille.huma,gerth,engels}@upb.de
[2] HRS-Hotel Reservation Service, Germany
Oliver.Juwig@hrs.de

Abstract. Service-oriented computing (SOC) promises to solve many issues in the area of distributed software development, e.g. the realization of the *loose coupling pattern* in practice through service discovery and invocation. For this purpose, service descriptions must comprise structural as well as behavioral information of the services otherwise an accurate service discovery is not possible. We addressed this issue in our previous paper and proposed a UML-based *rich service description language* (RSDL) providing comprehensive notations to specify *service requests* and *offers*.

However, the automatic matching of service requests and offers specified in a RSDL for the purpose of service discovery is a complex task, due to multifaceted heterogeneity of the service partners. This heterogeneity includes the use of different underlying ontologies or different levels of granularity in the specification itself resulting in complex mappings between service requests and offers. In this paper, we present an automatic matching mechanism for service requests and offers specified in a RSDL that overcomes the underlying heterogeneity of the service partners.

1 Introduction

Service-oriented computing (SOC) enables reusability of software components through their independent development and deployment as *services*. They can be automatically discovered and consumed on the basis of their exposed service descriptions, which specify services in terms of their operations offering certain functionality and are invoked in a logical sequence.

To enable an automatic and accurate service discovery, services must be described comprehensively in terms of their structure and their behavior, since structurally similar services may have totally different behavior and vice versa. However, current service description standards [22], are either limited to structural aspects, or in the case of semantic web service approaches are either not comprehensive enough [13] or are not widely accepted in practice [19,8] due to their diversion from existing standards. In our earlier work, we addressed these

[⋆] This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

issues and proposed a UML-based *rich service description language* (RSDL) [10] comprising an elaborate set of notations for service description in terms of operation signatures, operation semantics as well as required/allowed operation invocation sequences known as *service protocols* of the service partners.

In this paper, we propose a matching mechanism for *service requests* and *offers* specified in this RSDL to enable an automatic and accurate service discovery in service-oriented computing (SOC). For this purpose, the matching mechanism has to consider different elements in *service requests* and *offers*. Further, the matching mechanism must take into account the multifaceted heterogeneity of the service partners in SOC.

This heterogeneity can lead to situations where service requests and offers are *structurally different* but may be *semantically similar* due to different reasons. For example, due to the use of independent ontologies, the service request and offer may be using different terms and concepts to specify similar functionality. These ontological differences need to be identified and resolved while matching service request and offer. Similarly, due to different understanding of the domain, the service partners may specify their requests/offers at different granularity levels, e.g. functionality specified in terms of a single operation in a service request may be specified by multiple operations invoked sequentially in a service offer and vice versa. This results in complex correspondences, such as 1:n, n:1, and n:m between requested and provided operations in the service requests and offers, which need to be identified by the matching mechanism.

The novelty of our matching mechanism lies in the comprehensiveness compared to existing works as it considers different elements of *service request* and *offer* during matching. Further, we aim at an extensive evaluation of this mechanism as part of the service computing platform being developed at the Collaborative Research Center *On-The-Fly Computing*[1].

The remainder of this paper is structured as follows: In Section 2, we provide a typical SOC scenario from our industrial partner Hotel Reservation Service (HRS)[2]. Along this scenario, we introduce our RSDL [10]. In Section 3, we introduce a matching mechanism for service request and offer in RSDL that overcomes the underlying heterogeneity of the service partners. Section 4 discusses related work and finally, we conclude the paper and give an outlook on future work in Section 5.

2 Scenario

Our industrial partner Hotel Reservation Service (HRS) is a worldwide accommodation booking company that provides a web application[2] to its clients for online hotel booking. A typical booking use case at HRS consists of searching, viewing details of search result and online booking of a hotel room. To support this use case, the HRS application acts as a service requestor requesting the

[1] `http://sfb901.uni-paderborn.de`
[2] `http://www.hrs.com`

services of the partner hotels, which in turn act as service providers enabling HRS to access their room management system through their services.

Figure 1 gives a general overview of such an SOC scenario consisting of the following steps: Firstly, service requestors and providers specify their service requests and offers, respectively conforming to their local independent local ontologies. For instance, in the tourism domain several ontologies exist, e.g. the tourism ontology by the Open Travel Alliance (OTA)[3] or the HarmoNET[4] tourism ontology. Finally, service requests and offers are published and matched for service discovery.

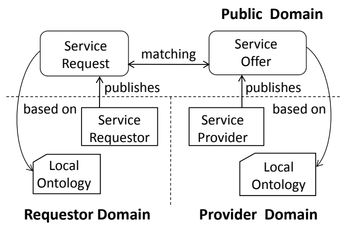


Fig. 1. A general overview of SOC

The current standards for service specification, such as WSDL [22] only specifies *structural* information and does not allow to specify the behavioral information, which may lead to inaccurate service discovery. To address this issue, a variety of approaches such as, WSDL-S [13], Web Ontology Language for services (OWL-S) [19], and WSML[3] by Web Services Modeling Architecture (WSMX)[8], etc. came up with notations for comprehensive description of the requests/offers of service partners. However, some of them, e.g. WSDL-S [13] are not comprehensive enough to cover different aspects of service descriptions like *service protocols*. While others [19,3] are still limited to academia and are not widely accepted in industry due to their diversion from existing standards.

To enable our vision of an automatic and accurate discovery of potential new services, we came up with a proposal for a UML-based RSDL [10]. Due to the use of existing UML notations, it is also easier in practical software engineering projects to adapt to the proposed language. The RSDL comprises the following artifacts for *rich service description* of service partners:

(A) A description of operation signatures using the existing standard WSDL [22],
(B) A semantic description of each operation using UML-based visual contracts (VC) [9,5], and
(C) A specification of service protocols using UML sequence diagrams for service requests and UML state charts for service offers.

After detailed discussion with our industrial partners, we came up with potential service request of HRS based on the proposed RSDL specified in Figure 2. It comprises the specification of operation signatures *(A)* and their semantic description in terms of visual contracts (VC) *(B)*: *checkAvailability()*, *viewDetails()*, *makeReservation()*, and *makePayment()*. A VC specifies the behavior of an operation using two UML object diagrams typed over the concepts contained in the local OTA ontology used by HRS. The object diagrams specifies the state before and after the invocation of an operation, respectively. We refer to these

[3] http://www.opentravel.org
[4] http://www.harmonet.org

object diagrams as *preconditions* of an operation op_i (or $Pre(op_i)$ for short) and *postconditions* ($Post(op_i)$ for short). In addition, the required invocation order of the operations is specified in terms of a UML sequence diagram *(C)*.

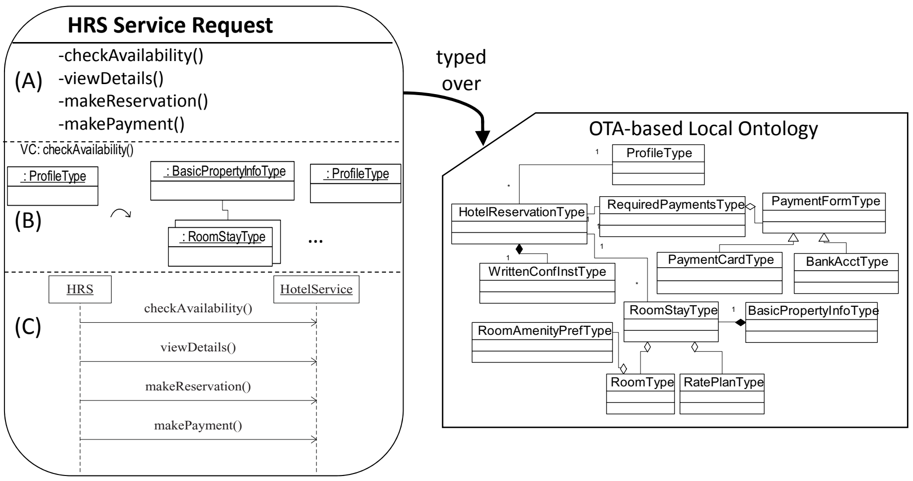


Fig. 2. Service Request of HRS with its OTA-based local ontology

Potential service providers (*Hotel X* and *Hotel Y*) may specify their service offers (e.g. based on the HarmoNET[4] tourism ontology) as shown in Figure 3. The service offer of *Hotel X* comprises the operation signatures *(A1)* and their semantic description *(B1)*: *getAvailableRoom()*, *makeABooking()*, *validateCredentials()*, and *payForBooking()*. Similarly, the offer of *Hotel Y* also consists of the operation signatures *(A2)* with their semantic descriptions *(B2)*: *searchRoom()*, *getRoomDetails()*, and *bookRoom()*. Both service offers specify allowed invocation sequences of their offered operations in terms of UML state charts *(C1 and C2)*.

To enable an automatic service discovery, we propose a mechanism to match *rich* service requests and offers is required, which considers all the aspects of service requests/offers and overcomes the underlying heterogeneity of the service partners.

3 Matching of Rich Service Descriptions

Figure 4 gives an overview of our proposed matching mechanism, which can be divided into two phases: a publishing phase and a searching phase. To publish a service request/offer, the service partners manually map their local ontologies to a global ontology in the first step. Here, we assume that the service partners agree on a common global ontology, which may be provided and maintained by

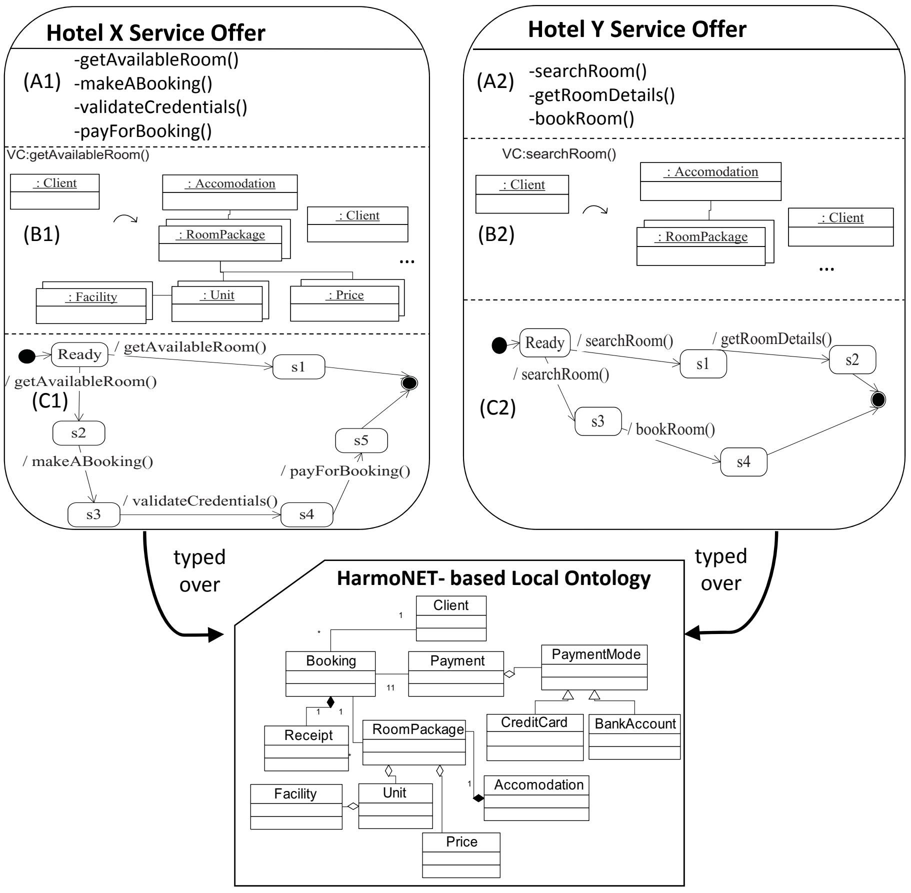


Fig. 3. Two Service Offers of Hotel X and Hotel Y

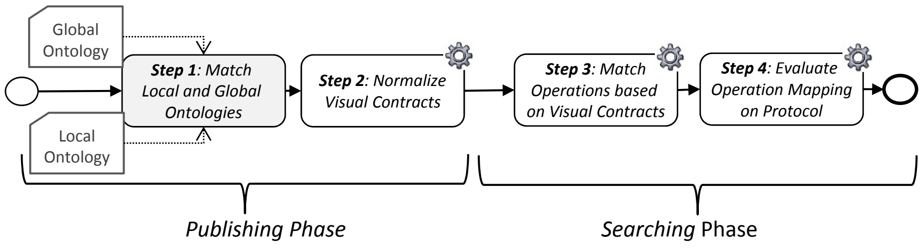

Global Ontology
Local Ontology
Step 1: Match Local and Global Ontologies
Step 2: Normalize Visual Contracts
Step 3: Match Operations based on Visual Contracts
Step 4: Evaluate Operation Mapping on Protocol
Publishing Phase
Searching Phase

Fig. 4. Matching of UML-based Rich Service Descriptions

a service marketplace provider. In Step 2, the VCs in the service request/offer are automatically normalized by translating them to a public representation typed over the global ontology. The searching phase starts with Step 3, where the provided and requested operations are matched based on the *normalized VCs*. In Step 4, the *operation mappings* are evaluated on the provider's service protocol to check whether the provider allows the requested invocation sequence. The individual steps are explained in detail in the following sections.

3.1 Local-Global Ontology Mapping and VC Normalization

The local ontologies of the service partners need to conform to a common representation,i.e., a global ontology before matching service requests and offers. For the following discussion, we select HarmoNET as the global ontology because it covers different aspects of the tourism domain quite comprehensively.

To establish a *local-global ontology mapping*, we rely on a manual 1:1 mapping mechanism because this is not the main focus of this paper. However, there is a variety of existing algorithms and techniques [11,17,16] for automatic local-global ontology mapping, which can be reused in future.

These mappings are established in two stages, i.e., *Class mapping* and *Association mapping*. During *Class mapping*, classes in the local ontology are mapped to classes in the global ontology on the basis of the similarity between class names and their attributes. The similarity between the names of the classes and the attributes can be determined manually or by using tool support for lexical matching [6]. Two classes are mapped, if their names are similar and they have similar attributes. During *Association mapping*, an association in the local ontology is mapped to an association in the global ontology if the *source* and the *target* classes of these associations are already mapped during *Class mapping*.

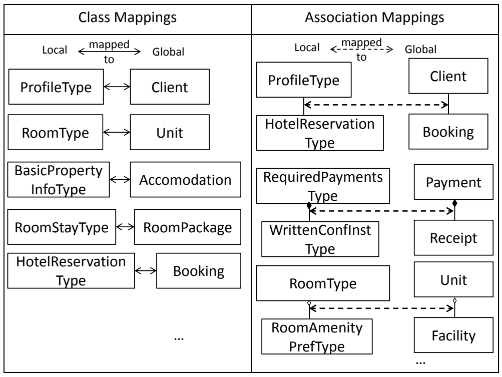


Fig. 5. Excerpt of the Local-Global Ontology Mapping for our Case Study

For our case study, a subset of the local-global ontology mappings is shown in Figure 5. On the basis of these mappings, the VCs contained in service requests/offers can be automatically normalized to a common representation to enable their matching in the next steps. In Figure 6, the VC of *checkAvailability()* by HRS is normalized to a common representation typed over the global ontology HarmoNET.

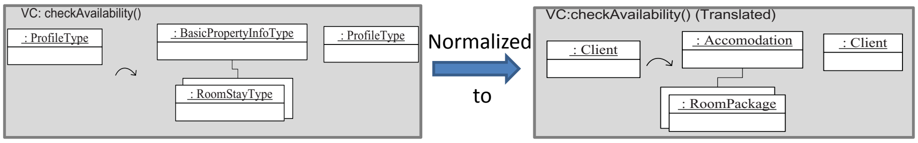


Fig. 6. Normalization of the checkAvailability() Operation of HRS

3.2 Operation Matching

In the second step, the operations in service request and offers are matched based on the normalized VCs using four different matching strategies, i.e. 1:1, 1:n, n:1, and n:m matching, which are discussed in the following sections.

1:1 Operation Matching Strategy. A 1:1 operation matching strategy has already been proposed by [9]. According to this strategy, a provided operation op_p matches a requested operation op_r if the following properties are fulfilled:

P1: op_r fulfills all the preconditions of op_p;
P2: op_p fulfills all the postconditions of op_r, i.e., $Post(op_p)$ *completely satisfies* $Post(op_r)$;

To this extent, $Post(op_p)$ *completely satisfies* $Post(op_r)$, if all the elements that are created, deleted, and preserved by op_r have corresponding elements that are also created, deleted, and preserved by op_p, respectively.

As shown in Figure 7, the VCs of *checkAvailability()* by *HRS* and *searchRoom()* by *Hotel Y* constitute a 1:1 mapping, because *checkAvailability()* satisfies all the preconditions of *searchRoom()*. In addition, $Post(searchRoom())$ completely satisfies $Post(checkAvailability())$, since all the objects added, deleted, or preserved by *checkAvailability()* are also added, deleted, or preserved by *searchRoom()*.

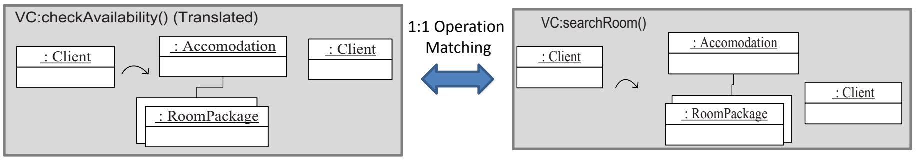


Fig. 7. 1:1 Operation Matching Example

However, in a realistic SOC scenario, there can be complex operation mappings and hence, the 1:1 matching notion does not suffice. For example, two HRS operations *checkAvailability()* and *viewDetails()* can be mapped to one operation *getAvailableRoom()* of *Hotel X*. To identify such complex mappings, more elaborated operation matching strategies are required.

1:n Operation Matching Strategy. In the case, where a 1:1 mapping between a requested and a provided operation cannot be established, because its postconditions are not completely satisfied by the provided operation, a 1:n operation matching may be possible, where multiple provided operations can be invoked in an *allowed* order to fulfill a requested operation.

Our proposed 1:n operation matching strategy maps a requested operation op_{req} to a sequence of provided operations $Seq_{result} = op_{pi} \rightarrow \ldots \rightarrow op_{pn}$. Thereby, Seq_{result} is an allowed sequence of operation invocations specified in the provider's service protocol (see Figure 3 C1 and C2) and every operation in the sequence contributes to fulfill the requested operation. For the 1:n matching strategy, we make the following assumptions:

- As Seq_{result} is a valid invocation sequence in the provider protocol, some earlier operations in the sequence may participate to fulfill the preconditions of an operation in Seq_{result}. Therefore, we assume that the preconditions of an operation $op_{pk} \in Seq_{result}$ are satisfied by the postconditions of the earlier operations in Seq_{result} and the preconditions of op_{req}, i.e. $Pre(op_{pk}) \subseteq Post(op_{pi}) \cup Post(op_{pk-1}) \cup Pre(op_{req})$.
- Further, we assume that every provided operation $op_{pk} \in Seq_{result}$ does not change any requested postconditions in $Post(op_{req})$ that are already satisfied by earlier operations in Seq_{result} and satisfies at least some of the requested postconditions in $Post(op_{req})$ that remain to be satisfied. Hence, $Post(op_{pk})$ *partially satisfies* $Post(op_{req})$, i.e. op_{pk} does not change any elements created, deleted, or preserved by $op_{pi} \rightarrow ... \rightarrow op_{pk-1}$ in Seq_{result} satisfying a postcondition of op_{req} and at least some of the elements that still need to be created, deleted, and preserved by op_{req} have corresponding elements created, deleted, and preserved by op_{pk}, respectively.

In our example, the 1:n matching strategy identifies a matching between the requested operation *makePayment()* of *HRS* and the invocation sequence consisting of the provided operations $validateCredentials() \rightarrow payForBooking()$ of *Hotel X* as shown in Figure 8.

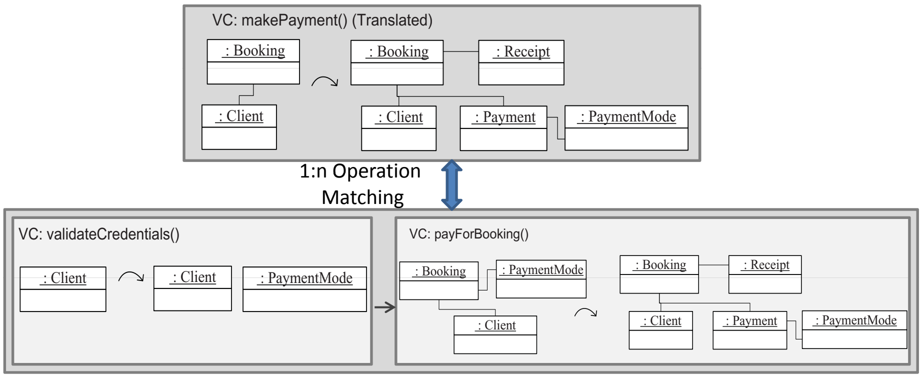


Fig. 8. 1:n Operation Matching Example

Our 1:n matching strategy is specified in Listing 1. As input, it takes a requested operation op_{req} and an invocation sequence Seq_{prov} of the provider service protocol. As output, it returns a sequence of provided operations Seq_{result}, which is a sub-sequence of Seq_{prov} and completely satisfies the postconditions of

op_{req}. The matching strategy is invoked for every invocation sequence specified in the provider service protocol until a mapping is established.

Listing 1: Algorithm for 1:n matching between a requested operation op_{req} and a provider sequence of operations Seq_{prov}

```
Input: Requestor Operation $op_{req}$
Input: An Invocation Sequence in the Provider's Service Protocol $Seq_{prov}$
// $Seq_{prov} = op_{p1} \rightarrow op_{p2} \rightarrow ... \rightarrow op_{pn}$
Output: A matching Sub-sequence $Seq_{result}$ of $Seq_{prov}$
oneToNMatching($op_{req}$, $Seq_{prov}$)
    $op_{start}$=$op_x \in Seq_{prov}$, where ($Post(op_x)$ partially satisfies $Post(op_{req})$)
    $AND\ Pre(op_x) \subseteq Pre(op_{req})$;                                // Step ①
    $Seq_{result} = op_{start}$;
    $op_i = op_{start}$;
    $PreCheck = Pre(op_{req})$;                                          // Step ②
    $PostCheck = Post(op_{start})$;
    while ($PostCheck$ NOT(completely satisfies) $Post(op_{req})$ AND $(i \leq n)$) do
        if $Post(op_i)$ partially satisfies $Post(op_{req})$ then             // Step ③
            if $Pre(op_i) \subseteq PreCheck$ then
                $Seq_{result} = Seq_{result} + op_i$;
                $PreCheck = (PreCheck \cup Post(op_i))$;
                $PostCheck = (PostCheck \cup Post(op_i))$;
        end
        else  $Seq_{result} = null$;
        $STOP$;                                                            // Step ④
        $i++$;
    end
    if ($PostCheck$ NOT(completely satisfies) $Post(op_{req})$ AND $(i > n)$) then
    $Seq_{result} = null$;                                              // Step ⑤
    return $Seq_{result}$;
end
```

The algorithm in Listing 1 works as follows: While iterating a provider's sequence, the first operation op_{pi} that *partially satisfies* the postconditions of op_{req} and whose preconditions are satisfied by op_{req} becomes the first operation of Seq_{result} (Step ①). In the given example, *validateCredentials()* is the first operation in the provider's sequence, whose postconditions *partially satisfies* the postconditions of the *makePayment()*, i.e., *PaymentMode* is added and we assume that *PaymentMode* will not be deleted by any subsequent provided operation in Seq_{result}. The preconditions of *validateCredentials()* are also satisfied by *makePayment(). PreCheck* and *PostCheck* are initiated to check the stepwise satisfaction of the postconditions of op_{req} and the preconditions of the provider operation under consideration (Step ②).

The provider's sequence is iterated until either the postconditions of op_{req} are *completely satisfied* or the provider's sequence ends. Each op_i whose postconditions *partially satisfies* the postconditions of op_{req} and whose preconditions are satisfied by the op_{req} or earlier provided operations is added to Seq_{result} and *PreCheck* and *PostCheck* are updated (Step ③), e.g., after *validateCredentials()*, *payForBooking()* is added to the Seq_{result}. Here, it is worth notifying that preconditions of *payForBooking()* are satisfied not only by *makePayment()* but also by the earlier provider operation in the sequence, i.e. *validateCredentials()*.

The match strategy terminates without a successful 1:n mapping when either an op_i fails to satisfy any postcondition of op_{req} (Step ④) or the end of a provider's sequence is reached and the postcondition of op_{req} are not completely satisfied (Step ⑤). The algorithm stops as soon as a *valid* Seq_{result} is found. In our example, $makePayment()$ is successfully mapped to $validateCredentials() \rightarrow payForBooking()$ (see Figure 8).

n:1 Operation Matching Strategy. There are two potential cases where an n:1 matching strategy is applicable:

(a) When a 1:1 mapping is established between a requested and a provided operation and it might be extended to a n:1 mapping. For example, *checkAvailability()* of *HRS* has 1:1 mapping to *getAvailableRoom()* of *Hotel X*. However, since the postconditions of *viewDetails()* of *HRS* are also completely satisfied by *getAvailableRoom()*, this mapping can be extended to an n:1 mapping.

(b) When a 1:1 mapping is not possible between a requested and provided operation because the postconditions of the provided operation *completely sastisfies* the postconditions of the requested operation but its preconditions are not satisfied by the requested operation.

Our proposed n:1 operation matching strategy maps a sequence of requested operations $Seq_{result} = op_{ri} \rightarrow \ldots \rightarrow op_{rn}$ to a provided operation op_{prov}, where Seq_{result} is a required sequence of operation invocations in the requestor's service protocol (see Figure 2 (C)).

Our n:1 matching strategy is specified in Listing 2. As input, it takes a provided operation op_{prov} and the requested invocation sequence Seq_{req}. As output, it returns Seq_{result}, i.e., a sub-sequence of Seq_{req}, which satisfies the precondition of op_{prov} and the postcondition of op_{prov} completely satisfies the postconditions of all the operations in Seq_{result}. For example, as shown in Figure 9, while matching the operations of *HRS* and *Hotel X*, after a 1:1 mapping between *checkAvailability()* of *HRS* and *getAvailableRoom()* by *Hotel X* is detected, $checkAvailability() \rightarrow viewDetails() \rightarrow makeReservation() \rightarrow makePayment()$ of *HRS* and *getAvailableRoom()* of *Hotel X* are provided as inputs to the n:1 matching strategy.

The algorithm in Listing 2 works as follows: While iteration, the first operation op_x in Seq_{req}, whose postconditions are *completely satisfied* by op_{prov} and that satisfy some (or all) preconditions of the op_{prov} is added to Seq_{result} (Step ①). As shown in Figure 9, *checkAvailability()* becomes the first operation of Seq_{result} because all its postconditions are satisfied by *getAvailableRoom()* and it also satisfies the preconditions of *getAvailableRoom()*. *PreCheck* is initiated to track the *satisfied* preconditions of the op_{prov} (Step ②).

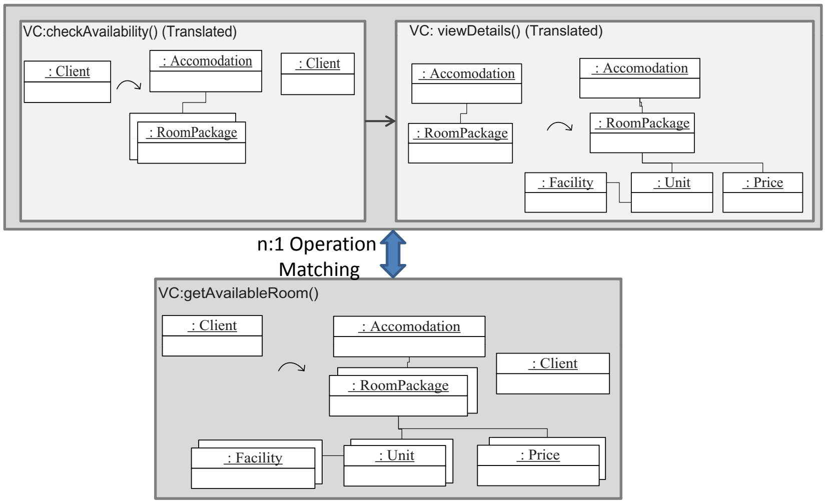


Fig. 9. n:1 Operation Matching Example

Listing 2: Algorithm for n:1 matching between a requestor operation sequence Seq_{req} and a provider operation Op_{prov}

```
Input: Provider Operation $op_{prov}$
Input: An Invocation Sequence in Requestor's Service Protocol $Seq_{req}$
// $Seq_{req} = op_{r1} \rightarrow op_{r2} \rightarrow ... \rightarrow op_{rn}$
Output: A matching Sub-sequence $Seq_{result}$ of $Seq_{req}$
nToOneMatching($op_{prov}, Seq_{req}$)
    $op_{start}$= The first $op_x \in Seq_{req}$, where ($Post(op_{prov})$ completely satisfies
    $Post(op_x)$) $AND$ $Pre(Op_{prov}) \cap Pre(op_x) \neq null$;        // Step ①
    $Seq_{result} = op_{start}$;
    $op_i = op_{start}$;
    $PreCheck = Pre(op_{start})$;        // Step ②
    while ($Post(op_{prov})$ completely satisfies $Post(op_i)$ AND $(i \leq n)$) do
        $Seq_{result} = Seq_{result} + op_i$;        // Step ③
        $PreCheck = (PreCheck \cup Pre(op_i))$;
        $i++$;
    end
    if $Pre(op_{prov}) \nsubseteq PreCheck$ then $Seq_{result} = null$;        // Step ④
    return $Seq_{result}$;
end
```

The requestor sequence Seq_{req} is iterated until either the postconditions of the next requested operation are not satisfied or the Seq_{req} ends.

Every requested operation op_i whose postconditions are *completely satisfied* by the op_{prov} is added to Seq_{result} and *PreCheck* is updated accordingly (Step ③). For instance, after *checkAvailability()*, *viewDetails()* is added to the Seq_{result}

but in the next iteration, the postcondition of *makeReservation()* are not satisfied by *getAvailableRoom()* and the algorithm stops. Till this point, if all the preconditions of op_{prov} are not satisfied then Seq_{result} is not a valid n:1 match for op_{prov}(Step④). In the given case, the n:1 operation matching strategy successfully maps $checkAvailability() \rightarrow viewDetails()$ of HRS to *getAvailableRoom()* of Hotel X (see Figure 9).

Here it is worth mentioning that if the n:1 matching strategy is initiated in the Case (b) as mentioned above, then we prompt the requestor that the match result is *inexact*. This is because the match is successful at design time but at the invocation time, a provided operation can not be successfully invoked until all its preconditions are satisfied. Therefore, the requestor has to decide whether he is able to satisfy all the preconditions of the provider's operation before its invocation.

n:m Operation Matching Strategy. Last but not least in some scenarios n:m mappings between requested and provided operations can occur. We distinguish between *basic* and *complex* n:m mappings. In the former category, existing 1:1, 1:n, and n:1 mappings can be extended to n:m operation mappings. In the latter category, existing 1:n and n:1 mappings overlap and can be combined into n:m mappings.

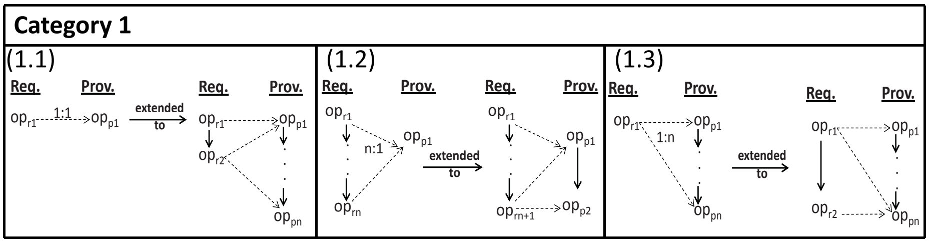


Fig. 10. First Category of n:m Mapping Cases

Figure 10 shows the *basic* n:m mappings. For instance, (1.1) shows a 1:1 mapping established between a requested operation op_{r1} and a provided operation op_{p1}. On further investigation, this mapping can be extended to an n:m mapping because the postconditions of next requested operation, i.e., op_{r2} are also *partially satisfied* by the postconditions of op_{p1}. However, the postconditions of op_{r2} are *completely satisfied* by a sequence of requested operations $op_{p1} \rightarrow \ldots \rightarrow op_{pn}$.

As an example, we consider another potential service partner *Hotel Z* providing the following operations in their allowed invocation sequence: $getAvailableRoom() \rightarrow makeABooking() \rightarrow getReceipt()$. Figure 11 shows the 1:1 mapping between *makeReservation()* of HRS and *makeABooking()* of Hotel Z. The postconditions of the next operation of HRS, i.e., *makePayment()* are also *partially satisfied* by *makeABooking()* of Hotel Z.

However, the postconditions of *makePayment()* are *completely satisfied* by the postconditions of $makeABooking() \rightarrow getReceipt()$. Therefore, the 1:1 mapping can be extended to an n:m mapping.

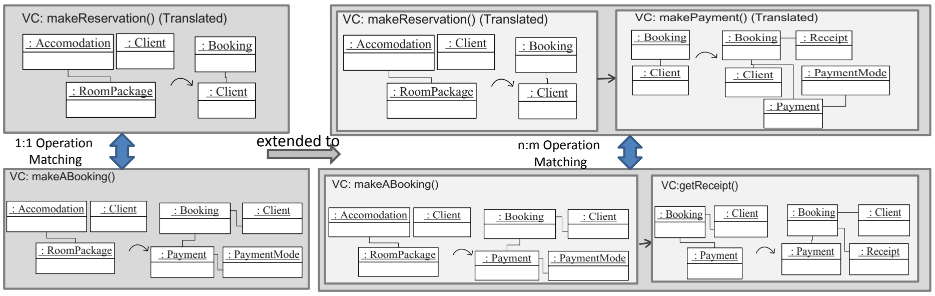


Fig. 11. n:m Operation Matching Example

The *basic* n:m mappings can be established using our existing matching strategies. To deal with (1.1) to (1.3), the existing 1:1 and 1:n operation matching strategies are reused with minor extensions.

As shown in Figure 12, there is a second category of *complex* n:m mapping cases. For example, the Case (2.2) depicts a situation where an existing 1:n mapping is between the requested operation op_{r1} and the provided operations $op_{p1} \rightarrow \ldots \rightarrow op_{pn}$ can be extended to an n:m mapping if some further operations in the service request, i.e., $op_{r2} \rightarrow \ldots \rightarrow op_{rn}$ can also be mapped to op_{pn}. We aim at defining a matching strategy to deal with such complex cases in our future work.

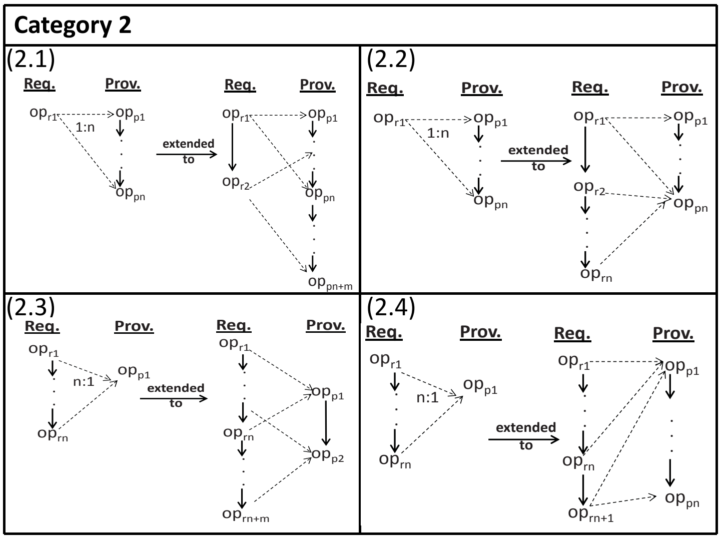
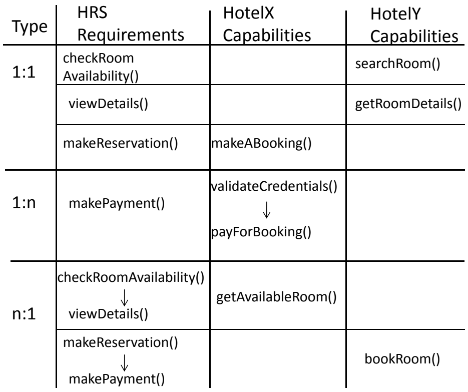


Fig. 12. Second Category of n:m Mapping Cases

Having computed all operation mappings for our given example (see Figure 13), we evaluate whether the matched operations can be invoked in the requested invocation order by inspecting the services protocols of the offered services in Step 4 of our matching mechanism.

3.3 Operation Mappings-Protocol Evaluation

In this step, we evaluate based on the established operation mappings, whether the provided service allows to invoke the operations in the order requested by the service request. For this purpose, traces are extracted from the service protocols contained in the service requests/offers and the operation mappings are evaluated on these traces. For the given example, we will elaborate this mechanism as illustrated in Figure 14.



Type	HRS Requirements	HotelX Capabilities	HotelY Capabilities
1:1	checkRoom Availability()		searchRoom()
	viewDetails()		getRoomDetails()
	makeReservation()	makeABooking()	
1:n	makePayment()	validateCredentials() ↓ payForBooking()	
n:1	checkRoomAvailability() ↓ viewDetails()	getAvailableRoom()	
	makeReservation() ↓ makePayment()		bookRoom()

Fig. 13. Operation Matching Results

First, we consider the traces of the service offer of *Hotel X*. *checkRoomAvailability()* → *viewDetails()* of HRS has a n:1 operation mapping to *getAvailableRoom()*. Therefore, it makes sense to check whether any of the Hotel X traces starts with invocation of *getAvailableRoom()*, which is indeed the case. The next invocation of HRS, i.e., *makeReservation()* has a 1:1 operation mapping to *makeABooking()*. Hotel X trace allows the invocation of *makeABooking()* after *getAvailableRoom()*. Similarly, the next requestor invocation *makePayment()* has a 1:n operation mapping to *validateCredentials()* → *payForBooking()*, which is also allowed in Hotel X trace. Therefore, the requestor's trace can be completely mapped to one of the provider's traces. Hence, the service offer by *Hotel X* completely satisfies the *HRS* service request.

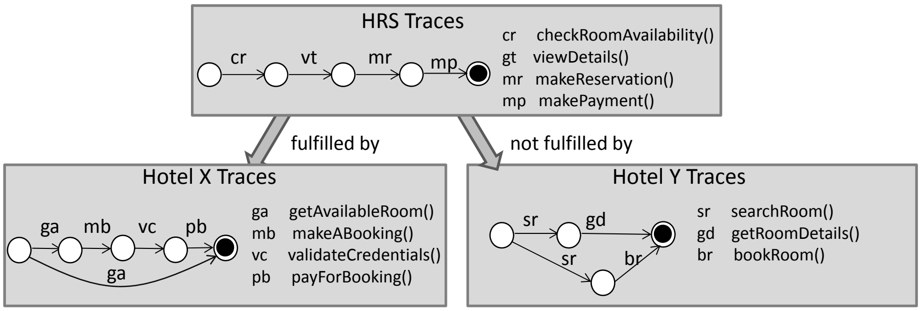


Fig. 14. Service Partners and the Traces in their Protocols

Applying the same mechanism it can be deduced that *Hotel Y* does not satisfy the required trace and hence fails to fulfill *HRS* service request. In this step, it became evident that only one of the provided services, i.e., *Hotel X* is able to completely fulfill the requestor's request.

4 Related Work

We will mainly discuss the related work in the area of service description matching in this section. There are different matching mechanisms [15,12,1] proposed for service requests and offers based on rich service descriptions [9,3,19]. For instance, [15] proposes a matching mechanism for VC-based service descriptions leaving some important issues unsolved, such as, dealing with the underlying heterogeneity, performing n:1 and n:m operation matching between service partners, and service protocol matching. Similarly, service matching approaches like [12,1] based on languages, such as, WSML and OWL-S, respectively, match the services on the basis of the operation structure and behavior in the service request and offer but do not consider the service protocols for this purpose.

Other approaches [14,2,20,4,21] come up with mechanisms for service protocol matching. However, most of these approaches are limited. For example, [20] ignores the underlying heterogeneity of the service partners domain while matching. Similarly, [14,2,21] resolve the heterogeneity but do not take operation semantics into account. Most promising in this regards is [4] that proposes a service protocol matching approach with local-global ontology mapping to overcome heterogeneity of local ontologies. It also deals with n:m matching between operations in the service request and offer. This approach particularly emphasizes on the stateful services and it considers the input/output parameters of the requested and provided operations and the *service states* to match the service protocols. However, a clear definition and structure of corresponding states is missing. In contrast, our proposed approach relies on operation semantics for protocol matching and the notion of corresponding operation semantics is clearly defined in the matching algorithms.

WSMX [8] propose a comprehensive mediator-based mechanism to match *user goals* and *service capabilities*. Multiple mediators are introduced to deal with different types of heterogeneity, e.g., *OOMediator* resolves the ontological differences between the service requestor and the provider. Even though we share the same aims, our approach differs from the WSMX on the fundamental issue of using a de-facto standard like UML [18].

The approach presented in [7] identifies semantic equivalences in structurally different process models by decomposing the models into fragments. This technique could also improve the service protocol matching in particular in case of service compositions.

5 Conclusion and Future Work

We proposed an automatic matching mechanism for service descriptions based on RSDL [10] enabling automatic service discovery despite underlying heterogeneity of the service partners. We have applied the proposed matching mechanism on a real-world case study of our industrial partner HRS. In future, we intend to extend our matching mechanism to consider additional heterogeneity aspects, such as complex mappings between ontologies, complex n:m operation matchings, etc. Another potential area for future work is to automatically define service compositions by combining different service offers to fulfill a service request. Additionally, we will evaluate our approach more extensively through a variety of case studies and an implemented system. This system will be part of a platform for service computing tasks being developed at Collaborative Research Center *On-The-Fly Computing*[1].

References

1. Mukhija, A., Dingwall-Smith, A., Rosenblum, D.S.: Dino: Dynamic and Adaptive Composition of Autonomous Services. White paper, Department of Computer Science, University College London, London (2007)
2. Brogi, A., Corfini, S., Popescu, R.: Semantics-based composition-oriented discovery of Web services. ACM Trans. Internet Technol. 8(4), 19:1–19:39 (2008), `http://doi.acm.org/10.1145/1391949.1391953`
3. de Bruijn, J., Lausen, H., Polleres, A., Fensel, D.: The Web Service Modeling Language WSML: An Overview. In: Sure, Y., Domingue, J. (eds.) ESWC 2006. LNCS, vol. 4011, pp. 590–604. Springer, Heidelberg (2006)
4. Cavallaro, L., Di Nitto, E., Pradella, M.: An Automatic Approach to Enable Replacement of Conversational Services. In: Baresi, L., Chi, C.-H., Suzuki, J. (eds.) ICSOC-ServiceWave 2009. LNCS, vol. 5900, pp. 159–174. Springer, Heidelberg (2009)
5. Engels, G., Güldali, B., Soltenborn, C., Wehrheim, H.: Assuring Consistency of Business Process Models and Web Services Using Visual Contracts. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 17–31. Springer, Heidelberg (2008)
6. European Bioinformatics Institute: Experimental Factor Ontology Tools (2011), `http://www.ebi.ac.uk/efo/tools`
7. Gerth, C., Luckey, M., Küster, J., Engels, G.: Detection of Semantically Equivalent Fragments for Business Process Model Change Management. In: Proceedings of the IEEE 7th International Conference on Services Computing (SCC 2010), pp. 57–64. IEEE Computer Society (2010)
8. Haller, A., Cimpian, E., Mocan, A., Oren, E., Bussler, C.: WSMX - A Semantic Service-Oriented Architecture. In: ICWS 2005, pp. 321–328. IEEE Computer Society (2005)
9. Hausmann, J.H., Heckel, R., Lohmann, M.: Model-based Development of Web Service Descriptions Enabling a Precise Matching Concept. International Journal of Web Services Research 2(2), 67–85 (2005)
10. Huma, Z., Gerth, C., Engels, G., Juwig, O.: UML-based Rich Service Description and Discovery in Heterogeneous Domains. In: Proceedings of the Forum at the CAiSE 2012 Conference on Advanced Information Systems Engineering. CEUR Workshop Proceedings, CEUR-WS.org, vol. 855, pp. 90–97 (2012)
11. Huma, Z., Rehman, M.J.U., Iftikhar, N.: An ontology-based framework for semi-automatic schema integration. J. Comput. Sci. Technol. 20, 788–796 (2005)
12. Klusch, M., Kaufer, F.: WSMO-MX: A hybrid Semantic Web service matchmaker. Web Intelli. and Agent Sys. 7, 23–42 (2009)
13. LSDIS Lab: Web Service Semantics, `http://lsdis.cs.uga.edu/projects/WSDL-S/wsdl-s.pdf`
14. Motahari Nezhad, H.R., Benatallah, B., Martens, A., Curbera, F., Casati, F.: Semi-automated adaptation of service interactions. In: Proceedings of the 16th international Conference on World Wide Web, WWW 2007, pp. 993–1002. ACM, New York (2007), `http://doi.acm.org/10.1145/1242572.1242706`

15. Naeem, M., Heckel, R., Orejas, F., Hermann, F.: Incremental Service Composition based on Partial Matching of Visual Contracts. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 123–138. Springer, Heidelberg (2010)
16. Noy, N.F.: Semantic integration: a survey of ontology-based approaches. SIGMOD Rec. 33, 65–70 (2004)
17. Noy, N.F., Musen, M.A.: The PROMPT suite: interactive tools for ontology merging and mapping. Int. J. Hum. Comput. Stud. 59, 983–1024 (2003)
18. Object Management Group (OMG): Unified Modeling Language (UML) – Superstructure, Version 2.3 (2009), http://www.omg.org/spec/UML/2.3/Infrastructure
19. OWL-S Coalition: OWL-based Web Service Ontology (2006), http://www.ai.sri.com/daml/services/owl-s/1.2/
20. Mateescu, R., Poizat, P., Salaün, G.: Adaptation of Service Protocols Using Process Algebra and On-the-Fly Reduction Techniques. In: Bouguettaya, A., Krueger, I., Margaria, T. (eds.) ICSOC 2008. LNCS, vol. 5364, pp. 84–99. Springer, Heidelberg (2008)
21. Spanoudakis, G., Zisman, A.: Discovering Services during Service-Based System Design Using UML. IEEE Trans. Softw. Eng. 36(3), 371–389 (2010), http://dx.doi.org/10.1109/TSE.2009.88
22. W3C: Web Service Description Language(WSDL) (2007), http://www.w3.org/TR/wsdl20/

A Product Line Modeling and Configuration Methodology to Support Model-Based Testing: An Industrial Case Study

Shaukat Ali[1], Tao Yue[1], Lionel Briand[1,2], and Suneth Walawege[3]

[1] Certus Software V&V Center, Simula Research Laboratory, Norway
[2] SnT Centre, University of Luxembourg, Luxembourg
[3] Cisco Systems Inc., Norway
{shaukat,tao,briand}@simula.no,
suneth_walawege@yahoo.com

Abstract. Product Line Engineering (PLE) is expected to enhance quality and productivity, speed up time-to-market and decrease development effort, through reuse—the key mechanism of PLE. In addition, one can also apply PLE to support systematic testing and more specifically model-based testing (MBT) of product lines—the original motivation behind this work. MBT has shown to be cost-effective in many industry sectors but at the expense of building models of the system under test (SUT). However, the modeling effort to support MBT can significantly be reduced if an adequate product line modeling and configuration methodology is followed, which is the main motivation of this paper. The initial motivation for this work emerged while working with MBT for a Video Conferencing product line at Cisco Systems, Norway. In this paper, we report on our experience in modeling product family models and various types of behavioral variability in the Saturn product line. We focus on behavioral variability in UML state machines since the Video Conferencing Systems (VCSs) exhibit strong state-based behavior and these models are the main drivers for MBT; however, the approach can be also tailored to other UML diagrams. We also provide a mechanism to specify and configure various types of variability using stereotypes and Aspect-Oriented Modeling (AOM). Results of applying our methodology to the Saturn product line modeling and configuration process show that the effort required for modeling and configuring products of the product line family can be significantly reduced.

Keywords: Aspect-Oriented Modeling, Product Line Engineering, Behavioral Variability, Model-based Testing, UML State Machine.

1 Introduction

Product Line Engineering (PLE) has gained significant attention in the recent years in both academia and industry because of its capability to deal with the ever increasing complexity and variation in software product families [1]. Using PLE has shown to be effective for enhancing quality and productivity in product development, and speeding up time-to-market in many organizations such as Boeing, Lucent, and Nokia [2]. We believe that PLE can also potentially help in significantly reducing the amount of modeling effort required for Model-based Testing (MBT). For instance, modeling a Video Conferencing System (VCS) in Cisco Systems Inc, Norway [3], requires modeling 20 subsystems with at least one state machine per subsystem, where many of these subsystems can run concurrently to each other. The modeling effort required for support MBT is often the main concern of industrial testers, especially when there is a lack of familiarity with modeling. Using PLE, we conjecture that the amount of modeling effort required for various products in a product line family can be significantly reduced through reuse, which is the motivation of this work.

While applying MBT on a VCS product [4, 5], we came to the conclusion that we needed a product line modeling and configuration methodology focused on reducing the overall modeling effort of MBT over the entire product line family. Since, as detailed in our discussion of related works (Section 6), no existing PLE methodology addresses in a comprehensive manner all relevant aspects of behavioral variability in UML state machines, we derived and reported in this paper. The methodology has been applied on one Cisco's VCS product line family called Saturn. It is worth mentioning that in this paper we only put our focus on modeling, not on automated derivation of executable test cases from models, which is discussed in [4, 5]. Our methodology mainly focuses on behavioral variability with a focus on UML state machines and this is due to two reasons. First, state machines are the main notation currently used for model-based test case generation [6-8] and are particularly useful in control and communication systems. Second, our industrial case study exhibits strong state-based behavior so that it is natural to provide support for UML state machines. However, in the future, our approach can be extended to other types of UML diagrams by following similar principles.

Our methodology starts with classifying various types of variability that exist in UML class and state machines, in order to model the commonality and variability of a product line family with the goal of supporting MBT. Different approaches (e.g., UML stereotypes) are proposed to specify different variability types. Note that these models need to be built once for the product line family and later on are configured for each product in the product line family. As part of our methodology, we also propose a configuration process (with six steps) to guide a tester to configure the product line models to support MBT, in which different techniques (e.g., OCL constraints, AspectSM [9]) are applied to specify configuration information. This information is in a subsequent step automatically processed by our existing tools (e.g., TRUST tool [5], our OCL solver [10]). For example, our proposed UML 2.0 profile AspectSM, which supports Aspect-Oriented Modeling (AOM), is adopted to specify one type of configuration information. AspectSM allows comprehensive aspect modeling for UML 2.0 state machines and supports modeling crosscutting concerns on all features of UML 2.0 state machines. It also supports all basic features of Aspect-Oriented Software Development (AOSD) [11] such as pointcuts, introduction, joinpoints, and advice. AspectSM has proven to reduce about 95% of the modeling effort in our previous work [9], improved readability [12], reduced modeling errors [13], and improved modeling quality [14]. We applied our methodology to the Saturn product line family and reported how we configure four products of the product line family. Results show that our product line modeling and configuration methodology can significantly reduce modeling effort.

The rest of the paper is organized as follows. Section 2 provides a summary of AspectSM. Section 3 defines our product line modeling methodology. Section 4 describes the product configuration process. In Section 5 we present results and discuss our industrial application. Section 6 relates our work to existing works in the literature and finally Section 7 concludes our work and provides future directions.

2 Aspect State Machines

This section provides an introduction to the AspectSM profile [9] [15], which is used to model aspect state machines. The profile was initially developed for modeling system robustness behavior, which is very common type of crosscutting behavior in many types of systems such as communication and control systems [6-8]. An example of a robustness behavior for a communication system is related to how the system should react, in various states, in the presence of high packet loss. The system should be able to recover lost packets and continue to behave normally in a degraded mode. In the worst case, the system should go back to the most recent state and not simply crash or show inappropriate behavior. In a control system, one needs to model, for example, how the system should react, in various states, when a sensor breaks down. AspectSM allows modeling UML state machine aspects as UML state machines (aspect state machines). Such an approach, relying on a standard and using the target notation as the basis to model the aspects themselves, is expected to make the practical adoption of aspect modeling easier in industrial contexts. In our previous work [9], we thoroughly compared AspectSM with the similar existing AOM profiles. Our findings showed that only AspectSM is exclusively based on standard UML notation and OCL, thus eliminates the need of learning additional non-standard notations or languages, and therefore making it easy to reuse open source and commercial technology. This is highly important in most industrial contexts and strongly affects the adoption of modeling technologies. In addition, it is easy to train people in the industry for standard languages such as UML and the OCL.

Though AspectSM was originally defined to support scalable, model-based, robustness testing, including test case and oracle generation, a fundamental question is whether it is easier to model crosscutting concerns such as robustness with AOM in general, and AspectSM in particular, than simply relying on UML state machines to do it all. In AspectSM, the core functionality of a system is modeled as one or more standard UML state machines (called base state machines). Crosscutting behavior of the system (e.g., robustness behavior) is modeled as aspect state machines using the AspectSM profile. A weaver [9] then automatically weaves aspect state machines into base state machine to obtain a complete model, that can for example be used for testing purposes. The AspectSM profile specifies stereotypes for all features of AOM, in which the concepts of *Aspect*, *Joinpoint*, *Pointcut*, *Advice*, and *Introduction* [9] are the most important ones. In this paper, we report an industrial application of AspectSM to model behavioral variability in UML state machines in a similar fashion as we modeled robustness behavior in [9].

An example of the application of AspectSM is shown in Fig. 1. An aspect state machine modeling crosscutting behavior *EmergencyStop* is shown in Fig. 1. This UML state machine is stereotyped as <<*Aspect*>>, which means that it is an aspect state machine. The <<*Aspect*>> stereotype has two attributes: *name* and *baseStateMachine*, whose values are shown in the note labeled as '1' in Fig. 1. The name attribute contains the name of the aspect (*EmergencyStop* in this example), whereas the *baseStateMachine* attribute holds the name of the base state machine, on which this aspect will be woven, which is *ElevatorControl* in this example. The aspect state machine consists of two states: *SelectedStates* and *ElevatorStopped*. *SelectedStates* is stereotyped as <<*Pointcut*>>, which means that this state selects a subset of states from the base state machine. There are three attributes of <<*Pointcut*>>, whose values are shown in the note labeled as '2' in Fig. 1. The *name* attribute indicates the name of the pointcut and *type* denotes the type of the pointcut, which is *Subset* in this case. In AspectSM, different types of pointcuts can be defined. A complete list of other types of pointcuts is presented in [10]. The third attribute *selectionConstraint* contains a query in OCL on the UML state machine metamodel, which selects all states of the base state machine except *ElevatorAtFloor* and *Idle*. All the model elements stereotyped as <<*Introduction*>> (one state, two transitions) will be newly introduced elements in the base state machine during weaving. This aspect introduces the *ElevatorStopped* state in the base state machine, and selects all states of the base state machines except *ElevatorAtFloor* and *Idle* (via *SelectedStates*) and introduces transitions from them to *ElevatorStopped* with trigger *EmergencyStopButtonPressed*. In addition this aspect introduces transitions from *ElevatorStopped* to all the states selected by *SelectedStates* with trigger *EmergencyStopButtonReleased*.

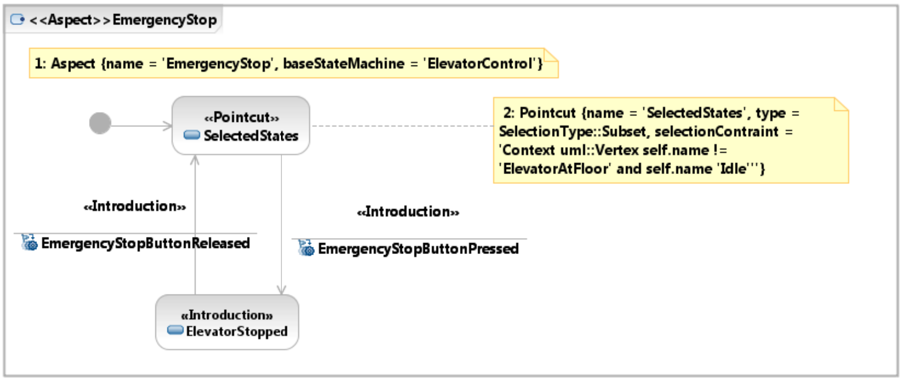

<<Aspect>>EmergencyStop
1: Aspect {name = 'EmergencyStop', baseStateMachine = 'ElevatorControl'}
«Pointcut» SelectedStates
2: Pointcut {name = 'SelectedStates', type = SelectionType::Subset, selectionContraint = 'Context uml::Vertex self.name != 'ElevatorAtFloor' and self.name 'Idle''}
«Introduction»
EmergencyStopButtonReleased
«Introduction»
EmergencyStopButtonPressed
«Introduction» ElevatorStopped

Fig. 1. An aspect state machine for crosscutting behavior *EmergencyStop*

3 Product Line Modeling Methodology

In this section, we describe our product line modeling methodology by showing how it captures the commonality and variability of a product line family as various models for the purpose of supporting MBT. These models should be then configured by, for example, selecting particular behavior models or assigning values to configurable parameters to form uniquely configured products. To explain our methodology with the help of examples, we will use the Saturn product line developed by Cisco. The

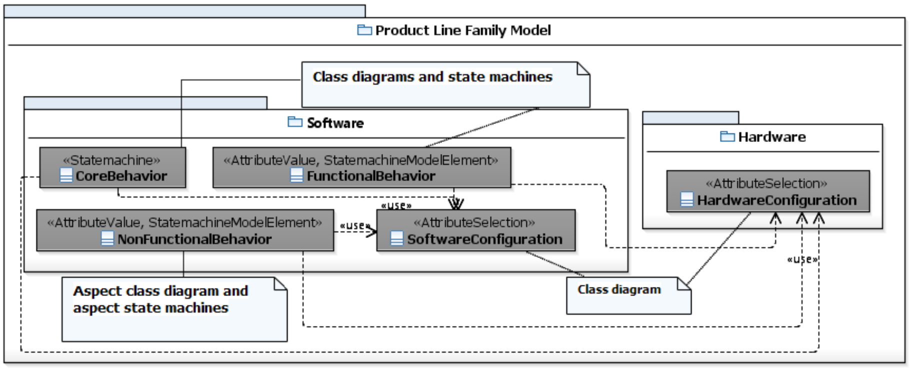
Saturn product line consists of products which differ from each other both in terms of hardware and software. A typical product in Saturn manages the sending and receiving of multimedia streams. Audio and video signals are sent through separate channels. There is also a possibility of transmitting presentations in parallel with audio and video. Presentations can be sent only by one conference participant at a time and all others receive it. Such behavior is central to the operation of a product in the Saturn product line. In Section 3.1, we present the overview of the product line models, followed by the discussion of various types of variability (Section 3.2).

3.1 Product Line Models

According to the needs, we classified product line models into various categories, as shown in the conceptual model (Fig. 2). These models need to be developed once and can then be used to as the basis to configure various products of the product line family for the purpose of supporting MBT. We organize these models into two packages: *Software* and *Hardware*. The *Software* package models are further classified into *CoreBehavior*, *FunctionalBehavior*, *NonFunctionalBehavior*, and *SoftwareConfiguration*. Core behaviors (*CoreBehavior*) are functional behaviors, which differ significantly from one product to another. For example, in our current application, core behaviors are central to the operation of a VCS, which are related to establishing videoconferences. Core behaviors are modeled using class diagrams and state machines. A class diagram specifies state variables of a VCS as class attributes and Application Programming Interface (API) of the VCS as class operations. A state machine specifies the behavior of a VCS and contains model elements such as states (precisely described as state invariants in OCL constraints, based on state variables, which serve as test oracle during MBT) and transitions with triggers (operations in the API defined in the corresponding class diagram).

Similar to core behaviors, functional behaviors are also modeled as class diagrams and state machines, which are configurable for various products. Nonfunctional behaviors (*NonFunctionalBehavior*) (e.g., those related to robustness and security properties) are modeled as configurable aspect class and state machines diagrams using AspectSM. For example, regarding the robustness of a product in Saturn, we are interested in modeling its behavior in the presence of faulty situations in its operating environment such as the network and other VCSs communicating with it. An aspect class diagram models various properties of the environment (e.g., packet loss and jitter in the network) as class attributes using MARTE Non-functional Properties (NFPs) defined in MARTE, a UML profile for modeling and analyzing real-time and embedded systems [16] [17]. An aspect state machine models the behavior of the product in the presence of faulty situations in the environment and how it deals with such situations by transiting to a degraded mode or by executing fail-safe procedures and transiting to the most recent safe state.

Package *SoftwareConfiguration* models various software configurations related to, for example, video conferencing protocols (H323 and SIP), as class attributes in a UML class diagram. These configurations may be used in state invariants or guards on transitions of the (aspect) state machines defined in *CoreBehavior*, *FunctionalBehavior*, and *NonFunctionalBehavior*. *HarwareConfiguration* is similar to *SoftwareConfiguration* except that it models hardware configurations, such as properties of a video output port, as class attributes in a class diagram.


Product Line Family Model
Class diagrams and state machines
Software
Hardware
«Statemachine» CoreBehavior
«AttributeValue, StatemachineModelElement» FunctionalBehavior
«AttributeValue, StatemachineModelElement» NonFunctionalBehavior
«use»
«AttributeSelection» SoftwareConfiguration
«AttributeSelection» HardwareConfiguration
Aspect class diagram and aspect state machines
Class diagram

*Stereotypes applied on the concepts indicate the variability type(s) involved in each package (Section 3.2).

Fig. 2. Conceptual model for product line modeling methodology

3.2 Variability Classification

There are four types of variability in our context, as shown in Fig. 3. We discuss each of them in detail.

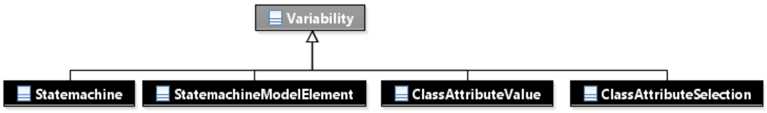


Fig. 3. Classification of variability types

State Machine Variability. This type of variability is related to the behaviors that are specified using significantly different state machines, in terms of model elements, across different products. Therefore, such behaviors should be captured using separate state machines and each individual state machine is considered as a variant and resolving the variation point becomes the selection of proper behavior (i.e., the selection of the state machine specifying the behavior). In the product line modeling phase, we build a repository of such state machine diagrams. Later on during the configuration phase, it is only matter of selecting the appropriate state machine diagrams. For the Saturn product line, there are 10 state machines in the repository corresponding to two core behaviors (Section 3.1) with five state machines per core behavior. For each product, one needs to select one core behavior, i.e., five state machines. In general, we don't expect the number of such core behaviors to be large; otherwise the products may not form a product line.

In our current example, i.e., the Saturn product line, there are two behaviors related to *state machine variability*: multi-way and multi-site. Their state machines differ significantly in terms of model elements such as triggers and guards; therefore we decided to model them separately. The multi-way behavior models the behavior of a VCS that can dial at most to only one Endpoint (EP1) and put the current call on hold to dial to another Endpoint (EP2). The VCS can then switch between EP1 and EP2, but can have only one active call. An excerpt of class diagram modeling APIs as operations and state variables as attributes for multi-way is shown in Fig. 4, whereas an excerpt of state machine modeling this behavior is shown in Fig. 6. Note that just for the sake of keeping the examples simple, the class diagram in Fig. 4 features just one class; however, the actual class diagram for multi-way contains several classes with relationships among them. Another variant is multi-site, which allows a VCS to make calls to three Endpoints simultaneously. The excerpts of the class diagram and state machine modeling such behavior is shown in Fig. 5 and Fig. 7, respectively. Notice that the state machines of both behaviors differ from each other in terms of model elements. For example, multi-site does not have operations related to holding and resuming calls and all transitions related to these operations are not needed in multi-way. Note that, to avoid cluttered models, we have hidden some information from the state machines presented in Fig. 6 and Fig. 7, such as guards and state invariants.

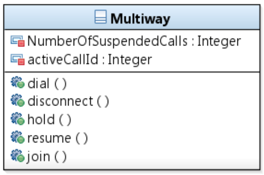


Fig. 4. An excerpt of the class diagram for *Multi-way*

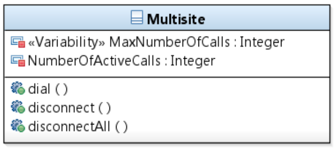

Multisite
«Variability» MaxNumberOfCalls : Integer
NumberOfActiveCalls : Integer
dial ()
disconnect ()
disconnectAll ()

Fig. 5. An excerpt of the class diagram for *Multi-site*

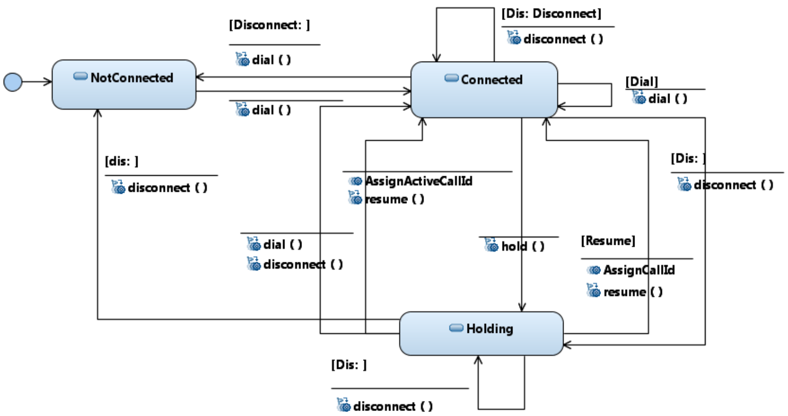


Fig. 6. An excerpt State machine for *Multi-way*

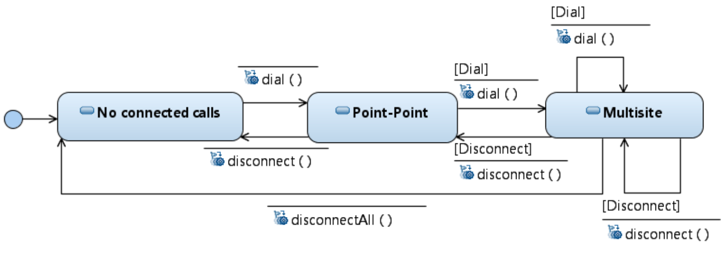


Fig. 7. An excerpt State machine for *Multi-site*

State Machine Model Element Variability. This is the second type of variability where differences in state machines of various products in terms of model elements are not extensive (i.e., a few model elements are variation points), as opposed to the case in *state machine variability*. Variation points in this case are model elements of state machines and aspect state machines, such as transitions and states. Model elements that are variation points are stereotyped as <<*Variability*>>. An example of this type of variability is shown in Fig. 8, where state invariants of all states are stereotyped as <<*Variability*>>, meaning that depending on the product state invariants need to be configured. Notice that in the diagram, just for the sake of explanation, we labeled states with <<*Variability*>> since stereotypes are not displayed visually on state invariants by the modeling tool we used. The example shows that state invariants (written in OCL) for all states in the state machine are variation points and need to be configured for each product. Also note that, to avoid cluttered models, we have hidden some information in the state machine presented in Fig. 8 such as guards and state invariants.

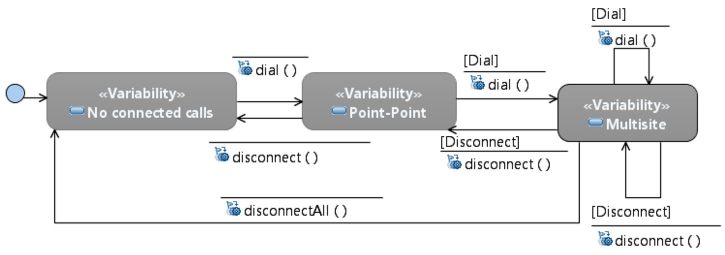


Fig. 8. Example for *model element* variability

Class Attribute Value Variability. For this type of variability, class attributes are configurable parameters. These parameters are used in guards and/or state invariants of a UML state machine. To distinguish the configurable parameters from other attributes of a class, we apply <<*Variability*>> on the configurable parameters of a

context Saturn inv:
self.call.NumberOfActiveCalls > 1 and self.call.NumberOfActiveCalls <= self.multisite.MaxNumberOfCalls

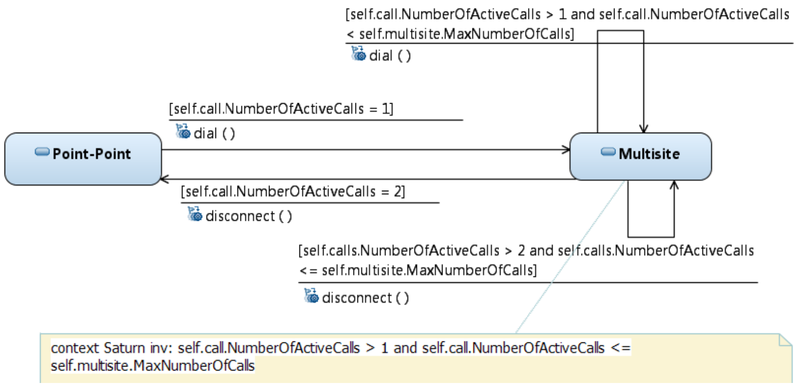

[self.call.NumberOfActiveCalls > 1 and self.call.NumberOfActiveCalls < self.multisite.MaxNumberOfCalls]
dial ()
[self.call.NumberOfActiveCalls = 1]
dial ()
Point-Point
Multisite
[self.call.NumberOfActiveCalls = 2]
disconnect ()
[self.calls.NumberOfActiveCalls > 2 and self.calls.NumberOfActiveCalls <= self.multisite.MaxNumberOfCalls]
disconnect ()
context Saturn inv: self.call.NumberOfActiveCalls > 1 and self.call.NumberOfActiveCalls <= self.multisite.MaxNumberOfCalls

Fig. 9. Example of *class attribute value variability* in a state machine

class. In Fig. 5, *MaxNumberOfCalls* is a configurable parameter labeled with <<*Variability*>>. Fig. 9 shows an example of the application of a configurable parameter on state machines. Configurable parameter *MaxNumberOfCalls* is used in a guard of self-transition to the *Multisite* state. Depending on the type of a product, the maximum number of calls in the product may vary. Similarly, *MaxNumberOfCalls* is also used in the state invariant of the *MultiSite* state, which is as below:

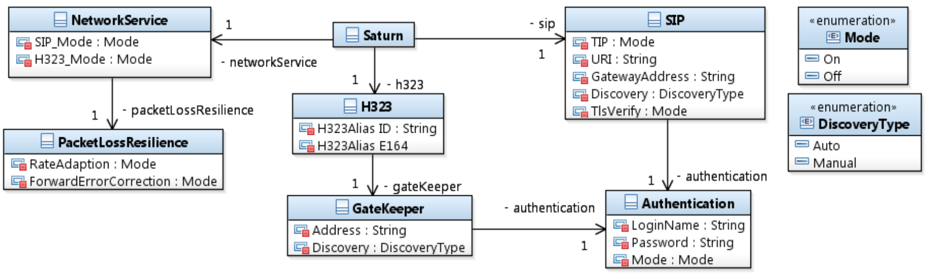


Fig. 10. An excerpt of software configuration for the Saturn product line family

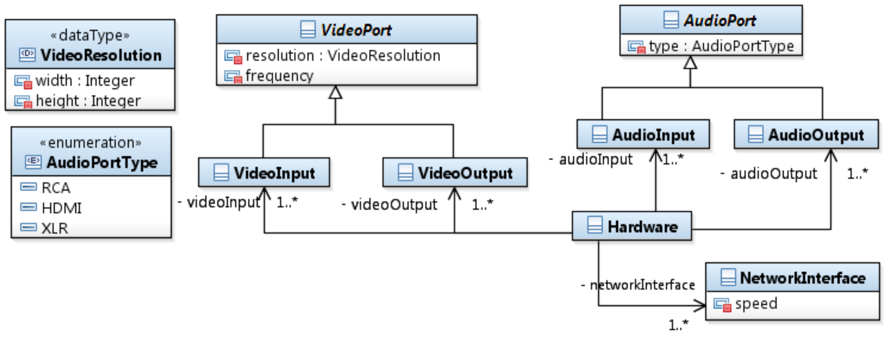


Fig. 11. An excerpt of hardware configuration for the Saturn product line family

Class Attributes Selection Variability. This type of variability is used to configure various state machines modeling functional behaviors based on various software and hardware configuration parameters. In addition, aspect state machines modeling non-functional behaviors, such as robustness behaviors (Section 2), can also be configured based on software and hardware configuration parameters. For each product line, software and hardware configuration parameters are modeled as attributes in class diagrams, as for example, the excerpts of Saturn software and hardware configurations shown in Fig. 10 and Fig. 11, respectively. The process for identifying this type of variability involves the following two steps: selecting a behavior to configure and selecting a subset of software and hardware configuration parameters relevant to the selected behavior. Note that certain software and hardware configuration parameters may have relationships with each other, which may need to be identified and defined. For this type of variability, a product modeler has to write constraints in OCL for product configuration as we will discuss in the next section.

4 Configuration Process for a Product

Before we discuss how a product is configured based on its product line models, which capture the commonalities and variabilities of all the products of the product line family (Section 3), we first present a conceptual model describing the structure of various models required for a configured product to support state-based testing. This conceptual model is presented in Fig. 12 and includes 1) class diagrams and UML state machines, which together specify the core behaviors of the product, 2) a set of class and state machine diagrams specifying the functional behaviors of the product, 3) a set of aspect class and state machine diagrams describing the non-functional behaviors of the product that are crosscutting the functional behaviors, 4), a set of aspect state machines specifying configuration information for resolving *State Machine Model Element Variability* and crosscutting state machines of core behaviors, functional and nonfunctional behaviors, 5) a software configuration class diagram plus constraints and/or aspect state machines, which capture configuration information, and 6) a hardware configuration class diagram plus constraints and/or aspect state machines, which capture configuration information.

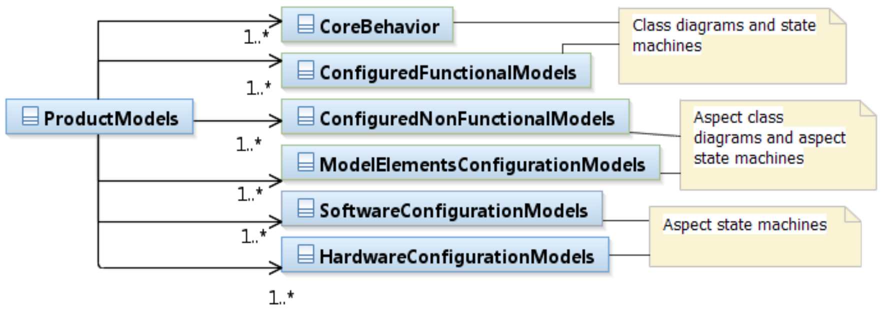

ProductModels
CoreBehavior
ConfiguredFunctionalModels
ConfiguredNonFunctionalModels
ModelElementsConfigurationModels
SoftwareConfigurationModels
HardwareConfigurationModels
1..*
Class diagrams and state machines
Aspect class diagrams and aspect state machines
Aspect state machines

Fig. 12. Conceptual model of the structure of configured product models

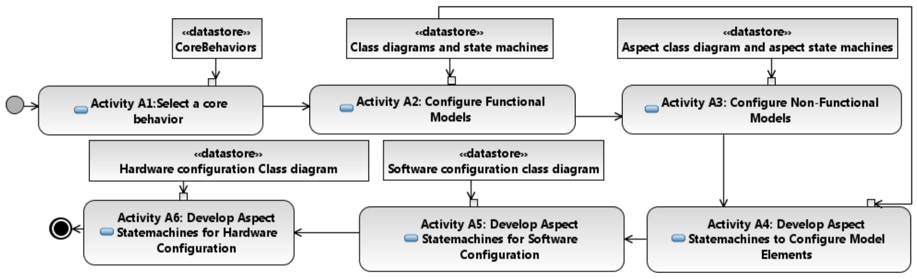

«datastore» CoreBehaviors
«datastore» Class diagrams and state machines
«datastore» Aspect class diagram and aspect state machines
Activity A1:Select a core behavior
Activity A2: Configure Functional Models
Activity A3: Configure Non-Functional Models
«datastore» Hardware configuration Class diagram
«datastore» Software configuration class diagram
Activity A6: Develop Aspect Statemachines for Hardware Configuration
Activity A5: Develop Aspect Statemachines for Software Configuration
Activity A4: Develop Aspect Statemachines to Configure Model Elements

Fig. 13. Product configuration process

The overall process of configuring a product is shown as an activity diagram in Fig. 13. In the rest of the section, we will explain each activity of the process in detail along with examples.

4.1 Activity A1: Select a Core Behavior

This step selects one or more core behaviors from the set of the core behaviors available for a product line, which may be modeled using one or more state machines. In other words, this step resolves state machine variability (Section 3.2). For instance, in the case of Saturn product line, we have two options of core behaviors: Multi-site and Multi-way as discussed in Section 3.1. To configure a Saturn product, we either select Multi-site or Multi-way as its core behavior. Recall that each core behavior of the Saturn product line is specified using a class diagram and a state machine.

4.2 Activity A2/A3: Configure Functional/Non-functional Models

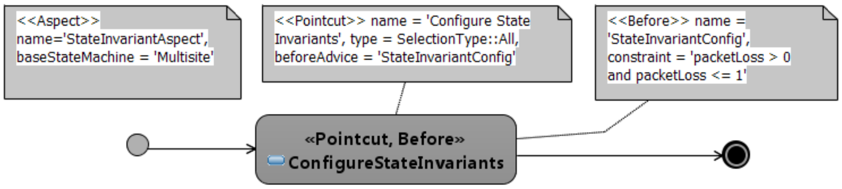
This activity involves configuring functional/non-functional models of the product line family. To achieve this, we configure the configurable parameters specified as class attributes in class diagrams by assigning particular values to these parameters. This step resolves class attribute value variability (Section 3.2). For example, in Fig. 5, *MaxNumberOfCalls* is a configurable parameter and holds information about the number of simultaneous video calls a product can make. For instance, one product can handle three simultaneous calls and hence this attribute needs to be set to '3' for the product. To specify the value for the attribute, the following constraint is defined:

```
context Multisite inv:
self. MaxNumberOfCalls = 3
```

Note that in the above example; we could have simply assigned value 3 to parameter *MaxNumberOfCalls* of an instance of class *Multisite*, but instead we decided to use constraints due to reasons that are both general in nature or specific to our tool support: 1) An attribute can take range of possible values and may possibly have relationships with other configurable and non-configurable attributes; therefore OCL constraints should be specified to capture such complex cases, 2) By writing constraints, we don't need to make any change to the product line models since we define constraints for each product, and 3) To automatically generate executable test cases from the product models plus constraints specifying configuration information, using our existing MBT tool called TRUST [5], the constraints are solved using our search-based OCL Solver [10].

4.3 Activity A4: Develop Aspect State Machines to Configure Model Elements

This activity involves configuring model elements using aspect state machines. For the example presented in Fig. 8, state invariants of the states stereotyped as <<*Variability*>> can be configured using an aspect state machine shown in Fig. 14. In the figure, attribute *baseStateMachine* of <<*Aspect*>> indicates that this aspect state machine will be woven into the *'Multisite'* state machine shown in Fig. 7. The <<*Pointcut*>> stereotype on *ConfigureStateInvariants* selects all states of *Multisite* and apply a before advice '*StateInvariantConfig*'. Attribute *constraint* of <<*Before*>> has value *'packetLoss > 0 and packetLoss <=1'*, resulting in this constraint being conjuncted to the state invariants of all the states in the *Multisite* state machine. More details on aspect state machines and configuring more complicated model elements can be found in [9]. In our another previous work, we developed a weaver [9] for aspect state machines in Kermeta [18], which reads a base state machine and aspect state machines and produces a woven state machine. The woven state machine is later on used for executable test case generation using our TRUST tool [5].



<<Aspect>> name='StateInvariantAspect', baseStateMachine = 'Multisite'
<<Pointcut>> name = 'Configure State Invariants', type = SelectionType::All, beforeAdvice = 'StateInvariantConfig'
<<Before>> name = 'StateInvariantConfig', constraint = 'packetLoss > 0 and packetLoss <= 1'
«Pointcut, Before» ConfigureStateInvariants

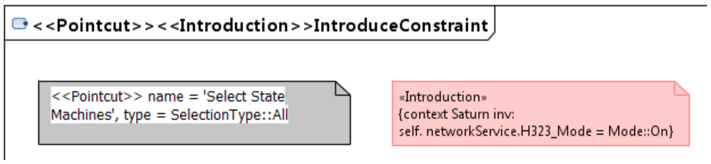
Fig. 14. Example of configuring a product for state machine model element variability

4.4 Activity A5/A6: Specifying Hardware/Software Configurations

In this activity, we specify constraints on selected hardware/software configurations, which are captured as class attributes in class diagrams (e.g., shown in Fig. 10 and Fig. 11). For example, for the Multi-site state machine (Fig. 7), to make H323 videoconferencing calls, attribute H323_Mode in class *NetworkService* (Fig. 10) has to be set to *true*, to successfully execute test cases. The ranges of values of these attributes along with their relationships with other attributes can again defined in the same way as for the class value attributes variability during activities A2/A3 (Section 4.2). An example is shown below:

context Saturn inv:
self. networkService.H323_Mode = Mode::On

Above, we only presented how to specify a constraint on one configuration parameter, but ideally such constraints are defined based on a set of software and hardware configuration parameters. In addition, these constraints are normally associated with more than one state machine across various subsystems. To apply these constraints to more than one state machine, again we use AspectSM, as this is clearly a crosscutting activity. For instance, if we need to specify the above constraint for all state machines of a product, we need to develop as aspect state machine as discussed in [9]. Fig. 15 provides an excerpt of an example of configuring a product to resolve class attributes selection variability. In this figure, we can see that the constraint that sets *H323_Mode* to *On*, is captured as an attribute of AspectSM stereotype <<Introduction>> and this constraint is applied to all states of a product (shown as an attribute of <<Pointcut>>). Note that both stereotypes are applied to an instance of uml::Statemachine metaclass. This information is omitted due to space limitation.



<<Pointcut>><<Introduction>>IntroduceConstraint
<<Pointcut>> name = 'Select State Machines', type = SelectionType::All
«Introduction» {context Saturn inv: self. networkService.H323_Mode = Mode::On}

Fig. 15. Example of configuring a product for constraint variability

When automatically generating test cases, such constraints are solved using our OCL solver [10] to generate valid configuration specifications. In addition, to set those valid configurations such that automatically generated test cases from the state machines using our model-based testing tool TRUST [5] can be executed, statements corresponding to these configurations in a particular test scripting language such as Python, are inserted to the generated test cases.

5 Case Study

Our case study is a product line family of VCSs called Saturn, developed in Cisco Systems Inc, Norway [3]. The Saturn family consists of various hardware codecs ranging from C20 to C90. C20 is the lowest end product with minimum hardware and has lowest performance in the family.

Saturn product line family consists of 20 subsystems such as audio and video subsystems. Each subsystem can run in parallel to the subsystem implementing the core functionality (Section 3) that deals with establishing videoconferences. Each subsystem has at least one state machine specifying its functionality and on average such state machine has five states and 11 transitions. The biggest subsystem state machine has nine hierarchical state machines with 22 states and 63 transitions. Saturn product family models for non-functional behaviors consist of five aspect class diagrams and five aspect state machines modeling various robustness behaviors. The largest aspect state machine specifying robustness behavior has three states and ten transitions, which would translate into 1604 transitions in standard UML state machines if AspectSM were not used. Saturn product line family models also consist of 124 hardware configuration parameters and 99 software configuration parameters.

The results of configuring various products in Saturn product line family are summarized in Table 1. The columns show the various types of product line models (Section 3.1), which must be configured for various products. The *Core Behavior* column indicates the configuration of the state machine variability for each product, where all the products support multi-site except C20. The *Functional Behavior* and *Non-Functional Behavior* columns show the number of instances of the class attribute value variability. For all the products, 13 and 11 configuration parameters need to be configured for each product regarding functional behavior and non-functional behavior, respectively. The *Software Configuration* and *Hardware Configuration* columns show the number of class attributes that need to be configured. For hardware configuration, as shown in the table, less number of parameters must be configured for the lowest end product C20 as compared to its higher end products.

Using our methodology the modeling effort is dividing between developing one set of product line family models and configuring products. In our study, from Table 1 we can see that the latter is roughly equal to specifying on average 13 OCL constraints for functional behaviors, 11 OCL constraints for non-functional behaviors, and modeling on average nine aspect state machines for hardware/software configurations. On the other hand, without using our methodology, for each product one needs to devise a set of models roughly equivalent to the size of Saturn product family models. This means that given the four products in the Saturn product line family, we would need to create the equivalent of four sets of Saturn product line family models (with a minimum of 20 state machines, five aspect state machines, and hundreds of OCL constraints). As the number of products of the product line family increases, one can easily see that without using our methodology, modeling quickly gets out of hands.

Table 1. Summarized results for configuring various products

Product	Core Behavior	Functional Behavior	Non-Functional Behavior	Software Configuration	Hardware Configuration
C20	Multi-way	13	11	5	12
C40	Multi-site	13	11	6	16
C60	Multi-site	13	11	6	19
C80	Multi-site	13	11	6	42

Since modeling behavioral variability may crosscut various state machines (the configuration activities A5 and A6 in Fig. 13) or may crosscut several model elements within one state machine (the configuration activity A4 in Fig. 13), using AspectSM, such crosscutting variability can be modeled separately, thereby saving modeling effort by taking advantage of AOM features.

Modeling behavioral variability using AspectSM provides enhanced separation of concerns. For instance, in Fig. 14, crosscutting variability in the state invariants is captured as aspect state machines. This means that a modeler, or several of them with possibly different expertise, can focus on each type of crosscutting variability. For example, in our current application, crosscutting variability related to media quality and crosscutting variability on networks can be handled separately using our AspectSM variability configuration mechanism. This is very important for our industrial partner since they have separate groups for different kinds of testing activities including functional, video, audio and network testing. Using our methodology each group can independently model crosscutting variability related to their expertise.

6 Related Work

A wide variety of UML-based product line modeling approaches has been proposed in literature. The work reported in [19] extends the UML class diagram notation to model structural variability by defining a set of stereotypes to capture for example variation points (<<*VariationPoint*>>) and variants (<<*Variant*>>). Similar kinds of UML profile-based approaches [19-21] have been proposed to model variability in UML class and sequences diagrams. The way we model class attribute variability is similar to these approaches since we apply <<*Variability*>> to class attributes of class diagrams.

The approaches reported in [22] and [23] support product line modeling by defining various stereotypes on use case diagrams, collaboration, and state chart diagrams. The stereotypes are similar to those defined in [19-21]. For example, for state charts [22], a transition can be marked as a variation point and for different products the transition is defined in different ways. Our methodology of specifying *state machine model element variability* is similar to these approaches as we stereotype state invariants in a state machine using <<*Variability*>>, which are variation points. However, in [22], variability configuration is realized by creating new models for different products with resolved variation points. In contrast, our approach involves specifying various product configurations using AOM and more specifically using the AspectSM profile. By doing so, the configurations are captured separately from product line models, thereby enhancing separation of concerns. In addition, our methodology defines a comprehensive classification of variability in UML state machines and provides a detailed product configuration process to deal with each type of variability, which are missing in the existing related works published in the literature. Furthermore, our objective is to reduce modeling effort required for MBT, which is not focused by any existing work in the literature.

Several AOM approaches have been proposed (e.g., [24-27]) to model variability, most of which are defined by introducing new variability modeling languages. For instance, the work reported in [27] defines two metamodels, one for defining assets (commonality) and the second for modeling variability. These two metamodels are then weaved using AOM such as to obtain a unique metamodel extended with variability concepts. Instead of using any of these approaches, we rely on our UML profile (AspectSM) since, in our case study context as in many others, minimizing extensions to UML is expected to ease practical adoption. In addition, we have a weaver for AspectSM [9], which has been successfully applied to support MBT in industrial contexts.

Since our product line modeling and configuration framework is proposed in the context of supporting automated state-based testing, modeling and configuration mechanisms are proposed for various types of behavioral variability in UML state machines. As visible from our discussion above, existing works did not address such objectives as their motivations were different.

7 Conclusion and Future Work

Exploring a new application domain for such technology, we propose a product line modeling and configuration methodology to support Model-Based Testing (MBT) in the context of product lines. This was applied to a Video Conferencing System (VCS) product line called Saturn developed by Cisco Systems Inc, Norway, for the purpose of reducing the overall modeling effort required for MBT. The methodology includes: 1) defining a classification of variability that can exist in UML state machines, 2) proposing a set of standard-based mechanisms (e.g., stereotypes) to model different types of variability, 3) defining a product configuration process addressing various types of variability using as Aspect-Oriented Modeling (AOM)—and more specifically a profile named AspectSM—and the Object Constraint Language (OCL), and 4) automated processing of configuration specifications to support MBT by using our aspect weaver and a previously developed search-based constraint solver.

We applied our methodology to the Saturn product line family and configured four of its products. Results showed that the overall amount of modeling effort required for creating test-ready models, for multiple products, in order to support MBT can be significantly reduced using our methodology. Moreover, using AspectSM provides enhanced separation of concerns since various variability types addressing different concerns (e.g., Audio and Video) can be separately modeled by different people according to their expertise and responsibilities. Given that, in our previous work, we conducted various controlled experiments showing that using AspectSM can significantly reduce modeling effort, improve quality of models, and reduce modeling errors, using it to support product line, model-based testing seems to be a promising approach.

In the future, we are planning to apply our methodology to other VCS product lines in Cisco. We also plan to conduct a field study by involving industrial testers to determine how easy it is for them to configure a product by using our methodology. In addition, we plan to conduct a series of controlled experiments in academic settings to assess our methodology from various aspects such as ease of application, modeling quality, and modeling effort.

Acknowledgements. We would like to thank Marius Christian Liaaen from Cisco Systems Inc, Norway for very useful discussions and providing us access to various Video Conferencing Systems for this work. Lionel Briand was supported by a FNR PEARL grant.

References

1. Northrop, L.M.: SEI's Software Product Line Tenets. IEEE Software 19, 32–40 (2002)
2. `http://splc.net/fame.html`
3. `http://www.cisco.com`
4. Ali, S., Briand, L.C., Arcuri, A., Walawege, S.: An Industrial Application of Robustness Testing Using Aspect-Oriented Modeling, UML/MARTE, and Search Algorithms. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 108–122. Springer, Heidelberg (2011)
5. Ali, S., Hemmati, H., Holt, N.E., Arisholm, E., Briand, L.C.: Model Transformations as a Strategy to Automate Model-Based Testing - A Tool and Industrial Case Studies. Simula Research Laboratory, Technical Report (2010-01) (2010)
6. Drusinsky, D.: Modeling and Verification using UML Statecharts: A Working Guide to Reactive System Design. In: Runtime Monitoring and Execution-based Model Checking, Newnes (2006)
7. Gomaa, H.: Designing Concurrent, Distributed, and Real-Time Applications with UML. Addison-Wesley Professional (2000)
8. Lavagno, L., Martin, G., Selic, B.V.: UML for Real: Design of Embedded Real-Time Systems. Springer (2003)
9. Ali, S., Briand, L.C., Hemmati, H.: Modeling Robustness Behavior Using Aspect-Oriented Modeling to Support Robustness Testing of Industrial Systems. Accepted for Publication in the Systems and Software Modeling (SOSYM) Journal (2011)
10. Ali, S., Iqbal, M.Z., Arcuri, A., Briand, L.: A Search-Based OCL Constraint Solver for Model-Based Test Data Generation. In: 11th International Conference on Quality Software (QSIC). IEEE (2011)
11. Filman, R.E., Elrad, T., Clarke, S., Aksit, M.: Aspect-Oriented Software Development. Addison-Wesley Professional (2004)
12. Ali, S., Yue, T., Briand, L.C., Malik, Z.I.: Does Aspect-Oriented Modeling Help Improve the Readability of UML State Machines? Under Consideration for a Publication in a Journal (2011)

13. Ali, S., Yue, T.: Comprehensively Evaluating Conformance Error Rates of Applying Aspect State Machines for Robustness Testing. In: International Conference on Aspect-Oriented Software Development (AOSD 2012). ACM (2012)
14. Ali, S., Yue, T., Briand, L.C.: Empirically Evaluating the Impact of Applying Aspect State Machines on Modeling Quality and Effort Simula Research Laboratory, Technical Report (2011-06) (2011)
15. Yue, T., Ali, S.: Bridging the Gap between Requirements and Aspect State Machines to Support Non-functional Testing: Industrial Case Studies. In: Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., Kolovos, D. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 133–145. Springer, Heidelberg (2012)
16. `http://www.omgmarte.org/`
17. Iqbal, M.Z., Ali, S., Yue, T., Briand, L.: Experiences of Applying UML/MARTE on Three Industrial Projects. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, pp. 642–658. Springer, Heidelberg (2012)
18. IRISA and INRIA, `http://www.kermeta.org/`
19. Clauss, M.: Generic modeling using uml extensions for variability. In: OOPSLA (2001)
20. Ziadi, T., Hélouët, L., Jézéquel, J.-M.: Towards a UML Profile for Software Product Lines (2004)
21. Edson Alves de Oliveira, J., Gimenes, I.M.S., Huzita, E.H.M., Maldonado, J.C.: A variability management process for software product lines. In: Proceedings of the 2005 Conference of the Centre for Advanced Studies on Collaborative Research, pp. 225–241. IBM Press, Toranto (2005)
22. Gomaa, H., Shin, M.E.: Multiple-View Meta-Modeling of Software Product Lines. In: Proceedings of the Eighth International Conference on Engineering of Complex Computer Systems, p. 238. IEEE Computer Society (2002)
23. Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley Professional (2004)
24. Jayaraman, P., Whittle, J., Elkhodary, A.M., Gomaa, H.: Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 151–165. Springer, Heidelberg (2007)
25. Morin, B., Klein, J., Barais, O., Jezequel, J.-M.: A generic weaver for supporting product lines. In: Proceedings of the 13th International Workshop on Early Aspects, pp. 11–18. ACM, Leipzig (2008)
26. Groher, I., Voelter, M.: Using Aspects to Model Product Line Variability. In: Early Aspects Workshop at SPLC (2008)
27. Morin, B., Perrouin, G., Lahire, P., Barais, O., Vanwormhoudt, G., Jézéquel, J.-M.: Weaving Variability into Domain Metamodels. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 690–705. Springer, Heidelberg (2009)

Sensitivity Analysis in Model-Driven Engineering

James R. Williams, Frank R. Burton, Richard F. Paige, and Fiona A.C. Polack

Department of Computer Science, University of York,
Deramore Lane, York, YO10 5GH, UK
{jw,frank,paige,fiona}@cs.york.ac.uk

Abstract. Sensitivity analysis has been used in scientific research to explore the validity of models. Software engineering is inherently uncertain; we propose that sensitivity analysis can be used to analyse and quantify the effects of uncertainty when model management operations are applied to models. In this paper, we consider forms and measures of uncertainty in software engineering models. Focusing on data uncertainty, we present a framework for sensitivity analysis, and create an instantiation of the framework for the CATMOS decision-support tool. We show how this can be used to qualify the output of the entailed model management operations and thus improve both the confidence and understanding of models.

1 Introduction

Models are created for a purpose. Historically, models have been used for illustration and to aid problem understanding. In Model-Driven Engineering (MDE), models have become primary artefacts in the development lifecycle. Models are commonly used in code generation, decision-making processes, simulation, or as input to some form of *model management operation* (MMO). More concretely, MMOs are executable artefacts which manipulate models in some way and are often forms of *transformation*, *comparison*, or *validation*.

All software engineering suffers from some degree of uncertainty [14]. Any form of modelling is subject to different levels of uncertainty, such as errors of measurement or interpretation, incomplete information, and poor or partial understanding of the domain [11]. When MMOs are applied to a model, uncertainty can lead to unexpected behaviour, or a small change in a model might result in a large change in the output of the MMOs. Modelling uncertainty can have a significant impact on artefacts that use the models or their information.

Sensitivity analysis provides a means to explore how changes in an input model affect the output of an MMO. Sensitivity analysis can provide a modeller with confidence that a model and its associated MMO(s) resemble the domain, and can expose areas in the domain that require a deeper understanding [11]. Furthermore, highlighting sensitive parts of a model can provide insight into the execution of an MMO, if the execution is influenced significantly by sensitive parts of the model [10]. Sensitivity analysis can also show the relationships between model elements that may not be apparent from simply examining a model and its MMO.

In this paper we introduce sensitivity analysis in MDE, and present an extensible framework that enables metamodel developers to provide sensitivity analysis tool support for their domain. We provide a metamodel for capturing the data uncertainty in models, which is used to analyse the effect of uncertainty on the results of an MMO.

To motivate and illustrate the use of sensitivity analysis, we introduce CATMOS, an acquisition planning tool for capability management (section 2). CATMOS generates a set of solution models from a problem analysis; providing an analysis of these solutions improves the basis for acquisition decision-making. Section 3 introduces sensitivity analysis, a technique developed in the modelling of natural systems [11], and section 4 shows how it might apply to MDE. In section 5, we outline our sensitivity analysis framework, applying it to a set of CATMOS solution models in section 6. Related work is considered in section 7, whilst section 8 considers the pragmatics and challenges of sensitivity analysis in MDE.

2 Motivating Example

CATMOS (Capability Acquisition Tool with Multi-Objective trade-off Support) is an MDE tool that provides support for capability-based planning, to facilitate systems of systems acquisition [3]. CATMOS is targeted at acquisition scenarios that involve significant costs. In these situations, a better understanding of uncertainty can increase confidence in decisions, and thus in investments to be made in acquisition. In CATMOS, users model their problem scenario by defining a set of desired system capabilities and a set of available components. CATMOS then generates a set of output models that represent combinations of components that can meet the capability requirements of the scenario. Every solution model has a cost and a capability score, and the solution set can be represented as a Pareto front of capability versus cost (i.e. no solution dominates any other in all objectives), which can be used by stakeholders in decision making.

2.1 The Airport Crisis Management Scenario

In this paper, we apply sensitivity analysis to a CATMOS application to airport crisis management (ACM). CATMOS is used to inform a decision as to what set of resources should be invested by the airport in a scenario in which a fire breaks out at an airport gate [9]. Figure 1 shows a Pareto front produced by CATMOS, in which each star represents one of the solution models proposed. The circles represent models from "non-dominating rankings": in-effect, the next best Pareto front. The measurements used to calculate the capability (y-axis) are features of the problem scenario that are all subject to uncertainty; the level of uncertainty can affect the interpretation of the results produced by CATMOS. For instance,

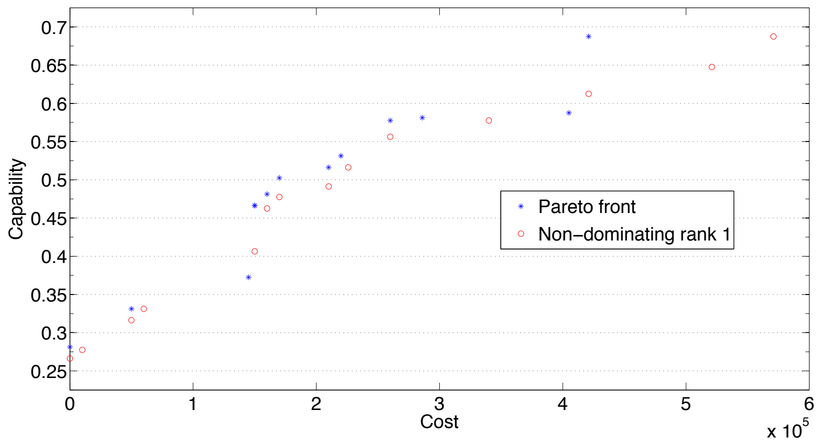


Fig. 1. The Pareto front and first non-dominating rank produced by CATMOS for the ACM problem

if the capability of a Pareto-optimal solution is subject to high uncertainty, but the capability of a nearby non-dominating solution is less uncertain, the decision makers may prefer the non-dominating solution. Sensitivity analysis allows us to explore uncertainty in each model's capability score.

3 Sensitivity Analysis

Any form of modelling is potentially subject to uncertainty that can be introduced by numerous possible sources, such as errors of measurement, incomplete information, or from a poor or partial understanding of the domain [11]. Sensitivity analysis aims to determine how variation in the output of a model[1] can be attributed to different sources of uncertainty in the model's input [11]. The goal is to increase the confidence in a model by understanding how its *response* (output) varies with respect to changes in the inputs.

In [11], sensitivity analysis is proposed for the following forms of model validation: to determine whether a model is sensitive to the same parameters as its subject; to identify that a model has been tuned to specific input values (and thus is inflexible to model change); to distinguish factors that result in variability of output and those with little influence on the output (which could be omitted); to discover regions of the space of input values that maximise result variation; to find optimal input values (for model calibration); to find factors that interact (and thus need to be treated as a group) and expose their relationships.

Sensitivity analysis approaches broadly fall into local or *one-at-a-time* (OAT), and *global* analyses. OAT analyses address uncertain input factors independently,

[1] In this section, the term "model" is used in the abstract sense of modelling, and although this includes MDE models, the term should be read in the broader setting.

revealing the extent to which the output is determined by any one input [11]. Global analyses perturb all input factors simultaneously, and can thus address dependencies among inputs. To avoid combinatorial explosion, sensitivity analysis approaches use *input space sampling* techniques, such as random sampling, importance sampling, and latin hypercube sampling [11]. A range of statistical correlation or regression approaches can be used to interpret the results of sensitivity analysis.

The next section contexualises sensitivity analysis in MDE.

4 Sensitivity Analysis in MDE

Ziv et al [14] declared the *Uncertainty Principle in Software Engineering* which states that "uncertainty is inherent and inevitable in software engineering processes and products". This links well with sensitivity analysis, which aims to quantify the effects of said uncertainty.

In MDE, a model can be considered as representing both the abstract model of a domain and its operating context – a set of MMOs that apply to it. The areas of uncertainty are then potentially both the set of variable input factors and the parameters of MMOs. The *response* is (part of) the output of the MMO(s).

The uses of sensitivity analysis (above) relate to validation, that a model faithfully represents its domain, or that a model is faithful to a more abstract specification. Model validation in MDE is important, and is commonly addressed using task specific languages, such as OCL. Sensitivity analysis provides an alternative, exploratory approach to validation. Furthermore, it presents an opportunity to understand the effects of modelling decisions, which can be crucial in complex or poorly understood domains.

In this section we describe three areas that uncertainty can arise in MDE, and discuss the different ways in which we might measure the response of an MMO.

4.1 Areas of Uncertainty

In section 2, we note that capability measurement is affected by uncertainty. The sources of uncertainty are the same in software engineering as in scientific modelling (errors of measurement or interpretation, incomplete information, poor or partial understanding of the domain [11]). We identify three areas of uncertainty in MDE and consider ways in which we might quantify their effects on the application of MMOs.

Data Uncertainty. In modelling data structures, uncertainty is introduced when deciding types and values of attributes – for instance, string types and * multiplicities are often used because of uncertainty about the domain. In modelling transitional systems (e.g. using state machines), uncertainty arises in determining the values used in transition guards. In modelling reactive systems (e.g. using Petri nets), the firing conditions of transitions may be similarly uncertain.

Strengthening or weakening Boolean conditions may significantly affect the behaviour of systems. Sensitivity analysis can be applied to attributes, guards and firing conditions, to validate the states and behaviours of modelled systems.

To analyse data uncertainty, we can vary the values of attributes with respect to a range or distribution of possible values, and see how the MMO output is affected.

Structural Uncertainty. Uncertainty also arises when making decisions about the structure of a model. One example of structural uncertainty is deciding between aggregation and composition for an association. This decision could have a huge effect on an MMO. For example, if the MMO deletes the owner element then the type of association determines which elements remain available for the remainder of the MMO's execution.

Analysing structural uncertainty might be achieved through pattern matching and replacement. A modeller could define a set of patterns of model elements (possibly with respect to the underlying metamodel) where each pattern represents a set of equivalent patterns. Sensitivity analysis could then be used to analyse the effect of replacing parts of the model with different patterns, to discover the effect on the output of the MMO. This would allow the modeller to optimise their models in conjunction with the associated MMOs. Alternatively, one might perform simple mutations to a model (e.g. deleting elements) and analysing the effects of these mutations on the output. This would highlight the parts of the model that are important to the execution of the MMO(s) and thus sensitive to change.

Behavioural Uncertainty. Whereas the previous two categories related directly to models, behavioural uncertainty relates to the operating context of the model – i.e. the MMO. The model and its MMO may be developed by different, independent teams, and it may not be known to the modeller what the operation does or how it does it. Knowing the operating context of a model removes this uncertainty – for example, a modeller may make a different design decision if the model is to be used for code generation as opposed to illustration.

Analysing behavioural uncertainty is challenging, though has been attempted in [1].

In the rest of this paper, we focus on analysing data uncertainty. First, we consider the issue of measurement.

4.2 Measuring the Response

Sensitivity analyses in scientific modelling tend to address readily-evaluable parameters. However, in MDE, the result of applying an MMO is often another model (e.g. in the case of a model-to-model transformation), so it can be challenging to provide a useful measure of the effects of changing the input model. A simple count of differences between the original output model and each of the output models created from varying the input, may not be a suitable measure of impact. Focusing the analysis on the effects on a small part of the output model, however, may be more appropriate. In the cases where a model is used to generate code, the response might be measured by executing the generated code and analysing, for example, the execution trace, the memory consumption, or the program's output.

Measurement is domain specific. Saltelli et al [11] comment that different measures of sensitivity directly affect the outcome of the analysis, and declare that there is no universal recipe for measuring the response. Any MDE framework that supports sensitivity analysis needs to provide the ability to support multiple forms of response measurement and comparison.

5 A Framework for Applying Sensitivity Analysis to Models

This section presents our implementation of an extensible framework for applying sensitivity analysis to models in MDE. Specifically, the framework is implemented on top of the Eclipse Modeling Framework (EMF) [12]. The framework provides a repeatable way of allowing metamodel developers to create tool support for rigorous analysis of models in their domain. Section 5.1 overviews the process which our framework implements. Section 5.2 describes a simple representation of models that makes the creation of variants of a model straightforward. We introduce a metamodel for expressing data uncertainty in section 5.3. Section 5.4 describes the framework's support for different sampling methods, and presents two default methods. Section 5.5 shows how we provide support for automated analysis of responses, as well as producing a sensitivity analysis report for the modeller.

5.1 Overview

Figure 2 illustrates the process of applying sensitivity analysis to a model. First, we need an *uncertainty model* for expressing the data uncertainty in the *input model* (explained in section 5.3). These two models are the inputs to an *input space sampler* – a bespoke model generator that selects variations of a model within the scope described by its uncertainty model. Selection is controlled by the sampling method of sensitivity analysis being applied (described in section 5.4).

Each of the generated models (*Model'*) in the sample is executed against the stated *model management operation* and the *output* is logged. The model management operation may be a model transformation, a simulation, or a complex workflow of operations, as determined by the user. Once all generated models have been executed, the set of outputs is fed into a *response analyser* which applies the sensitivity analysis and produces reports for inspection. As mentioned previously, there is no single, optimal way to analyse sensitivity, and the output of an MMO can take many forms. Therefore, whilst providing a number of default response analysers for numerical output, our framework also allows users to develop their own (see section 5.5).

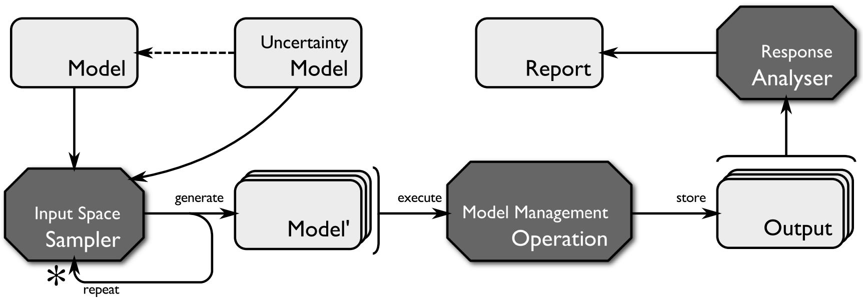


Fig. 2. The process of applying sensitivity analysis to a model

The model variation method, used by the input space samplers, is made possible by utilising a simple, integer-based, representation of models [13], which we briefly describe next.

5.2 A Simple Representation for Models

In [13] we show how a model can be represented as a single string of integers, composed of a number of *segments* where each segment represents a single object in the model. Figure 3 shows the structure of a segment in our representation. The first node in a segment (the *class node*) identifies the object's meta-class, and successive nodes (the *feature nodes*) define the values of features from that class. Feature nodes are grouped into pairs – the first member of the pair, the *feature selector node*, identifies a feature of the meta-class, and the second member of the pair, the *feature value node*, specifies the value of that feature.

In order to reference meta-elements, we assign identifiers to them. Each meta-class is given an identifier, and each of its meta-features are also assigned identifiers, unique to that meta-class only. Meta-elements that cannot be instantiated, such as enumeration types and abstract classes, are not assigned a class-level identifier.

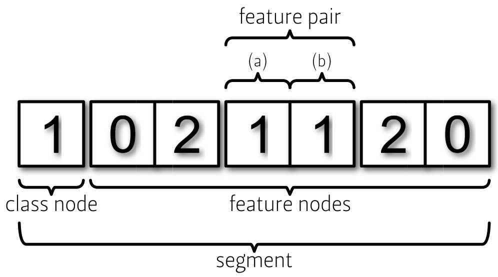
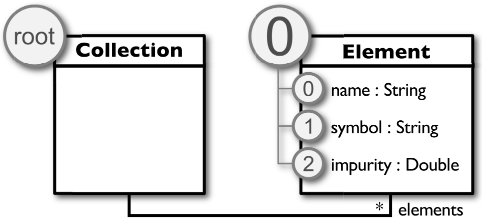
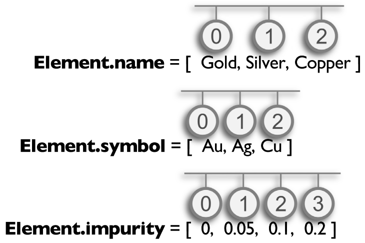


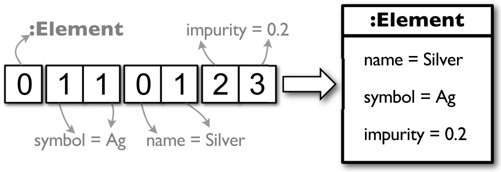
Fig. 3. The structure of a *segment*, the building block for representing models as integers. (a) is the *feature selector node*, and (b) is the *feature value node*.



(a) Step 1: Assigning identifiers to the metamodel.


Element.name = [Gold, Silver, Copper]
Element.symbol = [Au, Ag, Cu]
Element.impurity = [0, 0.05, 0.1, 0.2]

(b) Step 2: Data finitisation.



(c) Step 3: Mapping a segment to an object.

Fig. 4. The steps taken to instantiate an object from our integer-based representation. Meta-element identifiers are displayed in grey circles.

A key part of the mapping from integers to models is the process of *metamodel finitisation* [13]. The modeller is required to define a model that describes the set of all possible models that can conform to the metamodel. If the data type of an attribute is infinite, this involves determining finite concrete values that the attribute can take. This allows us to index the possible values for use in the mapping. In addition, the modeller must specify a "root" meta-class, i.e. the container object in which all model objects reside. For a more complete description of finitisation, see [13].

A simple example of mapping a segment to an object is shown in figure 4. We annotate both the metamodel and its finitisation model with identifiers, including specifying which meta-class is the root. The root object is automatically instantiated during generation, independent of the generated integer strings, and therefore does not need to be assigned an identifier in this instance. The reverse mapping is also possible. We have implemented these mappings for *Ecore*, the metamodelling language of EMF.

5.3 A Metamodel for Data Uncertainty

Figure 5 is the metamodel we have defined to allow users to describe the data uncertainty in their models. Capturing uncertainty in a model allows developers to manipulate this information with MMOs. Specifying the object reference in the DataUncertainty class allows a modeller to describe the uncertainty of a specific

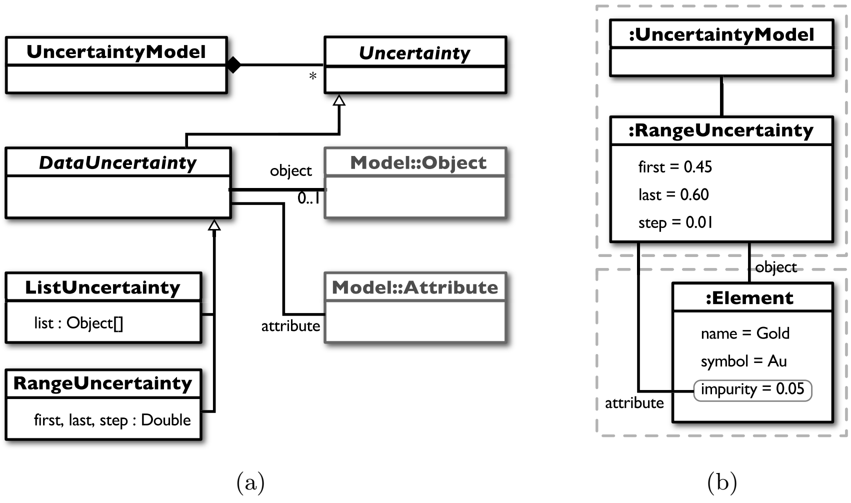


Fig. 5. (a) The metamodel used to describe uncertainty in a model. (b) An example uncertainty model. The dotted lines show the boundaries between the uncertainty model and the model which it references.

model element's attribute. If the user does not assign the object reference, then the uncertainty is valid for all instances of that attribute. For example, using the metamodel shown in figure 4a, we can either assign the uncertainty to the impurity attribute of the object whose name is Gold (figure 5b), or we can assign it to the impurity attribute of all instances of the Element class.

The uncertainty metamodel has been designed with evolution in mind. Currently the metamodel only supports data uncertainty, but the metamodel can be extended to capture other forms of uncertainty (by extending the Uncertainty class) whilst maintaining backward compatibility with existing uncertainty models. We now describe how different sensitivity analysis methods can use the uncertainty model and the integer-based representation to create variations of a model for which to analyse.

5.4 Sampling the Input Space

The process of sampling the input space (i.e. generating variants of a model) is illustrated in figure 6. The variation of a model is created by mapping the model to a segment list and altering the appropriate feature pair value(s) before transforming back to a model which contains the new data. The transformation from a model to a segment list contains a hook for listeners to detect when segments and feature pairs are created. We have defined a feature pair creation listener which detects whether the feature pair being created represents one of the features described in the uncertainty model. If so, we keep a reference to that feature pair, along with the set of values described in the uncertainty model.

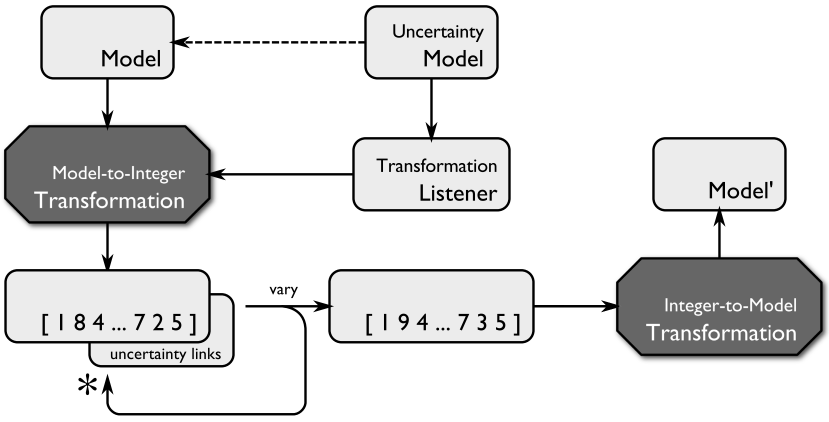


Fig. 6. Utilising an integer-based representation of models, to create variants of a model with respect to a set of specified uncertainties.

Once the model-to-integer transformation is complete, the sampling method controls which of the *uncertain feature pairs* to vary. Each variation of the segment list is mapped back into a model for consumption by the MMO. As there is commonly a large number of possible variations, our framework allows users to create custom sampling methods and provides two by default. A *one-at-a-time* sampler adjusts each attribute independently, creating samples for each possible value that each attribute can take, whilst maintaining all other attributes in their original form. A *random sampling* method adjusts all attributes simultaneously, selecting a value for each attribute at random from the uncertainty model. The user specifies the size of the sample that they desire.

5.5 Response Measures

The input space sampler produces a set of variations of a model which are then fed through the MMO to obtain the associated responses. The final component in our framework is the *response analyser* which provides various analysis methods and a HTML report generator. Framework implementations can choose which analyses to apply and include in the report. Furthermore users can extend the analysis methods with custom methods, or integrate with existing statistical analysis packages, such as *jHepWork*[2], *R*[3], or *MatLab*[4]. The results are saved to disk as a comma-separated-values (CSV) file, so users can apply further analyses as they gain a deeper understanding of the results. (In future iterations of this work, we plan to develop a metamodel to capture the results as models. Each analysis method will then be written as a model transformation, removing the burden of using CSV files and improving reusability.) The reporting component

[2] jHepWork website: `http://jwork.org/jhepwork/`
[3] R website: http://www.r-project.org/
[4] MatLab website: `http://www.mathworks.co.uk/`

of the framework utilises an open source template engine, *StringTemplate*[5], and users can define their own templates to incorporate new analyses in the reports.

Currently the framework provides scatter plot based analysis for both OAT and global sampling methods, as well as some useful statistics such as mean, median, percentiles and standard deviations. JFreeChart[6] is used to create a scatter plot for each of the uncertain attributes, illustrating the effect that the attribute has on the MMO's response.

6 Case Study: CATMOS and ACM

Section 2 introduced an acquisition decision tool, *CATMOS* [3], and a case study to which the tool has been applied. We have developed an instance of our sensitivity analysis framework for the CATMOS tool. This section briefly presents our implementation and describes some interesting results.

6.1 CATMOS Sensitivity Analysis

Extending the framework to support CATMOS required defining two Java classes (totalling approximately 300 LOC), although more would be required to provide a more sophisticated user interface. One class controls the loading of models, specifies the sampling method(s) and starts the analysis, and the second class provides the response calculation, controls the types of sensitivity analysis applied to the set of responses and generates reports. For repeatability, we have made the models analysed and generated reports available online[7].

The experimental goals that we wanted to analyse were:

EG1. Determine how each factor contributes to the overall capability score (the response) of the model;
EG2. Understand how the different model factors relate to one another;
EG3. Provide insight into the confidence of the frontiers produced by CATMOS.

We therefore applied OAT sampling and random sampling to create scatter plots for each factor in the 28 solution models (the Pareto front and first non-dominating rank) produced by CATMOS for the ACM case study. Furthermore, we developed a custom response measure which produces a plot showing the response distribution, based on the sampling, for each of the models in the solution set. The response measured was the capability score of the model. In total, we evaluated nearly 20,000 model variants which took approximately 3 hours to execute on a 2GHz Intel Core 2 Duo Macbook with 2 GB of RAM.

6.2 Results

The results found from attempting to answer the experimental goals described in the previous section are now presented.

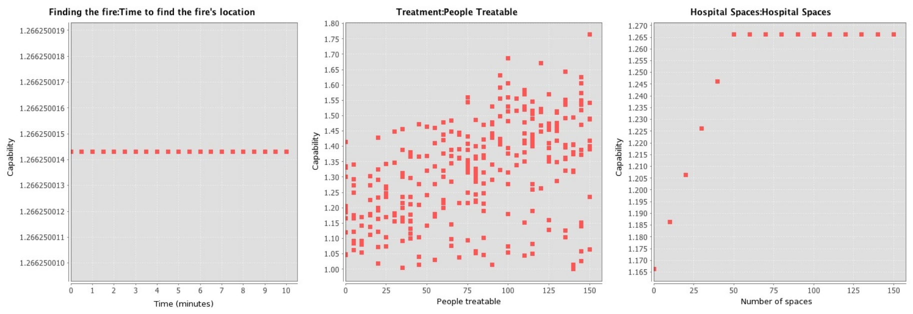
[5] StringTemplate website: `http://stringtemplate.org/`
[6] JFreeChart website: `http://www.jfree.org/jfreechart/`
[7] Results available at:`http://www.jamesrobertwilliams.co.uk/models12-sa`

EG1: Response Contribution. Figure 7 illustrates three types of response contribution which were observed. Figure 7a shows a factor which has no effect on the calculation of the capability provided by that solution (response of the MMO). One possible cause of this is that the range of values analysed were not appropriate, and more discussion with a domain expert would be required. However, upon analysing the solution more closely, it became clear that the component providing the particular capability had an unfulfilled dependency, meaning that it had no effect on the output. In this case, the offending component should be removed from the model to avoid misleading decision makers.

Figure 7b shows the effect that the "People Treatable" component had during the random sampling method. You can see that there is a positive correlation, highlighting the importance of this factor as it is heavily influential in determining the capability score. Furthermore, it validates the assumptions of the model – a solution that is able to treat more people should have a higher capability score. Other factors also showed relationships with the response that further validated the model. Figure 7c illustrates a factor which exhibits an upper limit. This kind of result proves useful in understanding the problem, allowing stakeholders to act out 'what-if' scenarios.

EG2: Factor Relationships. 28 solution models were analysed and examining the OAT results for each model showed that some factors affect on the response differs between solutions. This shows a dependency on other factors in the model. Figure 7c, for example, is limited by the factor defining the number of people who can be transported to the hospital. Examining the random sampling plots shows little to no correlation for most of the factors. Therefore, further analysis is required to understand the relationship between multiple factors. This is, however, out of scope for this paper.

EG3: Frontier Confidence. Figure 8 shows the response distributions, provided by random sampling, overlaid on top of part of the original Pareto front



(a) No effect (b) Positive Correlation (c) Upper limit

Fig. 7. Three different types of contribution towards the response

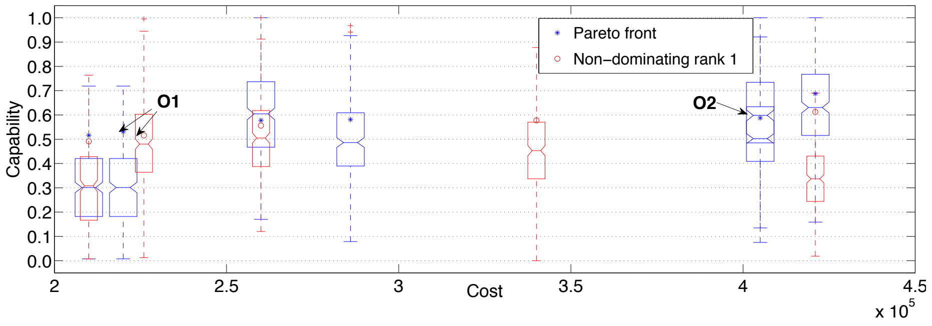


Fig. 8. Part of the random sampling response distribution for each model in the Pareto front and first non-dominating rank produced by CATMOS. Box width simply denotes the series – the wider boxes being related to the Pareto front.

and first non-dominating rank produced by CATMOS (figure 1). Two interesting observations can be seen. Firstly, **O1** in figure 8 highlights the case where a first non-dominating rank solution (red circle) appears to be, on average, better than a solution in the Pareto front (blue star). Secondly, **O2** highlights a case where CATMOS has returned two solutions in the Pareto front with the same capability and cost (this is allowed by CATMOS). A decision maker, may therefore think themselves safe in selecting either solution. The associated box plots, however, show that they have different interquartile ranges – suggesting that one solution is more sensitive to uncertainty than the other.

One flaw with this analysis method is that it does not take the probability distribution of each factor into account. For example, the random sample may have collected a large number of improbable combinations of factors, which then skewed the distribution. We plan to address this in future work (see section 8).

6.3 Summary

In this section we have shown how our implementation of the framework for the CATMOS tool has provided new insight into the results which CATMOS produces. This has shown how sensitivity analysis can provide deeper understanding of a problem, aid in the validation of a solution, and discover areas of a solution which need further investigation by domain experts.

7 Related Work

Uncertainty has been studied in the software engineering community for some time. Ziv et al [14] introduced the Uncertainty Principle in Software Engineering (UPSE) in 1997. They describe three sources of uncertainty in software engineering: the problem domain, the solution domain and human participation. They show how *Bayesian belief networks* can be used to model uncertainty in properties of software artefacts. The framework presented in this paper targets analysing data uncertainty of individual components (models) in a system, but could be used to provide belief values for those components in the Bayesian belief network of the entire system.

Easterbrook et al [5] use *multi-valued logics* to model uncertainty through the creation of additional truth values, and examine uncertainty using a multi-valued symbolic model checker. Similarly, [4] describes how *fuzzy logic* can be used to model uncertainty.

Harman et al [8] study the effects of data sensitivity on the results of a metaheuristic search algorithm for solving the *next release problem* [2]. Their aim is to identify which requirements are sensitive to inaccurate cost estimation. We believe that metaheuristic search techniques could prove fruitful at the sampling phase of sensitivity analysis. The search goal might be to discover a sample of input factor configurations which produce responses that vary dramatically from the original. The modeller would then be able to examine these extreme cases more carefully.

Autili et al [1] describe an approach to cope with uncertainty about the behaviour of reusable components with respect to a given goal. The approach aims to aid people in deciding which components to reuse in their system by extracting behavioural information from the set of possible components and estimating how well they fit together to solve the goal.

Goldsby and Cheng [7] address uncertainty in adaptive systems by automatically generating a number of system design models each exhibiting different behaviours. These models are then analysed to enable users to make decisions between trade-offs in non-functional characteristics and functional behaviour.

In the MDE domain, the closest work relating to sensitivity analysis that we have found is by Fleurey et al [6] who apply mutations to models in order to optimise a set of test models with respect to some metamodel and data coverage criteria. Creating these mutated models is similar to the process of creating variants of a model for sensitivity analysis, but their work is driven by different motivations – that of optimising test sets as opposed to discovering and analysing data sensitivity.

8 Conclusion and Future Work

In this paper we have motivated the need for supporting sensitivity analysis in MDE. Uncertainty can appear in all aspects of MDE; we have defined three categories of uncertainty – data uncertainty, structural uncertainty, and behavioural uncertainty. Due to the need for domain specific analysis of uncertainty, we have presented a framework that enables metamodellers to provide sensitivity analysis tool support for models conforming to their metamodels. This framework has an architecture which allows new sampling methods and response measures to be integrated, making it easy to tailor to new domains. We have illustrated the usefulness of this kind of framework by instantiating it to support an existing acquisition tool, and applied the analysis to a real problem. The analysis provided new insight into the solutions produced by the tool, which would better inform the acquisition decision makers.

There are a number of interesting directions for future work. Firstly, a deeper exploration of uncertainty in MDE is required. Uncertainty should be identified and dealt with, and so understanding where it can arise, and providing a way to aid users with managing uncertainty, would be very beneficial. This could possibly be in the form of tool support, or a set of guidelines or best practices.

With respect to the framework, we plan to extend the uncertainty metamodel to account for structural and behavioural uncertainty, and devise a method to analyse these. For example, for structural uncertainty, changing the type of an attribute would require changes to the underlying metamodel, which may not be trivial, and so some thought is required to determine the best approach to take. Further, we wish to include support for probability distributions for uncertainties. This would add an extra dimension to the analysis as it would also show the likeliness of a particular result occurring. Finally, we plan to develop a metamodel to capture the results of analysis in order to define analysis methods as model transformations, and thus increase reusability of analyses.

Acknowledgements. This research was supported by the EPSRC, through the Large-Scale Complex IT Systems project, EP/F001096/1. The primary author would like to thank Simon Poulding for his invaluable advice and feedback.

References

1. Autili, M., Cortellessa, V., Di Ruscio, D., Inverardi, P., Pelliccione, P., Tivoli, M.: EAGLE: engineering software in the ubiquitous globe by leveraging uncertainty. In: ESEC/FSE 2011, Szeged, Hungary, pp. 488–491 (September 2011)
2. Bagnall, A., Rayward-Smith, V.J., Whittley, I.: The next release problem. Information and Software Technology 43(14), 883–890 (2001)
3. Burton, F.R., Paige, R.F., Rose, L.M., Kolovos, D.S., Poulding, S., Smith, S.: Solving Acquisition Problems Using Model-Driven Engineering. In: Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., Kolovos, D. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 428–443. Springer, Heidelberg (2012)
4. Celikyilmaz, A., Turksen, I.B.: Modeling Uncertainty with Fuzzy Logic. STUDFUZZ, vol. 240. Springer (2009)
5. Easterbrooke, S., Chechik, M., et al.: xCheck: A model check for multi-valued reasoning. In: Proc. 25th International Conference on Software Engineering (ICSE 2003), Portland, Oregon, USA (May 2003)
6. Fleurey, F., Steel, J., Baudry, B.: Validation in model-driven engineering: testing model transformations. In: 1st International Workshop on Model, Design and Validation, pp. 29–40 (2004)
7. Goldsby, H.J., Cheng, B.H.: Automatically Generating Behavioral Models of Adaptive Systems to Address Uncertainty. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 568–583. Springer, Heidelberg (2008)

8. Harman, M., Krinke, J., Ren, J., Yoo, S.: Search based data sensitivity analysis applied to requirement engineering. In: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, GECCO 2009, pp. 1681–1688. ACM, New York (2009)
9. Paige, R.F.: Case study: Airport crisis management. In: Large Scale Complex IT Systems Workshop (2010)
10. Read, M.: Statistical and Modelling Techniques to Build Confidence in the Investigation of Immunology through Agent-Based Simulation. PhD thesis, University of York (2011)
11. Saltelli, A., Chan, K., Scott, E.M. (eds.): Sensitivity Analysis. Probability and Statistics. Wiley (2000)
12. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF Eclipse Modeling Framework, 2nd edn. The Eclipse Series. Addison-Wesley (2009)
13. Williams, J.R., Paige, R.F., Kolovos, D.S., Polack, F.A.C.: Search-based model driven engineering. Technical Report YCS-2012-475, Department of Computer Science, University of York (2012)
14. Ziv, H., Richardson, D.J., Klösch, R.: The uncertainty principle in software engineering. In: Proc. 19th International Conference on Software Engineering, ICSE 1997 (1997)

Modeling and Analysis of CPU Usage in Safety-Critical Embedded Systems to Support Stress Testing

Shiva Nejati[1], Stefano Di Alesio[1,2], Mehrdad Sabetzadeh[1], and Lionel Briand[1,2]

[1] SnT Center, University of Luxembourg, Luxembourg
[2] Certus Software V&V Center, Simula Research Laboratory, Norway
{shiva.nejati,stefano.dialesio,mehrdad.sabetzadeh, lionel.briand}@uni.lu

Abstract. Software safety certification needs to address non-functional constraints with safety implications, e.g., deadlines, throughput, and CPU and memory usage. In this paper, we focus on CPU usage constraints and provide a framework to support the derivation of test cases that maximize the chances of violating CPU usage requirements. We develop a conceptual model specifying the generic abstractions required for analyzing CPU usage and provide a mapping between these abstractions and UML/MARTE. Using this model, we formulate CPU usage analysis as a constraint optimization problem and provide an implementation of our approach in a state-of-the-art optimization tool. We report an application of our approach to a case study from the maritime and energy domain. Through this case study, we argue that our approach (1) can be applied with a practically reasonable overhead in an industrial setting, and (2) is effective for identifying test cases that maximize CPU usage.

1 Introduction

Many safety-critical systems, e.g., those in the avionics, railways, and maritime and energy domains, are increasingly relying on embedded software for control and monitoring of their operations. The safety-related software components of these systems are often subject to software safety certification, whose goal is to provide an assurance that the components are deemed safe for operation. Software safety certification needs to take into account various non-functional constraints that govern how software should react to its environment, and how it should execute on a particular physical platform [1]. These constraints, among others, include deadlines, throughput, jitter, and resource utilization such as CPU and memory usage [2,3]. Reasoning about these constraints is becoming more complex in large part due to the multi-threaded design of embedded software, the shift towards multi-core and decentralized architectures for execution platforms, and the increasing complexity of real-time operating systems.

In this paper, we concentrate on a particular type of non-functional constraints, namely *CPU usage*, and provide a framework to derive test cases to verify that the CPU time used by a set of concurrent threads running on a multi-core CPU does not exceed a given limit, even under the worst possible circumstances. Keeping CPU usage low in safety-critical applications is not merely for general quality reasons, but rather an important safety precaution since with CPU usage above a certain threshold, the system may fail to respond in a timely manner to safety-critical alarms. For example, if a fire and gas monitor is starved of CPU time due to CPU overload, it can have a delayed or miss response to a fire or gas leak with potentially serious consequences. As a result, test cases that can stress the system to maximize CPU usage are crucial for certification of safety-critical systems. Our approach to CPU usage modeling and analysis is driven by two main considerations:

1) Explicit modeling of time. Reasoning about CPU usage requires an explicit notion of time. Logic-based languages used for reasoning about concurrent software, e.g., most temporal logics, do not capture time explicitly [4]. Hence, while they can be used to reason about relative orderings of concurrent tasks, they cannot be used for computing constraints involving actual time values. We instead follow the common practice in standard languages such as the UML Profile for Modeling and Analysis of Real-Time and Embedded Systems (MARTE) [5], where time should be explicitly expressed.
2) Search-based optimization. Our goal is to find testing scenarios that maximize CPU usage by a set of parallel threads running on a multi-core platform. We refer to this testing activity as *stress testing* [6]. We characterize the stress test scenarios, i.e., test cases, by *environment-dependent parameters* of the embedded software, e.g., the size of time-delays used in software to synchronize with hardware devices or to receive feedback from the hardware devices. To stress test the system, we choose the environment parameters in such a way that the system is pushed to use the maximum amount of CPU. Finding such stress test cases requires to search the possible ways that a set of real-time tasks can be executed according to the scheduling policy of their underlying real-time operating system. In our approach, the search for stress test cases is formalized using a constraint optimization model that includes (1) a set of constraints describing a declarative representation of the tasks, their timing constraints and priorities, and the platform-specific information, and (2) a cost function that estimates CPU usage. Our approach for deriving test cases, while it may not result in provable system safety arguments, can always provide test cases within a time budget and given a (potentially partial) set of declarative constraints characterizing the embedded software and its environment.

Contributions of This Paper. We develop an automated tool-supported solution for deriving test cases exercising the CPU usage requirements of a set of embedded parallel threads running on a *multi-core* CPU. Specifically, we make the following contributions:

– A conceptual model that captures, independently from any modeling language, the abstractions required for analyzing CPU usage requirements in embedded systems (Section 3.1). To simplify the application of our conceptual model in standard Model-Driven Engineering (MDE) tools, we provide a mapping between our conceptual model and UML/MARTE (Section 3.2).
– Casting of the CPU usage problem as a constraint optimization problem over our conceptual model (Section 4). If done effectively, this enables the use of mature constraint optimization technologies that have not been used so far to address this type of problems. We provide an implementation of our approach in the COMET tool [7] (Section 4). COMET comes with efficient implementations of various search algorithms and is widely used in Operations Research for solving optimization problems.

– An industrial case study from the maritime and energy domain concerning safety-critical IO drivers. IO drivers are some of the most complex software components in the maritime and energy domain with sophisticated concurrent designs and many real-time properties (Sections 2). Our case study shows that our approach (1) can be applied with a practically reasonable overhead, and (2) can identify test cases maximizing CPU usage within time constraints (Section 5).

Structure of the Paper. In Section 2, we motivate our work using a specific industrial context. In Section 3, we present our conceptual model and show how it can be mapped to UML/MARTE. In Section 4, we formulate CPU usage as a constraint optimization problem. We provide an evaluation of our approach in Section 5. We compare with related work in Section 6, and conclude the paper in Section 7.

2 Motivating Case Study

We motivate our work with a class of safety-critical I/O drivers from the maritime and energy industry. These drivers are used in fire and gas monitoring applications, and their overall objective is to transfer data between control modules, and hardware devices, i.e., detectors. The variations between different drivers are mainly due to the different communication protocols that they implement in order to connect to different types of detectors built by different vendors. Such drivers are common in many industry sectors relying on embedded systems.

One of the main complexity factors in drivers is that they need to bridge the timing discrepancies between hardware devices and software controller modules. Hence, their design typically consists of several parallel threads communicating in an asynchronous manner to enable smooth data transfer between hardware and software. Often, several execution time constraints are included in the drivers' requirements to ensure that the flexibility in the design of the drivers does not come at the cost of overusing the resources of the execution platform. An example of such constraints is the following: *An I/O driver shall, under normal conditions, not impact heavily on the CPU time. When only one driver instance is running, the idle CPU time shall be above 80%.*

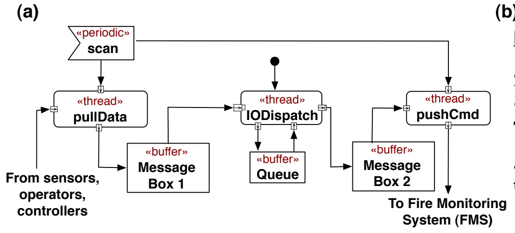
There are three important context factors from the case study influencing our formulation of the CPU usage problem in this paper:

1. Different instances of a given driver are independent in the sense that they do not communicate with one another and do not share memory.
2. The purpose of the CPU usage constraints is to enable engineers to estimate the number of driver instances of a given monitoring application that can be deployed on a CPU. These constraints express bounds on the amount of CPU time required by *one* driver instance. Our analysis in this paper, therefore, focuses on individual driver instances. The independence of the drivers (first factor above) is key to being able to localize CPU usage analysis to individual instances in a sound manner.
3. The drivers are not *memory-bound*, i.e., the CPU time is not largely affected by the low-bound memory allocation activities such as transferring data in and out of the disk and garbage collection. To ensure this, the partner company (over-) approximates the maximum memory required for each driver instance by multiplying the

number of detectors connected to the driver instance and the maximum size of data sent by each detector. Execution profiles at the partner company indicate that the drivers are extremely unlikely to exceed this limit during their lifetime.

Figure 1(a) illustrates an activity diagram capturing the overall architecture of the I/O drivers we focus on. Each driver consists of three parallel threads: Two of these threads, **pullData** and **pushCmd**, are executed periodically upon receipt of a trigger, i.e., **scan**. The **IODispatch** thread, however, is enclosed within an infinite (unconditional) loop. The **pullData** thread receives (pulls) data from sensors/human operators/control modules, then puts the data in an appropriate command form, and finally sends it to the **IODispatch** thread through a shared memory storage. The **pushCmd** thread receives commands from the **IODispatch** thread via another memory storage, and transfers them to the Fire Monitoring Systems (FMS). The memory storages in Figure 1 are shared only between the threads of one driver instance.

(a)



(b)

Data Transfer Scenario:

1. The system retrieves commands from its detectors
2. The system stores the commands in Queue
3. The system sleeps for 50 msec
4. The system reads the first command (or the command with the highest priority) from Queue
5. The system puts the command in Message Box 2 to be read by FMS and then goes to step **1**

Fig. 1. Driver case study: (a) Overview of the drivers' architecture. (b) The CPU intensive scenario of drivers. This scenario is subject to stress testing regarding CPU usage.

Drivers in our partner company have four modes of operation: maintenance, normal, initial and termination. Only the normal mode is critical with regard to CPU usage. In this mode, the connections with FMSs are established, and the *data transfer* scenario is enabled. The data transfer scenario of the drivers for an example communication protocol is shown in Figure 1(b). It describes a uni-directional communication where some delay is injected between the commands in each transmission iteration to ensure that the commands are received by FMSs at a slow enough rate so that the FMS can process them. To show that drivers satisfy their CPU usage requirement, we focus on data transfer scenarios of drivers only because other driver scenarios are not CPU intensive

The drivers run on a CPU with three separate cores. The operating system used is VxWorks [8] – a *Real-Time Operating System (RTOS)*. RTOSs share many features with general-purpose operating systems, but in addition have specialized kernels and a process scheduler that takes into account real-time constraints [3]. The installation of VxWorks in our study uses a *fixed priority preemptive scheduler*. This scheduling policy does not allow for a lower-priority task to execute while a high-priority task is ready. The **pullData**, **pushCmd**, and **IODispatch** threads communicate in an asynchronous way through buffers implemented using message queuing utilities provided by VxWorks. In the driver implementation, all accesses to the shared buffers are properly protected by semaphores and can potentially be blocking.

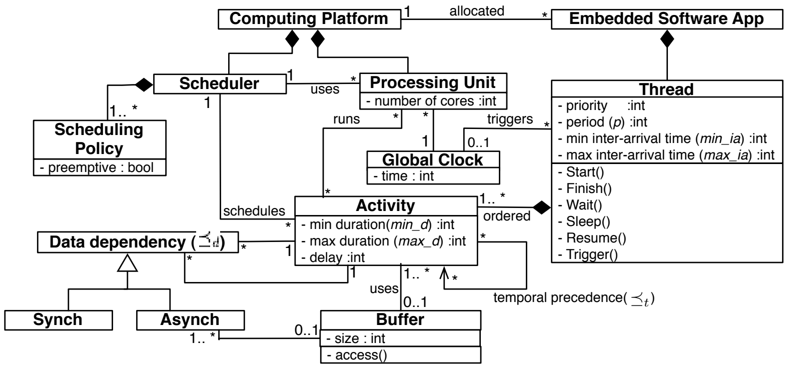


Fig. 2. The conceptual model characterizing the information required for CPU usage analysis

To estimate the CPU time used by a driver, we need to first include in our design models timing information such as how long it takes for the threads to run and their frequency. We then need to specify how the CPU usage can be characterized based on the input timing information. This requires capturing the *concurrent* dependencies between the threads, how these threads *communicate*, how the RTOS scheduler *preempt*s the threads, and how the threads can run on a *multi-core processor*. In the subsequent sections, we provide our solution that can address all these details.

3 Modeling Guidelines

In this section, we first provide a *conceptual model* that captures the timing abstractions necessary for analyzing CPU usage (Section 3.1). We then show how the abstractions are mapped to the UML/MARTE metamodel (Section 3.2).

3.1 Conceptual Model

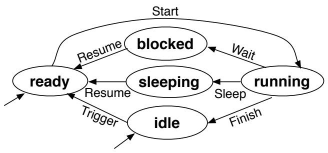
The conceptual model depicted in Figure 2 and explained below specifies the information required for analysis of CPU usage:

Thread. An embedded software application consists of a set $J = \{j_1, \ldots, j_n\}$ of parallel threads. A thread $j \in J$ can be *periodic* or *aperiodic*. Periodic threads, which are triggered by timed events, are invoked at regular intervals and as such their execution time is bounded by the (fixed) length of one interval, denoted $p(j)$ [9]. Any thread that is not periodic is called aperiodic. Aperiodic threads have irregular arrival times. In general, there is no limit on the execution time of an aperiodic thread, but one can optionally have a minimum inter-arrival time $min\_ia(j)$ and a maximum inter-arrival time $max\_ia(j)$ indicating the minimum and maximum time intervals between two consecutive arrivals of the event triggering the thread, respectively [10].

A common use of periodic threads is when we need to send/receive data regularly (e.g., **pullData** and **pushCmd** in Figure 1). In contrast, aperiodic threads are often used to process asynchronous events/communications (e.g., **IODispatch** in Figure 1).

Each periodic or aperiodic thread has a priority, denoted $priority(j)$ that is used by a priority-based scheduler to determine which thread should be running at each time.

During its lifetime, a thread may perform the following lifecycle operations: (1) Start(): to start execution after having been assigned to a CPU core by the scheduler. (2) Finish(): to complete its execution. (3) Wait(): to wait in order to synchronize with another thread or to acquire some resource. (4) Sleep(): to go to sleep. (5) Resume(): to indicate to the scheduler that it is ready to resume execution after a previous block or sleep period. (6) Trigger(): to indicate to the scheduler that it is ready to start a new execution in response to a new triggering event after having completed a prior round of execution. The above state machine shows the lifecycle of a typical thread. A thread consumes CPU time only when it is running.



Activity. An *activity* is a sequence of operations in a thread that can execute without needing to release the CPU until its very last operation. The only situation where an activity releases the CPU is when it is preempted by a (preemptive) scheduler so that the CPU can go to another activity belonging to a thread that has a higher priority. In other words, an activity is a sequence of operations that has Wait(), Sleep() or Finish() as its last operation but nowhere else in the sequence. Each thread $j \in J$ has a sequence $(a_1, \ldots, a_{m_j})$ of activities. We denote the set of all activities within a software application by A. Each activity a has an *estimated* minimum and maximum execution time denoted by $min\_d(a)$ and $max\_d(a)$, respectively. For each activity a, we denote the thread that owns that activity by $thread(a)$. Each activity a has a priority, inherited from its owning thread, i.e., $priority(thread(a))$. When activities end with a Sleep() operation, they are followed by a period of sleeping time. We use $delay(a)$ to denote the duration of the sleeping time. Activities of parallel threads can be related to one another by two kinds of relations: temporal precedence and data dependency.

Temporal Precedence. When an activity a must be executed prior to an activity a', i.e., a is a prerequisite of a', we say that a (temporally) precedes a' and denote this by $a \preceq_t a'$ where $\preceq_t \subseteq A \times A$. Temporal precedence relates activities belonging to the same thread only.

We unroll the loops by copying the loop body a certain number of times. The number of unrollings can be chosen as an input parameter, and depends on the amount of time during which we choose to observe execution of the threads in our case study (see the notion of observation time interval discussed in Section 4). Temporal precedence then indicates that the activities in the first copy of the loop body precede those in the second copy, and those in the second copy precede those in the third copy, and so on – an example is given in Section 3.2. We also ensure that the last activity in the ith copy of the loop is followed by the first activity in the $(i + 1)$st copy of the loop (see loop constraints in Section 4).

Data Dependency. An activity a may depend on another activity a' because a requires some data that is computed by a'. We denote this relation by $a' \preceq_d a$ where $\preceq_d \subseteq A \times A$.

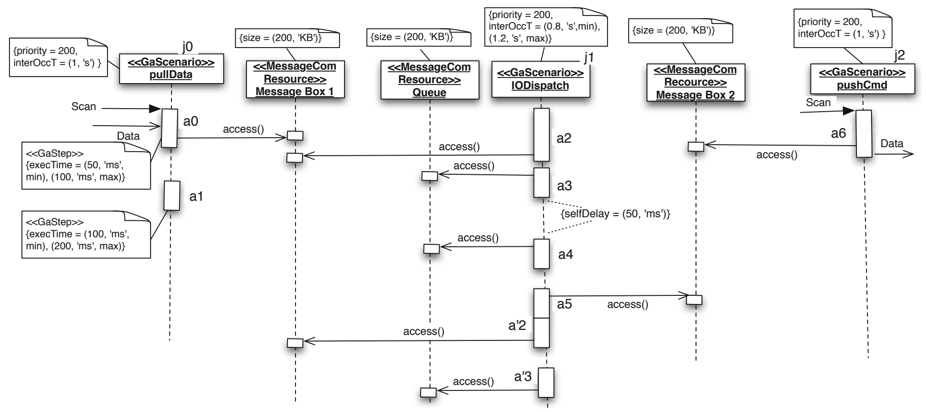

{priority = 200, interOccT = (1, 's') }
j0
<<GaScenario>> pullData
{size = (200, 'KB')}
<<MessageCom Resource>> Message Box 1
{size = (200, 'KB')}
<<MessageCom Resource>> Queue
{priority = 200, interOccT = (0.8, 's',min), (1.2, 's', max)}
j1
<<GaScenario>> IODispatch
{size = (200, 'KB')}
<<MessageCom Recource>> Message Box 2
{priority = 200, interOccT = (1, 's') }
j2
<<GaScenario>> pushCmd
Scan
Data
a0
access()
<<GaStep>> {execTime = (50, 'ms', min), (100, 'ms', max)}
a1
<<GaStep>> {execTime = (100, 'ms', min), (200, 'ms', max)}
a2
access()
a3
access()
{selfDelay = (50, 'ms')}
access()
a4
a5
access()
a'2
access()
a'3
access()
Scan
a6
Data
access()

Fig. 3. A sequence diagram capturing the data transfer scenario in Figure 1(b). The diagram is augmented with MARTE timing and concurrency stereotypes and attributes.

For any data-dependent pair of activities, we need to specify whether the communication is synchronous or asynchronous. The activities related by a data dependency relation may or may not belong to the same thread.

Buffer. Asynchronous communications may or may not use buffers. Buffer accesses by activities are protected by semaphores, and are blocking. Hence, each buffer access within an activity implies an implicit Wait() operation and indicates the last operation of that activity. Therefore, each activity can be related to at most one buffer. Also, at most one activity can access a given buffer at any point in time. The time an activity is blocked, waiting for a shared buffer, is determined by our scheduling constraints and is zero when the buffer is not locked by any other activity (see Section 4).

Computing Platform and Global Clock. In addition to information about the software application itself, we need information about the characteristics of the underlying computing platform. In particular, we require knowledge of the number of CPU cores, denoted c, which indicates the maximum number of parallel activities that the CPU can host. We further need to know whether the scheduling policy used by the real-time scheduler is preemptive or non-preemptive. Lastly, we need a (real-time) clock to model time-based events/triggers.

3.2 Mapping to UML/MARTE

In this section, we demonstrate how the abstractions in Figure 2 are mapped to the UML/MARTE metamodel. This mapping shows the feasibility of extracting the abstractions required for CPU usage analysis from standard modeling languages supported by industry strength tools. We begin by first describing the abstractions that are already present in UML. We then show how missing timing and concurrency aspects can be mapped to MARTE.

In UML, Active Objects have their own threads of control, and can be regarded as concurrent threads [11]. Active objects in UML sequence diagrams can be associated to lifelines with several Execution Specifications that match activities in our conceptual model in Figure 2. The notation for describing synch and asynch communication already exists in UML sequence diagrams where an Occurrence Specification indicates a sending or a receiving of a message.

Figure 3 shows a sequence diagram capturing the data transfer scenario described in Figure 1(b). As shown in the figure, each thread in the driver application has an object with lifeline. Similar to threads and as shown in Figure 3, buffers correspond to passive objects in UML, and can be represented using lifelines as well.

We represent each activity of a thread using an *activation* or an *execution specification* (a thin box on the lifeline of a thread that shows the interval of time that the thread is active). Each thread lifeline is made up of a sequence of activations corresponding to its activities: thread j_0 is composed of activities a_0 and a_1; thread j_1 is composed of a_2, a_3, a_4, a_5, a'_2, and a'_3; and thread j_2 is composed of a single activity, a_6. Activities a'_2 and a'_3 are repetitions of a_2 and a_3, respectively. Due to space reasons, we have not shown any repetition of a_4 or a_5, or any further repetitions of a_2 or a_3. As mentioned earlier, we use constraints to ensure that a_5 is followed by a'_2 (see Section 4).

The order of activations on a thread lifeline implies the temporal precedence between activities of that thread. For every pair a, a' of activities, $a \preceq_t a'$ if a and a' belong to the same thread j and a precedes a' as indicated by the lifeline of j. For example, in Figure 3, we have $a_0 \preceq_t a_1$ and $a_2 \preceq_t a_3 \preceq_t a_4 \preceq_t a_5 \preceq_t a'_2 \preceq_t a'_3$.

In sequence diagrams, a synchronous message from an activity a to an activity a' is shown using a solid arrow with a full head; an asynchronous message is shown by a solid arrow with a sticky head. Synchronous communication is blocking and does not require a buffer by default. I.e., the sending activity must wait until the receiving activity is ready to receive messages. Asynchronous communications may or may not use buffers. In our case study, and hence in the sequence diagram of Figure 3, all communications are asynchronous and buffered. Based on the information obtained from the drivers' design, for the activities in Figure 3, we have $a_0 \preceq_d a_2$, $a_3 \preceq_d a_4$, and $a_4 \preceq_d a_6$.

Even though UML sequence diagrams can already capture several concepts in the Embedded Software Application package in Figure 2, the schedulability concepts, and the timing and concurrency attributes in that figure do not have appropriate counterparts in UML. These concepts are captured by extensions of UML, in particular MARTE, which is geared towards both the real-time and embedded system domains.

MARTE provides a Generic Quantitative Analysis Modeling (GQAM) sub-profile intended to provide a generic framework for collecting information required for performance and schedulability analysis. The domain model of this sub-profile includes two key abstractions that closely resemble our notions of thread and activity respectively: *Scenario* and *Step*. Step is a unit of execution, and Scenario is a sequence of steps. We map $\langle\langle$GaStep$\rangle\rangle$ (resp. $\langle\langle$GaScenario$\rangle\rangle$) which is a stereotype representing the notion of Step (resp. Scenario) in the domain model of GQAM to our notion of activity (resp. thread). These two stereotypes can be applied to a wide set of behaviour-related elements in UML 2.0 metamodel, and in particular, to UML sequence diagrams. We also map our notion of buffer to $\langle\langle$MessageComResource$\rangle\rangle$ which represents artifacts for communicating messages among concurrent resources.

MARTE includes a list of measures that are widely-used for analysis of real-time properties of embedded systems. The majority of these are applied to Steps and

Scenarios, creating their sets of quantitative attributes. The top two rows of Table 1 show the mapping between our timing attributes to those of ⟨⟨GaScenario⟩⟩ and ⟨⟨GaStep⟩⟩ in MARTE. For example, we map interOccTime, the time interval between two successive occurrences of scenarios, to period or (max/min) inter-arrival times of threads, and execTime, the execution time of a step, to (min/max) duration of activities. Note that both of these measures can be specified either as single values or as max/min intervals. As an example, the sequence diagram in Figure 3 is augmented with the timing and concurrency stereotypes and attributes from MARTE.

We identified only one discrepancy in our mapping: In MARTE, individual steps have a *priority* attribute, indicating the priority of the step on their processing host, but this priority attribute does not directly apply to scenarios. At the implementation level, however, it is common to define priorities for threads rather than for steps within the threads. Hence, we specified priorities at the level of threads (Scenarios) and not for individual activities (Steps). In our mapping, we assume that the steps within a scenario have the same priority that carries over to the scenario which is a composite entity.

Information about the computing platform in Figure 2 is not captured on the sequence diagram but can be represented using MARTE streotypes applied to UML class diagrams. The GRM::Scheduling sub-profile already includes the schedulability concepts of Figure 2, i.e., ⟨⟨Scheduler⟩⟩ and ⟨⟨SchedulingPolicy⟩⟩. Finally, we map processing units in Figure 2 to ⟨⟨ComputingResource⟩⟩ from GRM::ResourceType sub-profile, and the global clock to ⟨⟨LogicalClock⟩⟩ from TimeAccesses::Clocks sub-profile. The latter allows us to define regular triggers/events in RTOSs, e.g., scan in Figure 1.

Table 1. Mapping abstractions in Figure 2 to UML/MARTE

	Concept	MARTE StereoType/attributes	MARTE Sub-Profile
Embedded Soft. App.	Thread - priority - period, - (min/max) inter-arrival time	«GaScenario» - *priority: NFP_Integer - interOccT: NFP_Duration[*] «TimedConstraint»	GQAM:: GQAM_Workload TimedConstraints
	Activity - (min/max) duration - delay	«GaStep» - execTime: NFP_Duration[*] - selfDelay: NFP_Duration[*]	GQAM:: GQAM_Workload
	Buffer - size - access()	«MessageComResource» - messageSizeElements: ModelElement [0..*] - sendServices/receiveServices: BehavioralFeature [0..*]	SRM:: SW_Interaction
Comp. Plat.	Scheduler	«Scheduler»	GRM::Scheduling
	Scheduling Policy	«SchedulingPolicy»	GRM::Scheduling
	Processing Unit	«ComputingResource»	GRM::ResourceTypes
	Global Clock - scan	«LogicalClock» - clockTick	TimeAccesses::Clocks

4 CPU Usage Analysis through Constraint Optimization

Figure 4 shows an overview of our solution for CPU usage analysis using constraint optimization. Our solution has four main elements: (1) time and concurrency information, (2) scheduling variables, (3) objective functions, and (4) constraints characterizing schedulability algorithms.

Intuitively, given the input (time and concurrency information), the goal is to compute values for the scheduling variables such that: (a) the schedulability constraints are satisfied, and (b) an objective function is maximized or minimized depending on the problem at hand. One main advantage of approaching our objectives as a constraint optimization problem is that such computations can be performed using off-the-shelf constraint optimization tools. We ground our formulation of the CPU usage problem on the COMET tool. This choice is motivated mainly by the efficient implementation of complete search in COMET and its support for parallel search (see Section 5), which is used for the evaluation of our approach in this paper. Below, we discuss each of the four main elements of our solution and outline their implementation in COMET.

```
// (1) Time and concurrency information (Input)

range Threads = 0..n-1;  range Activities = 0..m-1;
int c = 3;                          // Number of cores
int p[Threads] = ..;                // Periods
int priority[Threads] = ..;         // Priority
  ....

// (2) Scheduling variables (Output)
var{int} start[Activities];                    // Actual start times
var{int} end[Activities];                      // Actual end times
var{int} active[Activities, T];                // Active matrix for individual time points
var{int} eligible_for_execution[Activities];   // Start times in ideal situation

// (3) Objective function (maximizing CPU usage)
maximize
  sum (a in Activities, t in T) (active[a, t])/ c* sizeofT    // CPU usage computation function

// (4) Constraints (characterizing schedulability algorithms)
subject to
{
   forall (a in Activities )
        post (end[a] < p[thread(a)]);  // An activity must end before the period of its thread
        post (active[a, start[a]] == 1);  // ...
        ....
}
```

Fig. 4. CPU usage as a constraint optimization problem. The full COMET implementation can be found at [12].

(1) Time and Concurrency Information. All the input data in Fig 4 (part (1)) corresponds to the elements in our conceptual model in Section 3, and hence can be automatically extracted from the UML/MARTE models. We implement this information using COMET pre-defined data types. We define the notion of *observation time interval* as the time we spend observing the thread executions and denote it by T.

(2) Scheduling Variables. These variables specify a *schedule* for a given set of activities A during an observation time interval T. Specifically, a schedule specifies the actual start time $start(a)$ and the actual end time $end(a)$ for every activity $a \in A$. We denote the duration of an activity a by $d(a)$ (not to be confused with $delay$ which represents the delay time after activities). In non-preemptive scheduling, $d(a)$ is simply defined as the length of the interval between $start(a)$ and $end(a)$, i.e., $d(a) = end(a) - start(a)$. But for preemptive scheduling, a can be interrupted during its execution, and hence, it may not be executing continuously. Therefore, we define $d(a)$ to be a set variable representing the set of time points at which a executes. In addition, for an activity a and time point t, we define a function $active(a, t)$ as follows:

$$active(a,t) = \begin{cases} 1 \text{ if } a \text{ executes at time } t \\ 0 \text{ otherwise.} \end{cases}$$

To account for multi-core scheduling, we define a variable $eligible\_for\_execution(a)$, or $efe(a)$ for short, that returns the earliest possible time that a can start running assuming that the number of cores is infinite, and hence, there is no bound on the number of activities that can run in parallel.

We implement scheduling variables as COMET variables with a specific type and a finite range. Values for these variables are computed within a given observation time interval T. In our formulation, we have added a new dimension to the scheduling variables to compute these variables for multiple execution rounds, where the number of rounds is determined by T (not shown in Figure 4 to avoid clutter, see [12]).

(3) Constraints. We use first-order logic to express the constraints. All the constraints are provided below. We omitted constraint formulations when the formulations were straightforward or lengthy. The complete formulations are available at [12].

▷ **Well-formedness (sanity rules)**

- Every activity must finish before the period of its corresponding thread elapses and cannot start before the start time of that thread.
- The number of time points at which an activity is running is bounded by its min/max duration.
- An activity starts running at its start time, ends just before its end time, and does not run before its start time or after its end time.

▷ **Loop Threads.** Consider activities $a_0^i, \ldots, a_q^i$ representing the activities of iteration i of a thread. Then, for every iteration i, we must have: $start(a_0^{(i+1)}) \geq end(a_q^i)$.

▷ **Temporal Precedence.** For every $a, a' \in A$ s.t. $a \preceq_t a'$, we have $start(a') - end(a) \geq delay(a)$. Note that $delay(a) = 0$ if a is not followed by a delay.

▷ **Synch/Asynch Communication.** For every $a, a' \in A$ s.t. $a \preceq_d a'$, if the communication is synchronous then we have $start(a') \geq end(a)$.

▷ **Buffer.** For every $a, a' \in A$ s.t. $a \preceq_d a'$, if the communication goes through a shared buffer then if $start(a) < start(a')$, then $start(a') \geq end(a)$. This is because a locks the shared resource during its execution. Also, if a and a' access the same buffer (but no data dependency relation is known between them), then a and a' cannot be active at the same time at any given time.

▷ **Multi-Core.** The number of running activities at every time point is less than or equal to the number of cores:

▷ **Scheduling Policy**

- Each activity can potentially be preempted: $\forall a \in A \cdot end(a) - start(a) \geq d(a)$.
- The earliest time an activity a can start ($efe(a)$) is after the arrival time of its corresponding thread and after the earliest termination time of all the activities *preceding* a. Here, precedence includes both temporal precedence ($\preceq_t$) and data dependency ($\preceq_d$) orderings.
- At any time, if there are two activities that can be scheduled for execution in parallel but only one is running, the one that is not running has a lower priority.

Amongst the constraints above, only the scheduling policy constraints have a context-specific nature and need to change according to the specific policy used in a given system. The remaining constraints are generic and be reused across different domains and applications.

(4) Objective Functions. Our objective is to find combinations of input values that can generate schedules that consume the CPU time most, and hence, are more likely to violate CPU usage requirements of the system. To capture high usage of CPU time, we define two alternative objective functions. The first one computes average CPU usage, is denoted by f_{usage}, and is defined as:

$$f_{usage} = \frac{\sum_{a \in A, t \in T} active(a,t)}{T \times c}$$

The summation $\sum_{a \in A, 0 \leq t \leq T} active(a,t)$ measures the total time points when at least one activity is running, and $T \times c$ is the total available time on all the cores. The second objective function, called *makespan*, measures the total length of the schedule. We denote this objective function by $f_{makespan}$ and define it as:

$$f_{makespan} = \max_{a \in A} end(a)$$

The $f_{makespan}$ function is the time it takes for all the activities in an application to terminate after the arrival time of the first thread in that application. Makespan is a common metric for measuring response time [7].

By maximizing either f_{usage} or $f_{makespan}$, we compute schedules that are more likely to violate the CPU usage requirements. Note that f_{usage} or $f_{makespan}$ are heuristics as their accuracy is bounded by the accuracy of the input data and the precision of our constraints in characterizing the domain. Therefore, these functions should not be viewed as measures for the actual CPU usage of the system. In Section 5, we discuss how the input values maximizing these functions can be used to generate test cases for CPU usage requirements.

5 Evaluation

The main goal of our evaluation is to investigate whether our technique can effectively help engineers in deriving test cases for CPU usage requirements. The practical usefulness of our approach depends on (1) whether the input to our approach can be provided with reasonable overhead, and (2) whether the engineers can utilize the output of our analysis to derive test cases that can maximize CPU usage.

(i) Prerequisite and Overhead. As discussed before, the information required for CPU usage estimation is captured by the conceptual model in Figure 2. To gather this information, we first built UML sequence diagrams for the IO drivers in our partner company using the existing design and implementation of the drivers. The resulting sequence diagrams were iteratively validated and refined in collaboration with the lead engineer of the IO drivers. Sequence diagrams are popular for visualizing concurrent multi-threaded interactions and are intuitive to most developers as was confirmed in our industry collaboration [13].

The quantitative elements in Figure 2 for our case study were obtained as follows: The values for priority and period of the threads, and the size of the buffers were extracted from the certification design documents and the IO driver code. The min/max inter-arrival times of **IODispatch**, which is an aperiodic thread, and the values for the min/max duration of activities in our case study were extracted from the performance profiling logs of the drivers. We created the sequence diagrams augmented with the timing information over 8 days, involving approximately 25 man-hours of effort. This was considered worthwhile as such drivers have a long lifetime and are regularly certified.

Finally, we obtained the computing platform information in Figure 2 from the RTOS configuration and hardware design documents. Note that, given the mapping in Table 1, any modeling development environment that supports UML/MARTE can be used to develop and manipulate our input design notation.

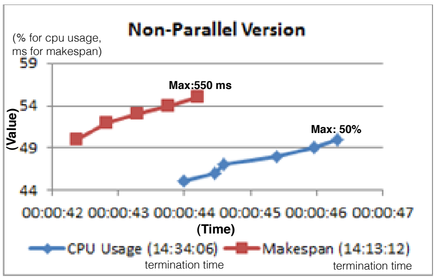
(ii) Test Case Derivation. The I/O drivers in our study are subject to certification based on the IEC61508 standard [14], which is one of the most detailed and widely-used functional safety standards. It specifies 4 levels of safety, called *Safety Integrity Levels (SILs)*. SIL1 is the lowest and SIL4 is the highest level. The drivers in our study need to be compliant to IEC61508 up to SIL2 or SIL3 depending on the context of their application. Stress testing (subjecting the system to harsh inputs with the intention of breaking it [6]) is classified by IEC61508 as “Recommended” for SILs 1-2 and “Highly Recommended” for SILs 3-4. “Highly Recommended” techniques/measures are often seen as “mandatory” by the certifiers, unless the supplier provides a convincing argument as to why a highly recommended technique/measure does not apply. Subsequently, the engineers in our partner company needed to stress test the drivers (mandatory for SIL3 deployments).

We characterize the stress test cases in our case study by the delay times that (potentially) follow execution of activities, i.e., the delay attribute of the activity class in Figure 2. IO drivers are instantiated in different environments with different numbers and kinds of detectors and FMSs. The delay times after the IO driver activities must be set to values that match the load and speed of the detectors and FMSs.

Based on the engineers’ intuition, large and complex hardware configurations, e.g., those consisting of several thousands of detectors, are more likely to violate the CPU usage requirements. To identify the suspicious hardware configurations, however, the analysis provided in this paper is necessary because the hardware configurations affect the delay times of the IO drivers activities, and subsequently, the CPU usage estimates.

For example, the size of the delay time at step 3 of the data transfer scenario in Figure 1(b) can heavily impact the CPU usage. Specifically, the delay time cannot be so small that **IODispatch** (Figure 1(a)) keeps the CPU busy for so long that it exceeds the given CPU usage requirement. Neither can the delay be too large, because then **pullData**, which is periodic, may miss its deadline. Specifically it may quickly fill up the **Message Box 1** buffer, which in turn causes **pullData** to be blocked and waiting for **IODispatch** to empty **Message Box 1**, which is now very slow due to a large delay time. As a result, **pullData** may not be able to terminate before its next scan arrival.

To derive stress test cases based on the delay times of the activities, in our formulation in Figure 4, we specify $delay$ as an output variable whose value is bounded within a range. The search then varies the values of these variables to maximize $f_{makespan}$ and f_{usage}. Those combinations that maximize our objective functions are more likely to stress the system to the extent that the CPU usage requirements are violated.



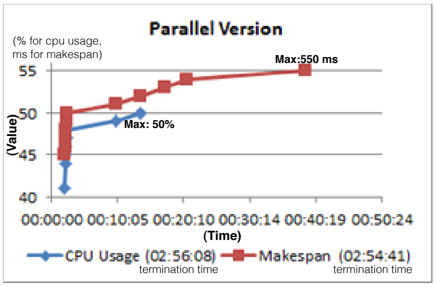


Fig. 5. The result of maximizing $f_{makespan}$ and f_{usage} (Section 4) for both parallel and non-parallel COMET implementations.

To perform the above experiment, we implemented the constraint optimization formulation in Figure 4 in COMET Version 2.1.0 [7]. We further used the native support of COMET for parallel programming to create a distributed version of our COMET implementation that divides the search work-load among different cores. To perform the experiment, we varied the observation time T from 1s to a few seconds and set the quantum time (i.e., the minimum time step that a scheduler may preempt activities) to 10 ms. The input model included eight activities belonging to three parallel threads.

Figure 5 shows the result of our experiment, maximizing $f_{makespan}$ and f_{usage} for both parallel and non-parallel COMET implementations. In both diagrams, the X-axis shows the time, and the Y-axis shows the size of $f_{makespan}$ in ms, and the percentage for f_{usage}. In our experiment, we used a complete (exhaustive) constraint solver of COMET, and ran it on a MacBook Pro with a 2.0 Ghz quad-core Intel Core i7 with 8GB RAM. As shown in the figure, the search terminated in both cases: after around 14 hours for the non-parallel version, and after around 2 hours and 55 min for the parallel version. The maximum computed values are: 50% for f_{usage}, and 550 ms for $f_{makespan}$. In the non-parallel case, the maximum result was computed after around 1 hour and 10 min for $f_{makespan}$, and 1 hour and 13 min for f_{usage}. No higher value was found in the remainder of the search which took more than 14 hours in total. In the parallel case, it took about 15 min to find the maximum for f_{usage}, and 40 min to compute that for $f_{makespan}$. To make sure these values were indeed maximum, the search continued until it terminated after 2 hours and 55 min.

In the end, we could compute maximum values for $f_{makespan}$ and f_{usage} in around 2 hours and 55 min using COMET's built-in support for parallel search. The values for the delay times maximizing $f_{makespan}$ and f_{usage} are candidates for stress test cases. We have recorded these values and have communicated them to our partner company.

Currently, the engineers at our partner company spend several days simulating their systems and monitoring the CPU usage without following a systematic strategy for stressing the systems to their CPU usage limits. We expect that by executing their systems based on the values produced by our approach, they can push the systems to states where the CPU usage is maximized and ensure that the input delay times remain within safe margins. The engineers at our partner company intend to test their system using our findings. Our experimental results and the input data values are available at [12].

6 Related Work

All approaches to performance engineering and schedulability analysis require a model of the time and concurrency aspects of the system under analysis [15]. Examples of such modeling languages include queuing networks [16], stochastic Petri nets [17], and stochastic automata networks [18]. Recently, there has been a growing interest in developing standardized languages to enhance the adoption of performance engineering concepts and techniques in the industry [19]. The most notable these languages is MARTE which extends UML with concepts for modeling and quantitative analysis of real-time embedded systems [5]. While a UML-profile, MARTE also encompasses the timing and concurrency abstractions in many other languages, e.g., Architecture Analysis and Design Language (AADL) [20]. As indicated by the mapping from our conceptual model to MARTE (Section 3.2), the abstractions we use in our work already exist in MARTE. However, MARTE is a large profile and by itself does not provide guidelines on what subset of it is required for a particular type of analysis. Our conceptual model can thus be viewed as a subset selection of MARTE, aimed specifically at CPU usage analysis.

The techniques for analysis of real-time systems can be divided into two general groups: (1) *Approaches based on real-time scheduling theory [9]*. These approaches estimate schedulability of a set of tasks through customized formulas and theorems that often assume worst case situations only such as worst case execution times, worst case response times, etc. Their results, therefore, can be too conservative because due to inaccuracies in estimating worst-case time values, the worst-case situations may never happen in practice. As a result, in general, we cannot rely on schedulability theory alone when dealing with analysis of real-time systems. Moreover, extending these theories to multi-core processors has shown to be a challenge [21,22].

(2) *Model-based approaches to schedulability analysis.* The idea is to base the schedulability analysis on a system model that captures the details and specifics of real-time tasks. This provides the flexibility to incorporate specific domain assumptions and a range of possible scenarios, not just the worst cases [23,24]. Most approaches that fall in this category, including our work, can deal with multi-core processors as well [24].

We formulate the problem of CPU usage analysis as a constraint optimization problem. Our work is inspired by *Job shop scheduling* – a well-known optimization problem where jobs are assigned to resources at particular times [25]. Job shop has several variations. Our formulation is closest to the *discrete resources* variant [7], but differs from it in that we need to specify scheduling policies used by the underlying RTOS.

Model checkers, in particular, real-time model checkers, e.g., UPPAAL [26], have been successfully used for the evaluation of time-related properties. Model checking is intended to be used for *verification*, i.e., to check if a given set of real-time tasks satisfy some property of interest. To adapt model checkers to checking different properties of real-time applications, the underlying state machines are built such that the question at hand can be formulated as a reachability query. For example, in [24], in order to analyze CPU-time usage, an *idle* state-machine is added to the set of interacting timed-automata to keep track of the CPU-time, and the error states were chosen so that their reachability could lead to violation of the CPU-time usage limit.

In our work, the property to be checked is captured by a *quantitative* objective function as opposed to a *boolean* reachability property, as in the case of model checking.

Therefore, our work is more geared towards *optimization* with applications in test-case generation rather than verification. One significant practical advantage is that, to adapt our formulation to check other kinds of real-time properties, it often suffices to change the objective function, and most of the constraints remain untouched. Lastly, to handle multiple cores, the existing UPPAAL-based solution in [22] assumes that a mapping between threads and cores is given a priori. Our approach in contrast does not require any mapping between threads and cores.

7 Conclusions and Future Work

We provided a practical approach to support derivation of stress test cases for the CPU usage requirements of concurrent embedded applications running on multi-core platforms. We proposed a conceptual model that captures, independently from any modeling notation, the abstractions required for analysis of CPU usage. We mapped our conceptual model onto the standard modeling language UML/MARTE to support the application of our approach in practice. We, then, formulated the CPU usage problem as a constraint optimization problem over our conceptual model, and implemented our formulation in COMET. Our evaluation on a real case study shows that our approach (1) can be applied with practically acceptable overhead, and (2) can identify test cases that maximize CPU usage. These (stress) test cases are crucial for building satisfactory evidence to demonstrate that no safety risks are posed by potential CPU overloads. Finally, we note that while our approach cannot provide proofs, we can always provide results given a (partial) set of declarative constraints and within a time budget.

Our solution draws on a number of context factors (Section 2) which need to be ascertained before our solution can be applied. While the generalizability of these factors need to be further studied, we have found the factors to be commonplace in many industry sectors relying on embedded systems. In the future, we plan to perform larger case studies to better evaluate the generalizability and scalability of our approach and experiment with other search methods, in particular, meta-heuristic search methods and hybrid approaches combining complete and meta-heuristic search strategies.

Acknowledgments. We thank Bran Selic for his useful comments on an earlier draft of this paper. We thank the research council of Norway for partially funding this work. L. Briand was supported by a FNR PEARL grant.

References

1. Jackson, D., Thomas, M., Millett, L. (eds.): Software for Dependable Systems: Sufficient Evidence? National Academy Press (2007)
2. Henzinger, T.A., Sifakis, J.: The Embedded Systems Design Challenge. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 1–15. Springer, Heidelberg (2006)
3. Lee, E., Seshia, S.: Introduction to Embedded Systems: A Cyber-Physical Systems Approach (2010), `http://leeseshia.org`
4. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)

5. A UML profile for MARTE: Modeling and analysis of real-time embedded systems (May 2009)
6. Beizer, B.: Software testing techniques, 2nd edn. Van Nostrand Reinhold (1990)
7. Hentenryck, P.V., Michel, L.: Constraint-Based Local Search. The MIT Press (2005), www.dynadec.com
8. Wind River VxWorks (2009), http://www.windriver.com/products/vxworks/
9. Liu, J.W.S.: Real-time systems. Prentice Hall (2000)
10. Sprunt, B.: Aperiodic task scheduling for real-time systems. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA (1990)
11. OMG: The Unified Modelling Language. Version 2.1.2 (2007), http://www.omg.org/spec/UML/2.1.2/
12. Alesio, S.D.: The CPU usage constraints in COMET. http://home.simula.no/~stefanod/comet.pdf
13. Harel, D., Marelly, R.: Specifying and executing behavioral requirements: the play-in/play-out approach. Software and System Modeling 2(2), 82–107 (2003)
14. IEC 61158: industrial communication networks - fieldbus specifications. International Electrotechnical Commission (2010)
15. Cortellessa, V., Marco, A.D., Inverardi, P.: Model-Based Software Performance Analysis. Springer (2011)
16. Lazowska, E., Zahorjan, J., Graham, S., Sevcik, K.: Quantitative system performance computer system analysis using queueing network models. In: Int. CMG Conference, pp. 786–788 (1984)
17. Marsan, M., Balbo, G., Conte, G., Donatelli, S., Franceschinis, G.: Modelling with generalized stochastic petri nets. SIGMETRICS Performance Evaluation Review 26(2), 2 (1998)
18. Plateau, B., Atif, K.: Stochastic automata network for modeling parallel systems. IEEE Trans. Software Eng. 17(10), 1093–1108 (1991)
19. Petriu, D.: Software Model-based Performance Analysis. John Wiley & Sons (2010)
20. Hudak, J., Feiler, P.: Developing AADL models for control systems: A practitioner's guide (October 2006)
21. Bertogna, M.: Real-Time Scheduling Analysis for Multiprocessor Platforms. PhD thesis, Scuola Superiore Sant'Annna, Pisa (2007)
22. David, A., Illum, J., Larsen, K., Skou, A.: In: Model-Based Framework for Schedulability Analysis Using UPPAAL 4.1, pp. 93–119. CRC Press (2010)
23. Briand, L., Labiche, Y., Shousha, M.: Using genetic algorithms for early schedulability analysis and stress testing in real-time systems. Genetic Programming and Evolvable Machines 7(2), 145–170 (2006)
24. Mikucionis, M., Larsen, K., Nielsen, B., Illum, J., Skou, A., Palm, S., Pedersen, J., Hougaard, P.: Schedulability analysis using UPPAAL: Herscehl-Planck case study. In: Proceedings of 4th International Symposiumo on Leveraging Applications of Formal Methods, Verification and Validation; track: Quantitative Verification in Practice (2010)
25. Applegate, D., Cook, W.: A computational study of the job-shop scheduling problem. INFORMS Journal on Computing 3(2), 149–156 (1991)
26. Behrmann, G., David, A., Larsen, K.G.: A Tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)

Weaving-Based Configuration and Modular Transformation of Multi-layer Systems[⋆]

Galina Besova[⋆⋆], Sven Walther, Heike Wehrheim, and Steffen Becker

Department of Computer Science
University of Paderborn
33098 Paderborn, Germany
{besova,swalther,wehrheim,stbecker}@mail.upb.de

Abstract. In model-driven development of multi-layer systems (e.g. application, platform and infrastructure), each layer is usually described by separate models. When generating analysis models or code, these separate models first of all need to be linked. Hence, existing model transformations for single layers cannot be simply re-used.

In this paper, we present a *modular* approach to the transformation of multi-layer systems. It employs *model weaving* to define the interconnections between models of different layers. The weaving models themselves are subject to model transformations: The result of transforming a weaving model constitutes a *configuration* for the models obtained by transforming single layers, thereby allowing for a re-use of existing model transformations. We exemplify our approach by the generation of analysis models for component-based software.

Keywords: Model weaving, multi-layer systems, model transformations.

1 Introduction

For multi-layer systems, to adhere to clean design principles like *separation of concerns* [28], a model-based design often derives separate models for the layers. These are possibly constructed by different domain experts and based on different meta-models. *Cloud Computing* applications provide us with specific sorts of multi-layer systems: the cloud computing stack of SaaS (*Software as a Service*), PaaS (*Platform as a Service*) and IaaS (*Infrastructure as a Service*) [3] allows to flexibly run application software on different platforms and infrastructures. Each layer might come with different models describing certain aspects of its structure and behavior.

For an analysis of the multi-layer system, the layers need to be linked to different extents: an analysis of non-functional properties, like performance, might also consider infrastructure models, while for functional analysis taking only application and platform models into account might suffice. Furthermore, the

[⋆] This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

[⋆⋆] Member of the International Graduate School "Dynamic Intelligent Systems".

flexibility to exchange the PaaS and IaaS layers might necessitate performing the analysis for various system configurations. Therefore the assumption of having one single fixed design model is no longer valid. Instead, we need to flexibly combine different models of different layers.

Models and model transformations are at the heart of model-driven development. The use of model transformations has been studied in diverse application areas, like code generation, model integration or model differencing. One particular application is the translation of design models into *analysis* models (e.g. [7,8,14,21]) to enable validation, simulation or verification. Analysis models serve as input to various analysis tools (e.g. model checkers [8] or simulation tools [7]).

In an MDE (model-driven engineering) framework, the generation of analysis models proceeds by the definition and execution of model transformations. Ideally, we aim at a *modular* and structured use of model transformations for our multi-layer system to handle different layers independently. A number of approaches have thus studied various forms of composition of model transformations [32], including higher-order transformations [29]. Modularity and re-use is also a key requirement for models in multi-layer systems: existing model transformations for separate layers should be re-used in the transformation of the overall system. For this, we do not compose the design models or model transformations directly. Instead, we transform layers models like application and platform models independently into compositional functional analysis models. The actual composition of the partial analysis models is guided by weavings of the design models. This does not only enable the re-use of existing transformations, but also the re-use of already generated functional analysis models and thus provides us with great flexibility during analysis.

Figure 1 shows our overall approach. Starting with partial models for several layers (possibly conforming to different meta-models) we first of all develop a *configuration* weaving model (Section 3.1) linking these models. The weaving

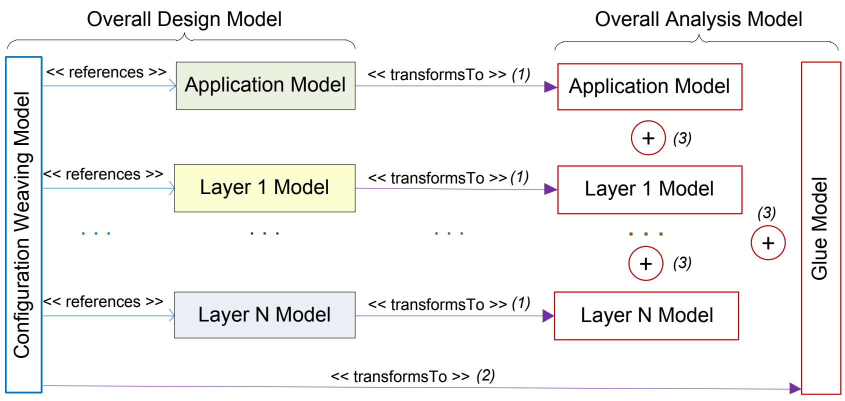


Fig. 1. Modular transformation approach for multi-layer systems

model conforms to a specific weaving meta-model and references models of the separate layers. Next, (possibly already existing) model transformations to the target models are applied on single layers (1). This gives us several partial target analysis models. Another part is generated out of the configuration weaving model in the form of a *glue* or adapter model (2), linking the separate analysis models (3) (Section 3.2). On the target side all analysis models conform to the same meta-model.

We exemplify our approach by the generation of functional analysis models for component-based software architectures to check for deadlocks and livelocks. For this, application and platform models are described within the Palladio Component Model (PCM) [7]. For the weaving model we employ AMW (Atlas Model Weaver [11]) and consequently ATL (Atlas Transformation Language [20]) for the transformations. Our target language for analysis is the process algebra CSP (Communicating Sequential Processes) [18], for which we have developed a custom meta-model. The compositional nature of CSP allows for a straightforward adapter generation: it is a CSP process linking the partial analysis models via synchronized parallel composition.

The paper is structured as follows: We will next give a short introduction into PCM and CSP, and alongside this will introduce our running example. Section 3 describes our overall approach of using weaving models as configurations for the system. In Section 4 we give an overview of the tools we used and some experimental results of the analysis. The following section then discusses related work and the last section concludes.

2 Background

This section outlines the foundations of both PCM and CSP to an extend needed for our approach. Both formalisms are introduced by means of our running example.

2.1 The Palladio Component Model

The *Palladio Component Model* (PCM, [7]) is a component model that supports modeling different views of a software system. As we focus on functional properties, we will only give an overview of the PCM views needed to specify interfaces, components and their connections. We will restrict the explanations roughly to the parts needed for our example.

In PCM a software component is specified by one or more *interfaces*. An interface contains a list of signatures[1]. The role of an interface (*provided* or *required*) is defined by its relation to a component. A *component* can be either a *basic* or a *composite* component. A basic component is implemented directly by a developer, while a composite component is build from other components using *connectors*. Interfaces and components are stored within a *repository*.

[1] Currently, we only consider method names, not parameters.

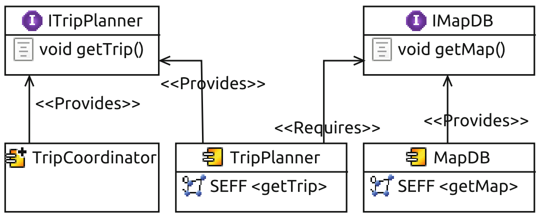


Fig. 2. PCM repository of the application layer

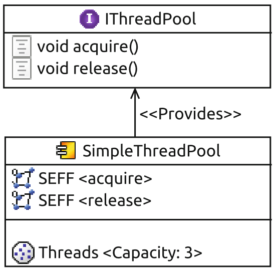


Fig. 3. PCM repository of the platform layer

Figure 2 shows a PCM repository. It contains some interfaces and components of a trip planning software that relies on a map database. The interface `ITripPlanner` is provided by a composite component `TripCoordinator` and a basic component `TripPlanner`. The basic component requires an interface `IMapDB`, which is provided by the basic component `MapDB`. Figure 4 shows the inner structure of the composite component `TripCoordinator`, which is assembled from the basic components using connectors. This represents an excerpt of our *application layer* model. In addition we provide parts of a model of the *platform layer*, namely a thread pool (Figure 3). The interface `IThreadPool` and the component `SimpleThreadPool`, which is capable of managing a limited number of threads (here 3), belong to this layer. New threads can be acquired and released using the methods `IThreadPool.acquire()` and `IThreadPool.release()` respectively.

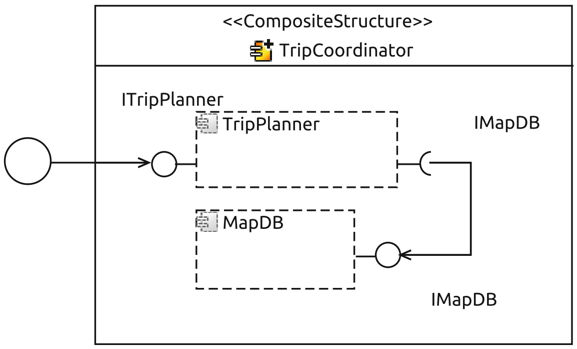


Fig. 4. Composite component `TripCoordinator`

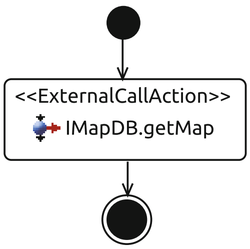


Fig. 5. SEFF of `TripPlanner.getTrip()` containing an *external call action*

To specify the behavior of components, and thus the dependencies between provided and required interfaces of a component, PCM provides *Resource Demanding Service Effect Specifications* (RDSEFFs, SEFFs for short). A SEFF contains a *start* and a *stop* action, actions to model internal computations, calls to required interface methods or the consumption of some countable resource, and the control flow.

Figure 5 shows `TripPlanner.getTrip()` as an example for a SEFF with an *external call action*. It specifies that during the execution of the method `TripPlanner.getTrip()` another external method, namely `IMapDB.getMap()`, is called. In the component `SimpleThreadPool` we see that the methods `acquire()` and `release()` also have SEFFs associated to them (see Section 3.2).

To summarize, we have now obtained an application layer model for the trip planning software and a model of a platform layer. Next, we are interested in analyzing these models with respect to functional properties, namely deadlock- and livelock-freedom. To this end, we need to generate input for some analysis tool. The tool we use here is FDR [2], which can analyze process specifications written in the process algebra CSP [18].

2.2 Communicating Sequential Processes

Communicating Sequential Processes (*CSP* [18]) is a process algebra that allows to formally specify the behavior of *processes* that are defined by sequences of atomic *events* they can participate in. Events in our setting will be method calls, sometimes split into start and end events. They can be equipped with inputs and outputs, e.g. `ev.v` denotes the occurrence of event `ev` with output value `v`, `ev?x` is an event `ev` reading some input into a variable `x`. The variable `x` can then be used in the rest of the process, for instance also as an output in `ev.x`.

Processes in CSP are build out of events using different composition operators. The termination of a process is indicated by the special event `SKIP` as in $\texttt{P} = \texttt{a} \rightarrow \texttt{SKIP}$ (here $\rightarrow$ is the prefix operator and stands for sequential composition). The recursively defined process $\texttt{THREAD} = \texttt{acquire} \rightarrow \texttt{release} \rightarrow \texttt{THREAD}$ participates in an infinitely alternating sequence of some events `acquire` and `release`. Processes can specify different potential behaviors by using the *choice* operator $\square$. For example, the process $\texttt{RPC} = (\texttt{send} \rightarrow \texttt{deliver} \rightarrow \texttt{RPC}) \;\square\; \texttt{FAIL}$ with $\texttt{FAIL} = \texttt{send} \rightarrow \texttt{FAIL}$ introduces an alternating sequence of `send` and `deliver` as well as an infinite loop, where the process can only participate in `send`.

If two processes can participate in the same events, it is possible to *synchronize* them. Two processes $\texttt{MAPDB} = \texttt{acquire} \rightarrow \texttt{getMap} \rightarrow \texttt{release} \rightarrow \texttt{MAPDB}$ and $\texttt{THREAD} = \texttt{acquire} \rightarrow \texttt{release} \rightarrow \texttt{THREAD}$ can be synchronized using the parallel operator $\|$ by specifying the *synchronization alphabet* as in $\texttt{SYSTEM} = \texttt{MAPDB} \underset{X}{\|} \texttt{THREAD}$ with $X = \{\texttt{acquire}, \texttt{release}\}$. A single process in this parallel composition can only execute `acquire` or `release`, if the other process can also participate in the same event.

Finally, the last feature of CSP important for our approach is the parameterization of processes. The process `TPOOL` given below, which represents a simplified result of transforming the component `SimpleThreadPool` into CSP, specifies a process that manages a limited number of *max* threads. The values of x and *max* are drawn from the natural numbers, with *max* being an arbitrary constant: $\texttt{TPOOL} = \texttt{TP}(\texttt{max})$, where

$$\begin{aligned}\texttt{TP}(0) &= \texttt{release} \rightarrow \texttt{TP}(1)\\ \texttt{TP}(\texttt{x}) &= (\texttt{acquire} \rightarrow \texttt{TP}(\texttt{x}-1)) \,\Box\, (\texttt{release} \rightarrow \texttt{TP}(\texttt{x}+1))\\ \texttt{TP}(\texttt{max}) &= \texttt{acquire} \rightarrow \texttt{TP}(\texttt{max}-1)\end{aligned}$$

By modeling the operations of software components as CSP events we can use CSP to specify the possible behavior and in particular the interaction of these components. Our objective is to analyze this behavior. As a first property, we consider *deadlock freedom*, i.e. we want to find out whether our trip planning software may deadlock. We will see that for this property an analysis of the application layer alone is not sufficient, as the thread pool model may influence deadlock-freedom. For other properties, other parts of the platform layer might be necessary. Thus we need a flexible way of combining different models for analysis.

3 Concept

This section describes the main ideas behind the proposed approach. Section 3.1 outlines the mechanism used to model the configuration of a multi-layer system. Section 3.2 provides an overview of our modular model transformation and composition approach. Both techniques are demonstrated on the case of generation of a CSP (Section 2.2) analysis model for the trip planning application example modeled in PCM (Section 2.1).

3.1 Weaving System Configuration

A configuration of a multi-layer system should contain information about individual layers as well as their interactions. In our example, the `TripCoordinator` belongs to the application layer model and the `SimpleThreadPool` constitutes the platform layer model. As an example of interactions between layers, we consider threads acquisition on the invocation of the `TripPlanner.getTrip()` and `MapDB.getMap()` methods with their later release on the method reply. To describe such configurations we employ the *model weaving* technique and use the *Atlas Model Weaver (AMW)* [11] for tool support.

The model weaving technique allows definition of various links between (meta-) model elements independent of the structure of the woven models. We use it to link actions of SEFFs (e.g. *start/stop* actions of `TripPlanner.getTrip()` and `MapDB.getMap()`) as *consumers* to the *resources* within other layers that they require (e.g. `SimpleThreadPool.acquire()`, `SimpleThreadPool.release()` within our platform layer). To model such resource usage links, we introduce a class *ResourceHandling* into our weaving meta-model.

Resource usages during the communication between components via external calls are modeled differently to emphasize their interaction support purpose. To model these usage links, we introduce a class *Communication* into our weaving meta-model. However, to keep our example simple, we do not include these links into our example configuration weaving model.

Figure 6 illustrates our weaving meta-model. It contains the mentioned link classes extending an abstract class *Usage* that models the association between the *resource* and *consumer* represented by *UsedModelElement* class. Class *UsageModel* represents a configuration model itself with *usages* and associations to the *UsedModel* class used for representing models of individual layers which are woven. Classes *UsedModel* and *UsedModelElement* encapsulate mechanisms for referencing woven models and model elements respectively.

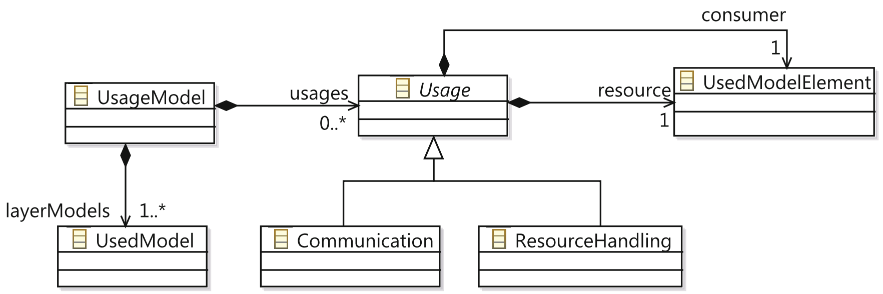
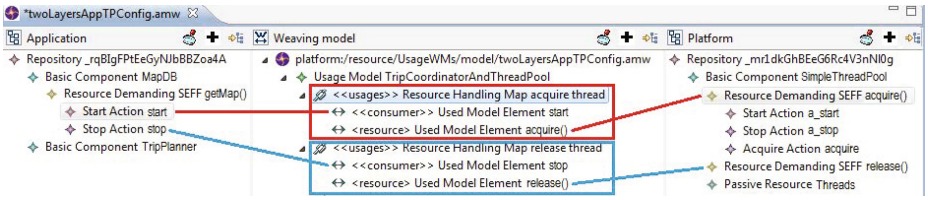


Fig. 6. Weaving meta-model for system configuration modeling

Figure 7 shows part of the configuration weaving model for our example, where the method `MapDB.getMap()` acquires a thread upon its invocation and releases it on reply. The model weaves an application layer model containing `MapDB` (Figure 7, left) with a platform layer model containing `SimpleThreadPool` (Figure 7, right) by the *ResourceHandling* class instances (Figure 7, middle): the *ResourceHandling* link *'Map acquire thread'* connects the *start* action of the method `MapDB.getMap()` as a *consumer* to the `SimpleThreadPool.acquire()` method SEFF as the corresponding *resource*; and the *ResourceHandling* link *'Map release thread'* connects the *stop* action of the method `MapDB.getMap()` as a *consumer* to the `SimpleThreadPool.release()` method SEFF as the *resource*.

We aim to keep the layers' models free of inter-layer usage and allow for a flexible alteration of this information through such configuration models. Therefore, a fully automatic generation of weaving models through some kind of dependency analysis is not possible. However, the usage of default configuration



```
*twoLayersAppTPConfig.amw

Application
  Repository _rqBIgFPtEeGyNJbBBZoa4A
    Basic Component MapDB
      Resource Demanding SEFF getMap()
        Start Action start
        Stop Action stop
    Basic Component TripPlanner

Weaving model
  platform:/resource/UsageWMs/model/twoLayersAppTPConfig.amw
    Usage Model TripCoordinatorAndThreadPool
      <<usages>> Resource Handling Map acquire thread
        <<consumer>> Used Model Element start
        <resource> Used Model Element acquire()
      <<usages>> Resource Handling Map release thread
        <<consumer>> Used Model Element stop
        <resource> Used Model Element release()

Platform
  Repository _mr1dkGhBEeG6Rc4V3nNI0g
    Basic Component SimpleThreadPool
      Resource Demanding SEFF acquire()
        Start Action a_start
        Stop Action a_stop
        Acquire Action acquire
      Resource Demanding SEFF release()
      Passive Resource Threads
```

Fig. 7. Part of the configuration model for the trip planning example

settings for specialization scenarios to allow semi-automatic generation could be incorporated.

At this point, the resource consumption scenario is modeled directly in AMW as a set of usages, one for each consumer/resource relation. With increasing numbers of layers and interactions between them, the correlated modeling overhead will need to be addressed. One possible solution, which we will consider in our future work, is the automatic generation of such configuration weaving models from a more compact configuration description, e.g. rules or OCL constraints.

3.2 Modular Transformation Concept

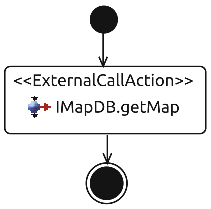
After the system configuration design decision has been defined in a separate weaving model, the entire multi-layer system model is given and can be transformed into a target model of our choice (e.g. CSP). The definition of this transformation depends on the source and target meta-model characteristics, as well as, on the desired properties of the transformation itself. Possibilities vary from a single monolithic transformation to a modular invocation of a set of (re-usable) transformations.

In our case, the proposed transformation approach takes advantage of the compositional layer-based nature of the source models. It allows flexible re-use of transformation results for individual layers, by transforming them into separate models (Figure 1, (1)), and later composing those (Figure 1, (3)) under consideration of the glue model obtained from the configuration model (Figure 1, (2)). Each of these three transformations is briefly described in the rest of this section.

Layer-Wise Model Transformations. As previously mentioned, each model of a specific layer is transformed into the corresponding partial target model. This transformation is goal, as well as, source and target meta-model specific.

In our case, the goal is the analysis of multi-layer systems, and the chosen source and target meta-models are PCM (Section 2.1) and CSP (Section 2.2) respectively. The transformation considers gray box behavior and interaction of components.

Figures 8 and 9 demonstrate the PCM to CSP transformation principle on the `TripPlanner.getTrip()` and `SimpleThreadPool.acquire()` SEFFs of the application and platform layer models of our example. Each of the SEFFs is transformed into a recursive process and its definition is formed from the transformation results for individual action steps.



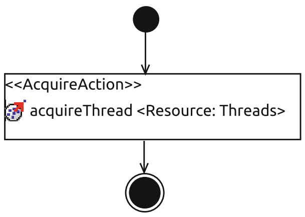
```
GETTRIP =
  invoke_getTrip_start?x → invoke_getTrip_end.x →
  invoke_getMap_start.TripPlannerID →
  reply_getMap_end.TripPlannerID →
  reply_getTrip_start.x → reply_getTrip_end.x →
GETTRIP
```

Fig. 8. Transformation of `TripPlanner.getTrip()` SEFF

Start actions are transformed into process prefixes containing two events representing the begin and end of an operation invocation (e.g. `invoke_getTrip_start?x` and `invoke_getTrip_end.x` for the *start* action of the `TripPlanner.getTrip()` SEFF, Figure 8). The first event accepts an input containing a caller identification and the second passes this parameter value further. This parameter is important for later synchronization with other processes.

Stop actions are transformed into similar prefixes (e.g. `reply_getTrip_start.x`, `reply_getTrip_end.x` for the *stop* action of the `TripPlanner.getTrip()` SEFF, Figure 8), however, the first event is not accepting any new input, but transmitting the parameter obtained by `?x`.

External calls are transformed into process prefixes of two events representing direct invocation and reply of the called operation (e.g. `invoke_getMap_start.TripPlannerID` and `reply_getMap_end.TripPlannerID` for the external call of `MapDB.getMap()` method). In our example the caller is the `TripPlanner` and, therefore, its ID is used as an event parameter value. Transformation details for other elements of PCM like branches, loops, etc., are omitted here, but a general idea can be found in [31].



```
ACQUIRE =
  invoke_acquire_start?x → invoke_acquire_end.x →
  acquireThread_start.SimpleThreadPoolID →
  acquireThread_end.SimpleThreadPoolID →
  reply_acquire_start.x → reply_acquire_end.x →
ACQUIRE
```

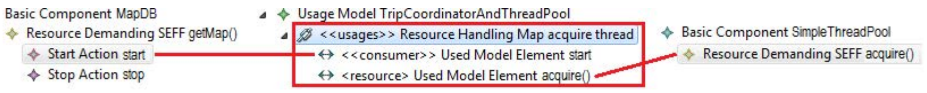
Fig. 9. Transformation of `SimpleThreadPool.acquire()` SEFF

Figure 9 shows the *acquire* action `acquireThread`, which is transformed similar to *start/stop* actions, but the prefix events have a fixed caller ID (e.g.

`SimpleThreadPoolID`). This is due to the fact, that in PCM an *acquire/release* action can only be called by the owner of the resource. The transformation of the *release* action is very similar. The passive resource itself is represented by an instance of the corresponding managing process (see Section 2.2, `TPOOL`) parameterized by the resource capacity.

All described transformations have been implemented using the hybrid model transformation language ATL [20]. They are applied to each layer model independently, and the results are later (re-)used in the composition of the overall target model (see Figure 1 (1, 3)).

Configuration Model Transformation. The weaving model for system configurations described in Section 3.1 contains information required to compose the transformation results of individual system layers into the complete target model. It is used to create a CSP glue model, which defines the composition of all layer processes within the system combined with a set of adapter processes for modeling usage information without altering the initial layer models. In particular, the glue model contains the `SYSTEM` process, which represents the complete system and a set of `ADAPTER` processes introducing interactions between the previously independent *consumers* and *resources* for each *usage*.


Basic Component MapDB
Resource Demanding SEFF getMap()
Start Action start
Stop Action stop
Usage Model TripCoordinatorAndThreadPool
<<usages>> Resource Handling Map acquire thread
<<consumer>> Used Model Element start
<resource> Used Model Element acquire()
Basic Component SimpleThreadPool
Resource Demanding SEFF acquire()

$$\begin{aligned}\texttt{ADAPTER\_MAP\_THREAD} = {} & \texttt{invoke\_getMap\_start?x} \rightarrow \\ & \texttt{invoke\_acquire\_start.MapDBID} \rightarrow \texttt{reply\_acquire\_end.MapDBID} \\ & \rightarrow \texttt{invoke\_getMap\_end.x} \rightarrow \texttt{ADAPTER\_MAP\_THREAD}\end{aligned}$$

Fig. 10. Transformation of *ResourceHandling* usage *'Map acquire thread'* into CSP adapter

Figure 10 shows an adapter process generated for one *ResourceHandling* usage with the `SimpleThreadPool.acquire()` *resource* and `MapDB.getMap()` start action *consumer*. Outer events of the adapter allow synchronization with the *consumer* and inner events with the *resource* operation. This ensures that the consumer action is completed only if the resource operation was successful. Adapters generated for communication usages have similar structure, but assume resource usage in both communication directions.

The `SYSTEM` process combines the layers processes (e.g. `TRIPPLANNER` and `THREADPOOL`) through adapter processes (e.g. `ADAPTER_MAP_THREAD`) depending on the usages defined between the layers:

$$\texttt{SYSTEM} = ((\texttt{TRIPPLANNER} \underset{\texttt{X}}{\parallel} \texttt{ADAPTER\_MAP\_THREAD}) \underset{\texttt{Y}}{\parallel} \texttt{THREADPOOL})$$

The alphabets `X` and `Y` define synchronization between the application layer and adapters, and adapters and the platform layer respectively. Outer adapter events with concrete values are in `X` = {`invoke_getMap_start`.`TripPlannerID`, `invoke_getMap_end`.`TripPlannerID`}, and inner adapter events are in `Y` = {`invoke_acquire_start`.`MapDBID`, `reply_acquire_end`.`MapDBID`}.

The described transformation for the weaving model (Figure 1, (2)) has been also implemented in ATL. The last step is to compose the obtained results into the complete target model.

Model Composition & Re-Use. Target models for each layer and for the configuration model are composed and re-used during the creation of the overall target model (Figure 1, (3)). For this purpose, an additional transformation has been implemented in ATL to merge the models incrementally by combining elements, omitting duplicates (by name) and updating references, if required. Such modularity requires compositional qualities from the target meta-model.

4 Tools and Experiments

This section describes the application of our approach on the running example and the tools we use.

Tools. We modeled all system layers in PCM with the help of the *Palladio-Bench* [4]. For CSP, we created a custom meta-model using the *Eclipse Modeling Framework (EMF)* [1], based on the CSP meta-model provided in [31]. The weaving model is specified within the *AMW (Atlas Model Weaver)* (see Section 3.1). The model transformations are implemented in *ATL* (*Atlas Transformation Language*, [20]).

To analyze the generated CSP models, we use the CSP refinement checker *FDR* [2]. A model-to-text transformation is used to obtain CSP_M code from our CSP model instances, which is used as an input format for FDR. FDR provides deadlock and livelock checks as well as specific refinement checks. Here, we only use the former two.

Experiments and Results. A typical scenario of the influence of platform models on the analysis of a software system model is the consumption of limited resources. Our examples are based on a scenario described in [13]: there, some Web-services execute complex computations from the domain of theoretical chemistry and, although proven deadlock-free, deadlock after the deployment due to the real servers' thread management strategies.

We start with the CSP model, that is generated from the application model consisting of the software components `TripPlanner` and `MapDB` (Section 2.1). As a platform layer model, we use the simple thread pool introduced in the same section, that is capable of managing a limited amount of threads. Additionally, we consider another platform model describing communication between connected software components.

Communication is modeled by an interface `ITransmission`, containing a single method `ITransmission.transmit()`. We model two different components to provide that interface: `LocalCall`, (not shown) contains only a trivial SEFF with only a start and a stop action; and `Rpc` (Figure 11) as a composite component, that models a high-level view of *Remote Procedure Calls* (RPC). As we focus on communication, it contains only two components, `RpcSender` and `RpcReceiver`, that are responsible for the remote method call. Furthermore, we use the RPC model to inject an error: the SEFF for `RpcSender.transmit()` contains an infinite loop as an alternative to the correct external call action to the connected receiving component. This way, a potential livelock is introduced; a livelock occurs when the system does not make any progress, but is not in a deadlock state.

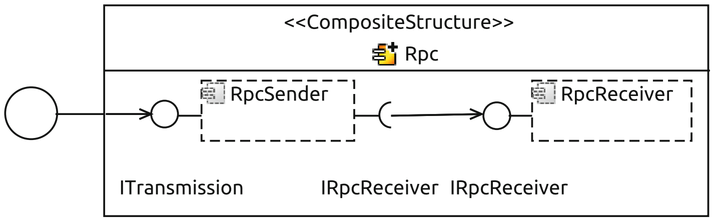


Fig. 11. Composite component modeling an abstract RPC behavior

This scenario provides various possible configurations, that can be checked for deadlocks and livelocks.

First of all, we analyze the stand-alone application layer model. Then, we create a configuration model such that both basic components of the application model require a thread from the thread pool model. We also combine the application model with each of the communication models, the trivial `LocalCall` and the `Rpc` components. Additionally, we use the failure injecting variant of the RPC model as well as the correct one. Finally, we combine application, thread management, and communication models such that the communication model components consume threads too. The thread pool is configured by different number of threads. For our experiments, we currently consider only a single user calling the modeled trip planning system.

We generate CSP models and check them with FDR. The application layer model is both deadlock- and livelock-free. Connected to the thread pool model, it deadlocks depending on the number of threads available: as two components require a thread, providing only one thread leads to a deadlock. The application model connected to the trivial communication model (`LocalCall`) is proved correct, while with the error-prone RPC model, it contains an expected livelock. A corrected RPC model fixes this. The configurations containing the application, the thread pool and a communication model, where the communication components consume threads too, is analyzed with the expected results: a deadlock is detected if the thread pool model does not provide enough threads. Figure 12 summarizes the results.

Of course, on this example the results given by FDR are to be expected. However, more complex case studies like [13] have shown that such an analysis is actually necessary to obtain the correct configuration of the platform layer (i.e., thread pool parameters). Our approach gives us a modular generation of different analysis models: the analysis model for the application layer remains the same, and is glued together with different platform analysis models.

Combination of models	**deadlock?**	**livelock?**
Application	no	no
Application + thread pool (1 thread)	yes	no
Application + thread pool (2 threads)	no	no
Application + local call	no	no
Application + RPC (erroneous)	no	yes
Application + RPC (correct)	no	no
Application + thread pool (3 threads) + RPC (correct)	yes	no
Application + thread pool (4 threads) + RPC (correct)	no	no

Fig. 12. Summary of model configurations and sample properties

5 Related Work

Model transformations play a key role in model-driven development. Their application areas include *model composition* [6,5,23], *abstraction* [27] and *refinement* [21], as well as *model translation* between different languages [12,22,17] (e.g. our PCM to CSP example).

Transformations themselves can be represented as models [9], and exposed to the above mentioned transformation operations. In this case, we get higher-order model transformations (HOTs) [29] operating on transformation models.

Composition of models [6,5,23,19,14,25,21] and model transformations [24,16,32] remains an active research topic aiming to increase flexibility during development and re-use. Model compositions in [6] are realized via direct composition of meta-models. The authors of [5] propose a generic composition approach that allows definition of compositions for specific domains by establishing correspondence relations to the generic part. This way composition operators and strategies can be re-used. A modular model composition approach in [23] uses fragment-based glue models, similar to weaving models, to connect models conforming to one meta-model. The mechanism defining mappings between the glue and the models relies on fixed integration points within models that cannot be flexibly changed. In [19], the authors explore concern-based composition of multi-format models with integration points within these concerns. The approach presented in [10] extends this multi-format idea by employing ontologies to integrate different meta-models based on common concepts, however, it does not address model-level relation of conceptually different, but related entities, that we are focusing on. Mentioned approaches do not promote modular composition of models the way the model weaving technique, that we use, does.

In [14,25,21] model composition approaches are driven by a specific goal. [14] proposes modular composition of analysis models transformed from service compositions with deployment models based on predefined resource consumption logic. The approach does not consider multi-layer systems and does not allow explicit modeling of resource usage, like in our case. [25] also propose modular composition of Petri net models for analysis. Both approaches are similar to our with regard to modularization, but we also allow flexible multi-layer model re-configuration via a non-invasive model weaving.

In [21] the authors propose a model-driven performance prediction approach which, like our method, considers PCM models, but only on the application level. It uses feature model instances, called performance completions, to enrich the application model with platform and infrastructure information. Unlike in our method, these performance completions are not independent re-usable models, that might also consume resources, but instances of a predefined set of features spread over different layers. To analyze performance, the authors first refine the annotated PCM model by means of a transformation specifically generated by a HOT based on selected performance completions. The chosen refinement strategy does not allow transformation modularization that we have.

Composition of model transformations has been studied by [24,16,32]. [24] proposes model transformation composition by chaining single transformations. We employ such a chaining technique to combine individual transformations used in our method. Authors of [16] advocate for the establishment of a model-driven development for model transformations to simplify their composition and re-use. [32] proposes a rule-based transformation composition technique. For our modular method this composition technique is too fine-grained. The approach proposed in [33] also has its focus on the re-use of transformations. However, it is achieved not by a composition, but by a generic to specific transformation mapping. This technique also uses weaving to define correspondences between the generic and specific meta-models. Approaches presented in [16,33] can be potentially integrated in our method for developing model transformations.

Translating model transformations constitute the backbone of MDE, and facilitate tool interoperability [12,22,17], as well as integration of formal analysis [14,7,25] in the development process. Tool interoperability and model integration approaches use techniques like matching transformations and weaving [12], or meta-model bridging defined by dedicated mapping components [22]. [17] provide an alternative to the meta-model weaving technique for model integration based on Triple Graph Grammars (TGGs, [26]). Our method uses techniques similar to some of these approaches, however, we focus on the model (and not meta-model) level integration techniques for multi-layer systems.

The already mentioned approach [14] additionally to model compositions also uses translating transformations. The authors translate service compositions under resource constraints into process calculi for verification. Approaches presented in [7,25] employ translation of component-based software models into various types of Petri nets to facilitate performance prediction. We use translation from PCM to CSP to demonstrate our modular transformation approach by facilitating formal analysis of multi-layer systems. However, we do not limit applicability of our method to this scenario.

The model weaving technique, that we employ, has also been used by other approaches [12,33,30,15] including those previously mentioned [12,33]. Some of them use weaving on meta-model level to define links for all model instances, whereas others, like us, use weaving on the model level in such contexts like web engineering [30] and aspect-oriented modeling (AOM) [15].

6 Conclusion

In this paper, we have proposed a modular transformation approach for multi-layer systems. It promotes the use of weaving models for the system configurations, which are then transformed into the glue for the partial target models obtained for the individual layers. This way we can re-use existing model transformations and even their already existing results. This approach has proven beneficial for an analysis which needs to flexibly (re-)combine different layer models to different extends. We have exemplified our approach on the analysis of component-based software with respect to deadlock/livelock properties.

In the future we, in particular, intend to carry out more case studies to identify an extend of linking facilities required for configuration weaving, and consider more concise ways to specify such configuration models. Furthermore, we aim to extend the application of our modular framework to other target meta-models, i.e., languages for performance and reliability analysis.

References

1. Eclipse Modeling Framework (EMF), http://www.eclipse.org/modeling/emf
2. Failures-Divergence Refinement (FDR2), http://www.fsel.com
3. National Institute of Standards and Technology (NIST), http://www.nist.gov/itl/cloud/
4. Palladio-Bench, http://www.palladio-simulator.org
5. Anwar, A., Ebersold, S., Nassar, M., Coulette, B., Kriouile, A.: Towards a generic approach for model composition. In: The Third International Conference on Software Engineering Advances, ICSEA 2008, pp. 83–90 (October 2008)
6. Balasubramanian, K., Schmidt, D.C., Molnar, Z., Ledeczi, A.: Component-based system integration via (meta)model composition. In: 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, ECBS 2007, pp. 93–102 (March 2007)
7. Becker, S., Koziolek, H., Reussner, R.: The Palladio component model for model-driven performance prediction. Journal of Systems and Software 82, 3–22 (2009)

8. Besova, G., Wehrheim, H., Wagner, A.: Reputation-based reliability prediction of service compositions. Electr. Notes Theor. Comput. Sci. 279(2), 3–16 (2011)
9. Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A.: Model Transformations? Transformation Models! In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 440–453. Springer, Heidelberg (2006)
10. Bräuer, M., Lochmann, H.: Towards semantic integration of multiple domain-specific languages using ontological foundations. In: Proceedings of the 4th International Workshop on (Software) Language Engineering, ATEM 2007 (2007)
11. Didonet del Fabro, M.: Metadata management using model weaving and model transformation. Ph.D. thesis, Universite de Nantes (September 2007)
12. Del Fabro, M.D., Valduriez, P.: Towards the efficient development of model transformations using model weaving and matching transformations. Software and System Modeling 8(3), 305–324 (2009)
13. Foster, H., Emmerich, W., Kramer, J., Magee, J., Rosenblum, D.S., Uchitel, S.: Model checking service compositions under resource constraints. In: ESEC/SIGSOFT FSE, pp. 225–234 (2007)
14. Foster, H., Uchitel, S., Magee, J., Kramer, J.: An integrated workbench for model-based engineering of service compositions. IEEE T. Services Computing 3(2), 131–144 (2010)
15. Groher, I., Völter, M.: Aspect-oriented model-driven software product line engineering. T. Aspect-Oriented Software Development VI 6, 111–152 (2009)
16. Guerra, E., de Lara, J., Kolovos, D.S., Paige, R.F., dos Santos, O.M.: *trans*ML: A Family of Languages to Model Model Transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 106–120. Springer, Heidelberg (2010)
17. Hermann, F., Ehrig, H., Orejas, F., Czarnecki, K., Diskin, Z., Xiong, Y.: Correctness of Model Synchronization Based on Triple Graph Grammars. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 668–682. Springer, Heidelberg (2011)
18. Hoare, C.A.R.: Communicating sequential processes. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1985)
19. Johannes, J., Aßmann, U.: Concern-Based (de)composition of Model-Driven Software Development Processes. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part II. LNCS, vol. 6395, pp. 47–62. Springer, Heidelberg (2010)
20. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: a model transformation tool. Science of Computer Programming 72(1-2), 31–39 (2008)
21. Kapova, L., Reussner, R.: Application of Advanced Model-Driven Techniques in Performance Engineering. In: Aldini, A., Bernardo, M., Bononi, L., Cortellessa, V. (eds.) EPEW 2010. LNCS, vol. 6342, pp. 17–36. Springer, Heidelberg (2010)
22. Kappel, G., Wimmer, M., Retschitzegger, W., Schwinger, W.: Leveraging Model-Based Tool Integration by Conceptual Modeling Techniques. In: Kaschek, R., Delcambre, L. (eds.) The Evolution of Conceptual Modeling. LNCS, vol. 6520, pp. 254–284. Springer, Heidelberg (2011)
23. Kelsen, P., Ma, Q.: A Modular Model Composition Technique. In: Rosenblum, D.S., Taentzer, G. (eds.) FASE 2010. LNCS, vol. 6013, pp. 173–187. Springer, Heidelberg (2010)
24. Oldevik, J.: Transformation Composition Modelling Framework. In: Kutvonen, L., Alonistioti, N. (eds.) DAIS 2005. LNCS, vol. 3543, pp. 1135–1136. Springer, Heidelberg (2005)

25. Salmi, N., Ioualalen, M.: Towards Efficient Component Performance Analysis in Component Based Architectures. In: Biffl, S., Winkler, D., Bergsmann, J., Aalst, W., Mylopoulos, J., Rosemann, M., Shaw, M.J., Szyperski, C. (eds.) Software Quality. Process Automation in Software Development. LNBIP, vol. 94, pp. 121–142. Springer, Heidelberg (2012)
26. Schrr, A., Klar, F.: 15 Years of Triple Graph Grammars. In: Ehrig, H., Heckel, R., Rozenberg, G., Taentzer, G. (eds.) ICGT 2008. LNCS, vol. 5214, pp. 411–425. Springer, Heidelberg (2008)
27. Sen, S., Moha, N., Baudry, B., Jézéquel, J.-M.: Meta-model Pruning. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 32–46. Springer, Heidelberg (2009)
28. Stahl, T., Völter, M., Bettin, J., von Stockfleth, B.: Model-driven software development: technology, engineering, management. Wiley Software Patterns Series. John Wiley, Chichester (2006)
29. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the Use of Higher-Order Model Transformations. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 18–33. Springer, Heidelberg (2009)
30. Vara, J.M., de Castro, V., Del Fabro, M.D., Marcos, E.: Using weaving models to automate model-driven web engineering proposals. IJCAT 39(4), 245–252 (2010)
31. Varró, D., Asztalos, M., Bisztray, D., Boronat, A., Dang, D.-H., Geiß, R., Greenyer, J., Van Gorp, P., Kniemeyer, O., Narayanan, A., Rencis, E., Weinell, E.: Transformation of UML Models to CSP: A Case Study for Graph Transformation Tools. In: Schürr, A., Nagl, M., Zündorf, A. (eds.) AGTIVE 2007. LNCS, vol. 5088, pp. 540–565. Springer, Heidelberg (2008)
32. Wagelaar, D.: Composition Techniques for Rule-Based Model Transformation Languages. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 152–167. Springer, Heidelberg (2008)
33. Wimmer, M., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W., Sánchez Cuadrado, J., Guerra, E., De Lara, J.: Reusing model transformations across heterogeneous metamodels. In: Proceedings of the 5th International Workshop on Multi-Paradigm Modeling @ MoDELS (2011), http://avalon.aut.bme.hu/mpm11/papers/mpm11_submission_3.pdf

Research-Based Innovation: A Tale of Three Projects in Model-Driven Engineering

Lionel Briand[1], Davide Falessi[2,3], Shiva Nejati[1], Mehrdad Sabetzadeh[1], and Tao Yue[2]

[1] SnT Centre, University of Luxembourg, Luxembourg
[2] Certus Software V&V Centre, Simula Research Laboratory, Norway
[3] University of Rome (Tor Vergata), Italy
{lionel.briand,shiva.nejati,mehrdad.sabetzadeh}@uni.lu,
{falessi,tao}@simula.no

Abstract. In recent years, we have been exploring ways to foster a closer collaboration between software engineering research and industry both to align our research with practical needs, and to increase awareness about the importance of research for innovation. This paper outlines our experience with three research projects conducted in collaboration with the industry. We examine the way we collaborated with our industry partners and describe the decisions that contributed to the effectiveness of the collaborations. We report on the lessons learned from our experience and illustrate the lessons using examples from the three projects. The lessons focus on the applications of Model-Driven Engineering (MDE), as all the three projects we draw on here were MDE projects. Our goal from structuring and sharing our experience is to contribute to a better understanding of how researchers and practitioners can collaborate more effectively and to gain more value from their collaborations.

1 Introduction

Research and innovation go hand in hand in all engineering disciplines and software engineering is no exception to this rule. Unless engineering research and innovation are done in tandem and synergistically, both will suffer: research may be poorly aligned with the "pain points" of the industry and will consequently have limited impact; and innovation will be hampered if the industry is deprived of an inflow of creative ideas and solutions stemming from research.

Motivated by the above, we have been seeking in the past few years ways to collaborate more closely with industry, both to ensure better alignment between our research and the current industrial needs, and further, to demonstrate to our industry partners the role of software engineering research in boosting innovation. This is what we refer to as *research-based innovation.*

This paper discusses our experience with three projects that have reached maturity, selected from a larger set of ongoing projects that we are currently conducting in collaboration with the industry. We reflect on the way we managed our interactions with our industry partners in these projects, our observations, and the decisions that we believe contributed to the collaborations being more effective. We discuss a number of lessons learned that emerged from our collective experience and illustrate these lessons with concrete examples.

All three projects use Model-Driven Engineering (MDE) technologies [1]. This adds yet another dimension of complexity: Despite the increasing momentum of MDE, conducting MDE research in an industrial context remains hard, mainly due to the difficulty of securing adequate buy-in from the partner companies. The lessons learned we discuss in the paper not only cover the researchers' role in research-based MDE projects but also the expectations from the industry in such projects, including upfront investment in learning and tailoring of MDE solutions and the existence of champions for the devised solutions.

Our focus on MDE makes our work a useful complement to recent initiatives by other researchers, most notably by Mohagheghi and Dehlen [2] and Hutchinson et al. [3,4], to investigate the success and failure factors for MDE in industrial settings and the perceptions of practitioners about MDE. Our work, however, differs from these initiatives in two ways: First, our goal is to provide insights about how to engage industry in *MDE research*, rather than applying MDE per se. The second difference is the source of information on which we draw our conclusions. Whereas Hutchinson et al. use surveys and interviews, and Mohagheghi and Dehlen use a literature review as their primary means for data collection, we rely on the experience gained through direct engagement with industry in research and development activities.

The rest of the paper is structured as follows: Section 2 introduces the overarching project (ModelME!) within which the three (sub)projects that we focus on in this paper were conducted. The section continues with an overview of each of the three projects. Section 3 describes the way we organized our industry collaborations. Section 4 discusses and illustrates the lessons learned from the three projects, organized according to the steps in our collaboration model (Section 3). Section 5 summarizes the paper and highlights important observations.

2 Context: The ModelME! Project

The projects discussed in this paper are part of a larger project, called ModelME! (*Model*-Driven Software Engineering for the *M*aritime and *E*nergy Sectors, `http://modelme.simula.no`). Broadly, the objective of ModelME! is to improve software engineering best practices for software-intensive systems in the Maritime and Energy (M&E) sectors. In this section, we briefly describe three (sub)projects within ModelME! that have reached maturity and have been validated in realistic settings. These projects are the source for the lessons learned discussed in Section 4.

2.1 Traceability and Slicing (TS)

This project concerns the problem of requirements to design traceability and slicing of design models to improve design inspections during software safety certification. The industrial partner in the project was a large supplier of programmable marine electronics. During our preliminary discussions, the engineers in the partner company noted some issues related to the preparation and assessment of software safety certification documents. To better understand the current software safety certification practices, we attended a number of certification meetings between the company's engineers and a certification body. Our observation of the meetings suggested that the majority of the issues identified during design inspections in the certification process arise due to poor structuring of the specifications and missing traceability, in particular, between safety requirements and software design.

Following our observations, we set our research goals to be: providing an information model to characterize the traceability links required in design safety inspections, a model-based methodology for establishing such traceability links, and a mechanism for extracting minimized and relevant slices of the design for a given safety requirement. Grounding our work on the Systems Modeling Language (SysML) [5], we have developed a tool-supported framework for design safety inspections in the context of safety certification [6,7], applied the framework to a number of selected software modules from our industry partner, and created guidelines tailored to the partner company for using our framework [8].

Our partner has now started using our guidelines in the design of its modules. The models built in our case study are planned to be used in the upcoming round of safety certification at the company. The project has thus far engaged three full-time researchers for six months and one full-time engineer for two months.

2.2 Configuration and Derivation of Subsea Control Systems (CDSCS)

Our second project has been carried out in collaboration with another large systems supplier in M&E, particularly known for their Subsea Control Systems (SCSs). The embedded software in SCSs has very large configuration spaces, including configuration for hardware architectures, for data communication protocols, and for the individual physical devices.

In our initial investigation, we observed that the hardware and software configuration processes at our partner company were isolated from one another, resulting in many configuration mismatches and errors that were often detected late and only after the integration of software and hardware. We therefore set the primary goal of this project to be: providing support for configuration and derivation of the software components in SCSs such that the complex dependencies between hardware and software are captured and preserved. To achieve this goal, we have developed a model-based approach for configuring the software embedded in SCSs. We use built-in UML features for modeling the architecture of SCSs and the architectural dependencies between the software and hardware elements. Our approach can capture complex software-hardware dependencies. The approach automates some of the configuration decisions and interactively guides users to make the remaining configuration decisions [9,10].

We have evaluated our approach on a case study from the partner company [10,11]. Our experience shows that our approach successfully enforces consistency of configurations, can automatically infer up to 50% of the configuration decisions, and reduces the complexity of making configuration decisions. We are now working on integrating our approach and tool into the current configuration process at the partner company. This project has thus far engaged two full-time researchers for one year, and one full-time engineer for three months.

2.3 Technology Qualification (TQ)

The third project concerns the assessment of new technologies. New technologies usually include novel aspects that are not addressed by existing standards and cannot be certified in the sense that more conventional safety-critical systems are. To demonstrate the safety and reliability of new technologies, these technologies are often subject to a specific kind of assessment, which in many industries is known as Technology Qualification (TQ). The TQ project was conducted in collaboration with DNV (Det Norske Veritas), which engages in various qualification projects, with a focus on M&E, particularly offshore platforms and subsea systems.

As with the other two projects discussed earlier, our first step was developing a better understanding of the needs and priorities of our industry partner. The following observations were made about the current TQ practice based on meetings and interviews with domain experts: (1) There is not adequate traceability between the safety and reliability goals of a new technology, the identified risks, and the evidence collected to show that the technology is fit for purpose; (2) The process taken to elicit expert judgment is not always explicit, with a potential negative impact on the transparency of the qualification process; (3) Time and budget overruns can occur if the evidence collection effort is not focused on building and improving the right evidence information.

In response to the above, we have developed a model-based approach for probabilistic assessment of new technologies [12,13], which combines goal models [14], expert probability elicitation [15], and Monte Carlo simulation [16]. To facilitate the adoption of the approach, we have developed a prototype but highly usable tool to support it. We have further created a handbook for internal use by DNV providing practical guidelines on how to use the research results and the tool. Our solution has so far been applied in two industrial case studies both concerning off-shore technologies. The first case study is described in [12].

The approach is currently being validated and refined internally at DNV in other projects. An annex based on the results of our work is being considered for DNV's technology qualification recommended practice [17,18]. The TQ project has thus far engaged two full-time researchers for a year, two master students for six months, and five DNV staff members for approximately four months.

3 A Collaboration Model for Research-Based Innovation

To collaborate more efficiently with our partners, and further to ensure that we record the insights gained from the collaborations for future projects, we apply a unified collaboration model across the projects in our group. One of the first tasks we perform when engaging with a new industry partner is to discuss this model. We have found this to be useful in two ways, first to put in place a progression path for the project, and second, to clarify what we need from the industry partner at each stage, and what outcomes they can expect from the collaboration as the project progresses.

Our collaboration model in Figure 1, builds on and refines the collaboration model proposed by Gorschek et al. [19]. Below, we first describe our collaboration model and then discuss how it modifies Gorschek et al's. In Section 4, we will present the lessons learned according to the steps of our collaboration model.

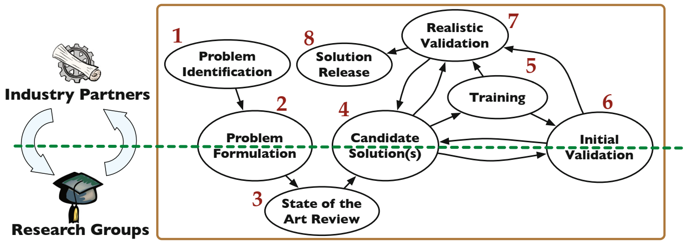


Fig. 1. Adaptation of Gorschek et al's model [19] for research-based innovation

The first step, *Problem Identification*, aims to identify and obtain an initial understanding of the problem that the partner company wishes to address. This is often done through meetings and organizing workshops where the industry experts at the partner company give presentations about their perceived challenges. In the second step, *Problem Formulation*, the identified problem is formulated in a more precise manner, and the context factors and working assumptions are clearly specified. If the problem is large and multi-faceted, it may further need to be decomposed into sub-problems that can be prioritized and tackled independently.

In the third step, *State-of-the-Art Review*, a critical review of the research literature and existing commercial or open source technologies takes place in order to identify to what extent the goals are already addressed and what are the open issues to deal with through research. In other words, the research will benefit from both the current practice and the published literature. In the fourth step, *Candidate Solutions*, one or more potential solutions are devised. These solution(s) will be iteratively refined based on the subsequent evaluation steps (steps 6 and 7).

The fifth step, *Training*, is an incremental step. In the early stages, training focuses on building up the background necessary for the practitioners to form an (early) judgment about the feasibility of the solutions. Particularly, early training should cover the modeling constructs that the solutions expect as input. While not a definitive feasibility assessment, this allows practitioners to determine if the constructs are "natural" and "easy to build" in realistic settings given the resources they have available. In later stages and as the solutions mature, training shifts towards practical guidelines and detailed methodological steps for applying the solutions. In addition, training is a nice way to discuss with the industry partners the value of MDE in general.

In the sixth step, *Initial Validation*, we conduct a preliminary evaluation of the solutions, either in an artificial or an industry setting. In an industrial setting, we use a mix of seminars, hands-on workshops, and surveys for initial validation. If validation in an artificial setting is possible, we may use controlled studies, e.g., controlled experiments.

If the results of initial validation are promising, we move up the evaluation ladder to *Realistic Validation*. In this step, we run case studies in industrial settings, starting with pilot studies first and then spreading to wider use. The details of the proposed solutions will be refined, in particular by providing practical guidelines, and tool support will be developed. During the pilot studies, only a small group of stakeholders will be engaged, and experience and viewpoints on practices and tools will be collected through interviews and questionnaires. A typical result from pilot studies is that the practices are better streamlined to reduce the overhead associated with learning and using these practices. Subsequently, the streamlined practices and tools are rolled out to a wider group, data collection is performed on these projects to further assess and refine, on a wider scale, the proposed technologies.

We note that our collaboration model allows one to bypass Initial Validation and move directly to Realistic Validation. This flexibility is desirable for two reasons. First, it allows for more agility in the execution of a research-based project if there are constraints on timelines and/or available resources. Second, such flexibility is required when a solution cannot be meaningfully evaluated outside a realistic setting (e.g., when expert judgment is required).

In the eighth (and final) step, *Solution Release*, the refined tools and training material are released to the industry partner for broader application according to the exploitation plans of the industry partner. Here, the research team plays a primarily supportive role, e.g., through consultancy and maintenance services. While the Solution Release step deserves careful consideration in research-based projects, the three projects upon which we draw in this paper have only been recently released, thus offering limited insight about the longer-term interactions between research and industry in this step. A longer-term investigation is required to provide a more conclusive picture of this step.

As we stated earlier, our collaboration model is an adaptation of the collaboration model proposed by Gorschek et al. Our model offers two refinements over Gorschek et al's:

(1) We propose an explicit step for training, which starts long before the validation steps. A major obstacle in conducting industrial MDE research is the perception that one has to learn languages like UML in their entirety, before being able to benefit from MDE. This perception often leads to the conclusion that the learning curve associated with MDE is too steep. In reality, practitioners have to learn only what they really need from a language like UML, which typically constitutes a small fragment of the language. This fragment is determined by the modeling methodology, which specifies what part of the notation is used for a given objective and how the variation points in the semantics are

specialized to address the objectives. A key goal of training should then be to minimize the learning curve by narrowing the training material to the language fragment that the practitioners actually need.

(2) We distinguish only between two validation steps, initial and realistic; whereas Gorschek et al. distinguish three: validation in academia, static validation (early stage industrial validation) and dynamic validation (late stage industrial validation). Dynamic validation corresponds to realistic validation in our model; but our model combines validation in academia and static validation into one, namely initial validation. This is because we find the distinction between validation in academia and static validation blurry. A pure validation in academia requires good benchmarks which are in general scarce for MDE. The chances of finding suitable benchmarks decreases even further noting that industry-driven research problems are defined in light of the context factors and working assumptions of the industry partner. As a result, even when benchmarks are used, they need to be tweaked and aligned with the partners' needs first. Subsequently, all validation activities at all steps are informed by real-world considerations. Further, as we said earlier, we allow for bypassing initial validation when the solution inherently requires a realistic setting.

Figure 2 shows the distribution of project effort over the steps of our collaboration model for the three projects that we described in Section 2. As noted earlier, the projects are too young to provide detailed insights about the Solution Release step in our collaboration model. Therefore, we do not consider Solution Release in the project effort distribution. We further note that the percentages in Figure 2 are estimations. Keeping track of real effort was not possible due to the lack of control on how much work the industry partners did related to the collaboration outside the meetings and workshops we had with them. As can be seen from the figure, the relative effort spent on Steps 1, 3 and 4 are comparable across projects; whereas, there are discrepancies across projects between the relative effort spent on Steps 2, 5, 6 and 7.

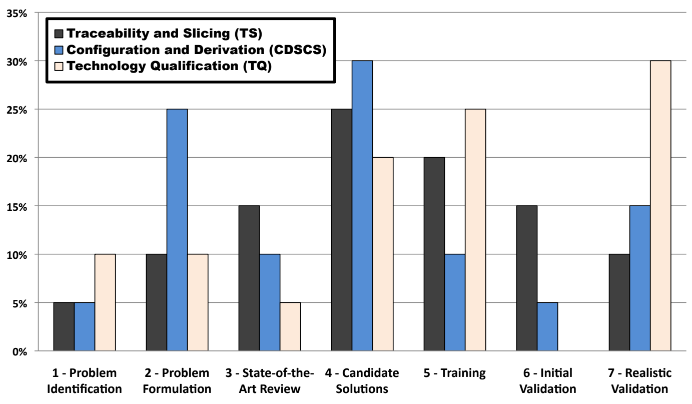


Fig. 2. Approximate effort distribution for the three projects. Horizontal axis represents the steps of the collaboration model; vertical axis represents the percentage of overall project effort in a given step.

Step 2 in CDSCS used more of the project effort compared to TS and TQ, because in this project, multiple concrete software products had to be examined in order to develop a complete picture about the types of the configurable parameters and the architectural dependencies. The training step in CDSCS instead took comparatively less effort than in the other two projects. This was mainly due to the larger participation of the CDSCS partner in formulating the problem and defining a solution, and thus receiving more exposure to modeling concepts before detailed training. As for Initial Validation, the TS project required a larger percentage of effort, because the project involved a benchmark case study and was validated in an artificial setting first. Finally, the TQ project used a larger percentage of the project effort over Realistic Validation than the other two projects. This was because none of the aspects of the solution developed for TQ could be validated outside a realistic setting. Therefore, the evaluation for TQ was focused entirely on Realistic Validation. Note that this also had implications on the level of training required in the TQ project. Since there was no initial validation, the experts involved in the TQ project had less exposure to the solution ahead of realistic validation than the experts in the TS and CDSCS projects. This gap in solution familiarity had to be compensated for via training.

Our experience from these projects indicates that an important success factor in industry research is having a precise and concrete problem formulation. That is, a reasonable amount of effort should be spent during the initial steps of a project to adequately specify the problem and capture its context factors and working assumptions. Investing effort early on for problem formulation not only improves the credibility and validity of the final solution, but could also reduce the training effort (as was the case in CDSCS).

4 Lessons Learned

In this section, we describe the lessons learned from the three MDE projects outlined in Section 2. Several of these lessons are general and not limited to MDE per se. However, since they came out of MDE projects, their usefulness and whether they convey the right priorities in other types of research projects has to be further investigated.

4.1 Problem Identification

LL1. The Stated Problem Is Often Only a Manifestation of One or More Fundamental Problems. This lesson underlines the importance of observational studies in early stages of a project. In the TS project (Section 2.1), the problem initially stated by the engineers was to extend and refine their Failure-Mode and Effect Analysis (FMEA) techniques [20]. After attending software certification meetings with third-party certifiers, we observed that the majority of the issues that the certifiers raised were due to the lack of traceability from safety requirements to design.

In the CDSCS project (Section 2.2), initial discussions suggested that the most pressing issues concerned the integration of third-party components. The data collected from the systematic domain analysis that followed these initial discussions provided us with more insights and showed that a major fraction of the integration problems stem from improper configuration of software components.

In the TQ project (Section 2.3), the issue initially raised was the need to increase the transparency and cost-effectiveness of the technology qualification process. This high-level objective was refined through interviews, observation of technology qualification activities, and reviewing of some past technology qualification projects. The refined research needs that came out from analyzing the high-level objective were discussed in Section 2.3.

4.2 Problem Formulation

LL2. Build a Domain Model as early as Possible. A domain model is a useful tool to structure the detailed discussions about a problem with an industry partner, to uncover the tacit knowledge of experts about their domain, and to avoid ambiguity and misunderstandings over terminology. Industry partners often see immediate value in domain modeling: with relatively small effort, they get a reusable "mind map" of concepts they have to deal with on a regular basis. Domain modeling is highly interactive and uses intuitive notations like UML class diagrams (or SysML block diagrams if SysML is used). Both factors contribute to leaving a good "first impression" about MDE.

In the TS project, domain modeling was performed using SysML and spanned two days, involving approximately 10 person-hours of effort. The resulting SysML (context) diagram was used during problem formulation for determining the modules that were subject to safety certification, and subsequently, for identifying the links that these modules had to other modules in the system.

In the CDSCS project, a domain model using UML was constructed during problem formulation for one family of the company's subsea systems. The domain model took about 60-80 hours to create and served as a basis for identifying and classifying the concepts relevant to configuration of subsea systems.

In the TQ project, given that we were not concerned with the development of any particular system, and were focused on assessment, we did not initially see the value of building a domain model. Instead of building a domain model, we developed a glossary to distinguish assessment concepts such as goal, requirement, evidence, safety margin, etc. As the project progressed, it was realized that a glossary alone, while useful, was not sufficient, because the relationships between abstract and concrete terms, and the associations between different concepts could not be easily specified. For example, we needed to distinguish various types of safety evidence, e.g., testing results, analytical models, historical data. This can be much better done using a model than only a glossary. In the light of this observation, we are now extending our current assessment methodology [12] to accommodate an explicit step for domain modeling.

4.3 State of the Art Review

LL3. Carefully Consider the Context Factors and Working Assumptions. Context factors and working assumptions constrain what can be a feasible solution to a problem and are crucial in determining whether an existing solution can be adopted from the literature or a new solution needs to be developed.

The contextual factors in the TS project were: (1) The system requirements were expressed as natural language statements and there was little flexibility in using more rigorous requirements specification approaches (e.g., formal methods); (2) There was a technical constraint that a standardized notation (e.g., SysML) should be used for the design to minimize ambiguity and communication overhead when the licensing or regulatory body is performing safety certification; and (3) The goal pursued from traceability was very specific, namely compliance to the IEC61508 standard. We did not find any existing solutions in the literature that satisfied this particular combination.

Likewise, in the CDSCS project, the survey that we conducted of the existing variability modeling languages did not identify any specific solution that could be directly used for configuring architectural variabilities in SCSs and could further capture the hardware-software dependencies in such systems.

In the TQ project, our literature review identified good solutions for the sub-problems of the original problem. The main challenge was integrating the solution components into a complete and seamless solution. This was complicated by the fact that the solution components were multidisciplinary, and combining them required background from different fields.

In all three projects, the industry partners expected *end-to-end* solutions that were not only technically sound but could also satisfy their criteria about cost, training, integration with existing process, and organization culture.

4.4 Candidate Solutions

LL4. If Practitioners Cannot Conveniently Provide the Input Required by a Solution, The Solution Is Unlikely to Be Adopted. MDE solutions are often accused of requiring input models that are too complex for the engineers to build, or that are based on information which cannot be realistically obtained at an organization. A major consideration in defining an MDE solution is the simplicity and naturality of the models that it requires as input and making sure the input language is a suitable match for the expertise, processes, and the culture at the partner company (also see *LL3*).

LL5. Rely as much as Possible on Standardized Modeling Languages. Reliance on standardized modeling languages increases buy-in from the industry because it largely avoids “lock in”. Before a company invests into MDE technologies, they need to ensure that the technologies are going to be supported for a long time. Proprietary modeling languages are usually considered risky, because there is uncertainty as to how long they may be supported. Using standards is further advantageous for tool building, because solutions can be built on top of the existing commercial or open-source environments.

The TS project was based on SysML due to its rising popularity in systems engineering. We used only a subset of SysML that was essential for capturing the design of the IO modules at the partner company. We held four modeling workshops with the lead engineer of these modules to ensure that our requirements for the input models are reasonable. Thanks to the existing modeling platforms for SysML, we could develop a tool for our solution, as a plugin for the Enterprise Architect modeling environment (http://www.sparxsystems.com/products/ea/), with relatively little effort.

In the CDSCS project, we base our work on UML and its extension for Modeling and Analysis of Real-Time and Embedded systems (MARTE) [21], primarily because this combination can seamlessly capture software and hardware concepts. Our methodology was designed after conducting interviews with the engineers at the partner company and eliciting detailed information about their systems to ensure that the methodology would match their needs. The methodology has been implemented on top of the Rational Software Architect modeling environment (http://www.ibm.com/developerworks/rational/products/rsa/).

In the TQ project, the main decision about the input language concerned the specification and decomposition of safety and reliability goals. Before adopting goal modeling as the basis for our work, we made sure that key goal modeling concepts such as "goals" and "obstacles" were natural for the experts. Among the existing goal modeling languages, we chose KAOS (Knowledge Acquisition in Automated Specification) [14] for two main reasons: (1) the existence of an extended set of modeling guidelines in a textbook [14] which could be used for training; and (2) amenability of KAOS to quantitative assessment. This made KAOS a nice fit for the existing technology qualification practice which is based mainly on probabilistic assessment. To provide a usable tool, we implemented the KAOS notation as a UML profile for the Enterprise Architect environment, rather than developing a tool from scratch.

4.5 Training

LL6. Do Training Only Incrementally and Based on Needs. Training has to be incremental and tailored to the needs of the industry partner. On the one hand, engineers have little slack time and cannot be expected to attend extensive training. On the other hand, the engineers will not be able to apply the proposed solutions unless they have received adequate training.

LL7. For Training, Use Examples from the Domain Being Studied, Not Examples from Textbooks or Other Domains. No matter how complete and concrete the examples used for training are, if the examples are not related to the industry partner's domain, they will seldom be convincing enough. An example in an industry training course is not merely to convey an idea but also a critical means to demonstrate that the idea applies to the domain of interest. Naturally, examples drawn from a particular domain are also easier to remember and relate to for experts in that domain. Using such examples also increases the overall effectiveness of training by creating a greater incentive for engineers to actively participate in the training sessions.

In the TS project, training was integrated with the modeling review sessions that we conducted with the engineers. The goal of these sessions was: First, to validate and refine our design models for the IO modules, and second, to help engineers modify and build these models for other similar modules. During these sessions, the engineers were asked to comment on the models and to change them for other IO modules. At the end of these sessions, we provided the engineers with a technical report that included step-by-step guidelines for creating design diagrams and traceability links for their IO modules.

In the CDSCS project, we held a number of modeling tutorials focused specifically on the UML diagrams that were essential for understanding our methodology. In the tutorials, we used illustrative examples from the models that we had built for a family of subsea control systems at the company. The tutorials were interactive and the attendees were provided with booklets containing modeling guidelines and examples. At the end of the tutorials, they were asked to perform a number of modeling exercises.

In the TQ project, training was performed in a number of modeling workshops. We gave an introduction to KAOS based on its reference book [14]. For exemplification, we developed examples from real technology qualification projects. Each workshop included a hands-on training session where we interactively built and refined goal models with the experts. We observed that the experts became increasingly self-sufficient in goal modeling over time.

4.6 Initial Validation

LL8. Validation in an Artificial Setting May Be of Limited Value or Not Possible. Without good benchmarks, validation in an artificial setting may have limited value, because there may be too wide a gap in terms of assumptions and level of complexity between an artificial and a real case study. For some problems, an artificial setting may not even be possible, e.g., when expert judgment is involved.

LL9. Take Particular Note of Scalability Considerations During Initial Validation. Unfortunately, scalability often comes at the cost of lowering precision, which in turn reduces the conclusiveness of the analyses performed. During initial validation, it is important to discuss with the industry partner how a proposed solution will "degrade" in the face of reduced precision in the input models. A solution is less likely to be adopted if the degradation is too fast or is just binary (i.e., there is a precision point above which analysis is fully conclusive and below which analysis is fully inconclusive).

In the TS project, our initial validation involved applying our solution to the Production Cell System (PCS) [22] – a well-known benchmark for reactive systems, which has been previously used to evaluate the capabilities of various specification methods for safety analysis [22]. We constructed a complete set of

SysML models and traceability links for PCS. Throughout our benchmark study, we also interacted with our industry partner to learn about the design of their systems. This interaction influenced the way the design models in the benchmark were built. A high priority for the partner was to keep the design effort low, while satisfying all the criteria for compliance with the relevant safety standards. The partner was flexible to accept a reasonable increase in the amount of modeling effort if this meant the resulting models would be reusable for other modules, or exploitable for purposes other than certification, e.g., for staff training.

In the CDSCS project, no artificial cases studies were performed because we were unable to find representative benchmarks. In this project, we were given access to a real system by the industry partner at the beginning of the project. Initial validation was done through seminars in which we presented to the industry partner our solution at different stages of progress and obtained feedback. Scalability was a key requirement for the solution and had a direct influence on the level of abstraction of the architectural models that we built, and on the design of our configuration tool [9].

In the TQ project, no initial validation was performed. As stated earlier, our solution in TQ involves expert probability elicitation and the only plausible way to evaluate the solution was to apply it to a real case.

4.7 Realistic Validation

LL10. Choose Your Pilot Studies Wisely! New solutions are rarely put into use immediately and are almost always tried on pilot projects first. To increase the chances of a solution getting adopted, one must take the following factors into account when selecting pilot studies:

- Pilot studies should be representative in the eyes of the industry partners, so that they can believe the results. In other words, the pilot studies should adequately reflect the characteristics of the broader range of systems that the solutions are targeted at.
- Pilot studies should be complex enough for validation and yet commensurate with the available resources. Results from simplistic pilot studies may be unconvincing or even misleading. On the flip side, if the pilot studies require resources beyond what the research team can secure from the industry partner, the studies will not succeed.
- When feasible, pilot studies should be defined over *ongoing* activities at the partner company, as opposed to over past activities. From a practical standpoint, basing a pilot study on ongoing work is beneficial in two ways: First, the effort for the pilot study will be usually overlapping with the work that the staff at the partner company have to do anyways to deliver their products and services. Due to this overlap, the pilot study will no longer be viewed as a side activity and the staff will be more willing to spend resources on the pilot study. Second, if the pilot study contributes to improving the current activities, the solution will have an immediate and tangible impact on the company, thus increasing buy-in.

If a reenactment of past activities is chosen as the basis for a pilot study, the research team has to ensure that the company has a horizon for using the results of the pilot in the future; otherwise, it may become difficult to stimulate enough interest from the partner company to actively participate in the pilot.

LL11. Be Ready to Provide Substantial Help in Model Construction During Realistic Validation. Mentorship is a critical element for success in research-based innovation, particularly in the context of MDE. Specially, if the industry collaborators do not have a long history of using MDE, the research team has to set a good example during early case studies, which the collaborators can learn from, refer to, and reuse in the future. We believe that mentorship is best done through the deep involvement of the research team in the construction of the case study models. While time-consuming, "getting one's hands dirty" with model construction is also an excellent way for the researchers to understand the modeling needs, and further to show to the partner company that the researchers are genuinely interested and committed to addressing the partner company's problems, in turn helping with building trust (see *LL12*). Once the mentorship process is complete, researchers' assistance in model construction should be phased out to avoid the validity threats one may face in our empirical studies due to actively helping in creating the models.

In the TS project, the IO modules under study had a complex multithreading structure. We were told early on that, unless the models built to specify the multithreaded behavior of the modules were representative of all the modules, the development team would be unable to apply our solution, because the multithreading models were too expensive to build for the modules individually. The module selected for our case study was deemed as having all the multithreading features that the broader set of modules use. However, to get a grip on the complexity of the case study, we had to compromise on the communication protocols that the module could work with: we only considered the simplest communication protocols in our case study. The researchers led the effort on model construction for the study. The work was aligned with the current needs of our partner as the models were being prepared for the next round of certification.

In the CDSCS project, representativeness referred to covering as many variability types in the product family as possible. To achieve this, we chose three different products that were deemed by the industry partner as collectively covering the majority of the variability types in their systems. These products were planned to be used as a basis for their future product development. To manage the complexity of realistic validation, instead of building a product-line architecture model that exhaustively captures all the commonalities and variabilities in the products, we created a model that contains instances of all variability types. The modeling effort was led by the researchers with participation from both management and development staff at the partner company.

The TQ project differed from the TS and CDSCS projects in that in was done in collaboration with an assessment body rather than a system supplier. The body qualifies a diverse set of technologies ranging from purely mechanical equipment to software-controlled systems. Due to this diversity, it was infeasible to define representativeness in an actionable manner. The selection process for our pilot studies was opportunistic with the following constraints: (1) the pilot studies must not be too time-consuming for the experts and should preferably be on current/recent qualification projects, (2) experts in the domain areas of the pilot studies have to be available throughout the studies for goal model construction and expert elicitation. The modeling effort in our first pilot study was led by the researchers. In the second pilot study, the models were constructed by the experts with some help from the researchers.

4.8 Solution Release

LL12. Find Internal Champions for the Solution. For most industry research problems, the researchers get to collaborate with the technical staff at the partner companies, but the final decision about whether to adopt a proposed solution is made by the management team who may not have been involved in the collaboration. To be able to carry a new solution through to broad use, the technical staff at the partner company must champion the solution. In other words, they have to make a convincing case to the management about the solution's benefits. For this purpose, the technical staff often have to provide compelling business cases where the solution leads to cost savings or quality improvements, and further to propose a strategy for integrating the solution into the current development workflows at the company. This level of commitment does not materialize unless the industry collaborators develop a strong sense of trust in the researchers and the research being conducted. Building such trust takes *years* and requires the researchers to develop a deep appreciation of the business culture at the partner company [23].

The TS project is championed by the lead engineer of the IO modules, who has also developed and presented an exploitation plan to the management after the industrial case study was concluded. The CDSCS project is championed by the quality assurance team, who are currently investigating how the developed solution can be integrated into the existing tool chain at the company. For the TQ project, management was involved in the technical work of the project, giving us the opportunity to continuously synchronize our solution and tool support with the required business cases at the partner company.

5 Conclusion

This paper reports on experiences we have had with three industry partners in performing what we call research-based innovation. The fundamental position of this research paradigm is that, in software engineering as in other engineering disciplines, research and industrial innovation can be beneficially intertwined in order to ensure that the problems addressed by researchers are well-defined and relevant. We must also strive to account for all important context factors in devising solutions to software engineering problems and this requires close interactions with industry partners.

This research paradigm is also an effective way to transfer novel technologies to practitioners as they are involved early on in the development of the solutions, thus creating many opportunities for mentoring and a sense of ownership. Another way to describe this research paradigm is to highlight its *inductive* nature, that is the fact that we work from specific observations in concrete settings but attempt to build general solutions with clear working assumptions.

The lessons learned we report in this paper are focused on applications of Model-Driven Engineering. They are structured according to the various phases of our research-based innovation model (Figure 1), starting with Problem Identification and ending with Solution Release. The ultimate goal of structuring and sharing these experiences is to help future researchers and practitioners better cooperate and ensure the success of their collaboration endeavors. Some of these lessons learned focus on how to thoroughly understand the problem and context before working on a solution. In software engineering, characteristics of the system, organization, and human factors can have a strong influence on whether a solution is applicable and scalable.

We also discuss other factors that influence the development of a solution such as the need to account for modeling standards and assessing the feasibility of integrating the solution within the existing process. How to perform training is also addressed as being a key component of the paradigm. Industry practitioners usually have little time to devote to professional education and this is usually a significant obstacle to change. Last, we addressed the validation of the proposed solutions, both at an early stage and then later on in realistic project settings. A two-stage validation, though not always possible, is a way to alleviate the risks associated with novel solutions.

From a more general standpoint, this paper discusses ways to bridge the existing gap between software engineering research and practice, an issue we believe to be of crucial importance for the future of our profession.

Acknowledgments. We thank Vic Basili, Richard Torkar and Jose Luis de la Vara for their useful comments on an earlier draft of this paper. L. Briand was supported by a FNR PEARL grant.

References

1. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: FOSE, pp. 37–54 (2007)
2. Mohagheghi, P., Dehlen, V.: Where Is the Proof? - A Review of Experiences from Applying MDE in Industry. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 432–443. Springer, Heidelberg (2008)
3. Hutchinson, J., Rouncefield, M., Whittle, J.: Model-driven engineering practices in industry. In: ICSE, pp. 633–642 (2011)
4. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of mde in industry. In: ICSE, pp. 471–480 (2011)
5. Object Management Group (OMG): Systems Modeling Language (SysML), version 1.1. (2008), (http://www.omg.org/docs/formal/08-11-02.pdf

6. Nejati, S., Sabetzadeh, M., Falessi, D., Briand, L., Coq, T.: A sysml-based approach to traceability management and design slicing in support of safety certification: Framework, tool support, and case studies. Technical Report 2011-01, SRL (2011)
7. Falessi, D., Nejati, S., Sabetzadeh, M., Briand, L., Messina, A.: Safeslice: a model slicing and design safety inspection tool for sysml. In: SIGSOFT FSE, pp. 460–463 (2011)
8. Sabetzadeh, M., Nejati, S., Briand, L., Evensen Mills, A.: Using SysML for modeling of safety-critical software-hardware interfaces: Guidelines and industry experience. In: HASE (2011) (to appear)
9. Behjati, R., Nejati, S., Yue, T., Gotlieb, A., Briand, L.: Model-based automated and guided configuration of embedded software systems (2012) (in submission)
10. Behjati, R., Yue, T., Briand, L., Selic, B.: Simpl: A product-line modeling methodology for families of integrated control systems. Technical Report 2011-01, SRL (2011)
11. Behjati, R., Yue, T., Nejati, S., Briand, L., Selic, B.: Extending SysML with AADL Concepts for Comprehensive System Architecture Modeling. In: France, R.B., Kuester, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 236–252. Springer, Heidelberg (2011)
12. Sabetzadeh, M., Falessi, D., Briand, L., Alesio, S.D., McGeorge, D., Åhjem, V., Borg, J.: Combining goal models, expert elicitation, and probabilistic simulation for qualification of new technology. In: HASE (2011) (to appear)
13. Falessi, D., Sabetzadeh, M., Alesio, S.D., Briand, L.: Modus: A tool for goal-based safety and reliability assessessment of new technologies (2011) (submitted)
14. van Lamsweerde, A.: Requirements Engineering - From System Goals to UML Models to Software Specifications. Wiley (2009)
15. O'Hagan, A., Buck, C., Daneshkhah, A., Eiser, J., Garthwaite, P., Jenkinson, D., Oakley, J., Rakow, T.: Uncertain Judgements: Eliciting Experts' Probabilities. Wiley (2006)
16. Robert, C., Casella, G.: Monte Carlo Statistical Methods. Springer (2005)
17. Det Norske Veritas: Qualification procedures for new technology DNV-RP-A203, DNV (2001)
18. Det Norske Veritas: Technology qualification management DNV-OSS-401, DNV (2010)
19. Gorschek, T., Garre, P., Larsson, S., Wohlin, C.: A model for technology transfer in practice. IEEE Software 23(6), 88–95 (2006)
20. Ericson, C.: Hazard Analysis Techniques for System Safety. John Wiley & Sons (2005)
21. Object Management Group (OMG): A UML profile for MARTE: Modeling and analysis of real-time embedded systems (May 2009)
22. Lewerentz, C., Lindner, T. (eds.): Formal Development of Reactive Systems - Case Study Production Cell, Formal Development of Reactive Systems. LNCS, vol. 891. Springer (1995)
23. Tarr, P.: So you want to marry an industrial. Presentation (2011)

An Industrial System Engineering Process Integrating Model Driven Architecture and Model Based Design

Andrea Sindico[1], Marco Di Natale[2], and Alberto Sangiovanni-Vincentelli[3]

[1] Elettronica SpA, Roma, Italy
andrea.sindico@elt.it
[2] TECIP Institute, Scuola Superiore S. Anna, Pisa, Italy
marco.dinatale@sssup.it
[3] EECS Dept, University of California at Berkeley
alberto@eecs.berkeley.edu

Abstract. We present an industrial model-driven engineering process for the design and development of complex distributed embedded systems. We outline the main steps in the process and the evaluation of its use in the context of a radar application. We show the methods and tools that have been developed to allow interoperability among requirements management, SysML modeling and MBD simulation and code generation.

Keywords: System Engineering, Model-Driven Architecture, Model-Based Design, Platform-Based Design.

1 Introduction

The complexity of modern cyber-physical systems is rapidly growing. Advanced system engineering methodologies are required to integrate all the competencies and specialty groups required for the realization of a system using a structured development process from concept to production to operation. The motivations and objectives of the industrial design process we present in this paper are:

- Improve the quality of the requirements moving towards their definition in a formal language and the need for tracking requirements into design artifacts and hardware or software implementations;
- Improve the quality of the functional solutions by early verification and validation on models using simulation, model checking or other forms of automated verification;
- Produce reusable and documented components at all levels in the design flow;
- Automatically derive implementations from models that are provably correct. Also, automatically generate documentations and possibly test cases.

Model-driven approaches such as domain-specific modeling languages, Model-Driven Architecture (MDA) and Model-Based Development (MBD) are possible choices to form the backbone of the design flow. Albeit MDA and MBD share the same principle (models as primary artifacts driving the design and development process), they differ substantially in their details. Each on his own is incapable of addressing all the challenges of modern system design. However, their strengths and weaknesses are complementary: MBD languages enable the realization of mathematical specifications that can be exploited for system simulation, testing and behavioral code or firmware generation, but they lack expressive power to represent complex system architectural aspects and execution platforms. Moreover, their extension mechanisms are quite limited in scope. On the other hand, MDA languages are very good at representing architectural aspects and are designed for being easily extended and provide mechanisms to transform models expressed in a language into another. However it is still hard to exploit these kind of models for model execution and simulation.

Starting from requirement capture, our approach follows the tenets of Platform-Based Design (PBD)[2], in which a functional model of the system is paired to a model of the execution platform. In particular, we present in this paper an integration of MBD and MBA to realize a comprehensive system engineering process based on the INCOSE framework [23]at Elettronica S.p.A (ELT)[1], one of the European leaders in the production of Electronic Defence equipment (EW). We describe tools and model integration techniques, automatic code generation using both MDA and MBD tools and the development of domain specific metamodels and profiles, extending the OMG MARTE standard.

2 The Design Process

2.1 Our Objectives and Motivations to Change

As any industry the main reasons why we have decided to evolve and update our design and development process are: to increase quality; to increase productivity; to reduce costs. In order to achieve these objectives we have identified some aspects of our design and development process that could be positively affected, for what concerns quality, productivity and costs, by a model driven approach. The first aspect is: documentation. As a company operating in the defense industry we have to provide a lot of documentation when designing and developing a product. According to the MIL-STD-498 standard [24], to which ELT is conform, more than 20 different documents have to be provided just for the design and development of the software architectgure of a system. For each document a review process has to be done in order to ensure its contents are correct, enough descriptive and coherent. This is a very expensive activity that has to be repeated every time we modify a project. Thanks to the model driven workflow here described we have managed to automatically obtain almost all the documentation from the models. The second aspect we wanted to optimize is the interaction among engineers with different specialties (i.e. SW, HW, FW) which work together but use different modeling and development tools. Often errors, due to misunderstanding, crept in and an extra effort was frequently needed in order to integrate their design activities. The process we present in this paper is instead a unified framework enabling SW,FW and HW engineers to share models and reduce integration effort. The third aspect is the reuse of components which is now optimized by the fact that the exploitation of modeling languages enable the definition of a functional architecture which is independent of a specific actual execution platform. Only when it has to be deployed onto a specific platform the functional model is related to a platform model. Such mapping eventually enables the automatic generation of the SW and FW artifacts which realize the functional model for that platform.

2.2 The Overall Structure

The model driven design and development process we have defined (shown in Figure 1) starts with the *System Requirements Analysis and Definition* stage that establishes functional and performance requirements. The output of this phase is a *System Subsystem Specification* (SSS) [24] document. Requirements are expressed, documented, managed and linked to test descriptions (for verification) using the **DOORS** tool by IBM. In order to provide the required formalization to requirements, DOORS textual descriptions are supplemented by **SysML** state and interaction diagrams, block diagrams, (interface) type definitions and OCL constraints. For the creation and management of the SysML models, the **Topcased** open source tool, based on the Eclipse Modeling Framework (**EMF**) is used. The need to preserve the consistency of the requirements and of the links tracking the requirements to architecture-level design decisions and (refined) subsystem requirements led to the development of automatic transformations between the DOORS and SysML tools and the information managed by them, implemented in a custom Eclipse plugin (shown as ① in the figure). The plugin leverages the support offered by EMF for the OMG languages for the development of metamodels (ECORE) to transform the DOORS requirements modules in an Ecore model, and model-to-model **QVT** transformations to keep the DOORS and SysML models synchronized.

The following stage of architecture-level *Solution Definition* translates these requirements into a system architecture design and the corresponding document called *System Subsystem Design Description* (SSDD) [24] that encompasses the functional and execution platform architecture, and addresses the functional requirements defined in the SSS. In the ELT process, DOORS is the master tool for storing the textual requirements associated with the SSS and SSDD. System models of the functional architecture and the execution platform are defined in Topcased, and the plugin connecting Topcased and Eclipse with DOORS synchronizes the architecture-level requirements in a similar way to what is done with the system-level specifications. In addition, the plugin detects refinement chains in the SysML model and automatically imports them as DOORS links (step ② in the figure).

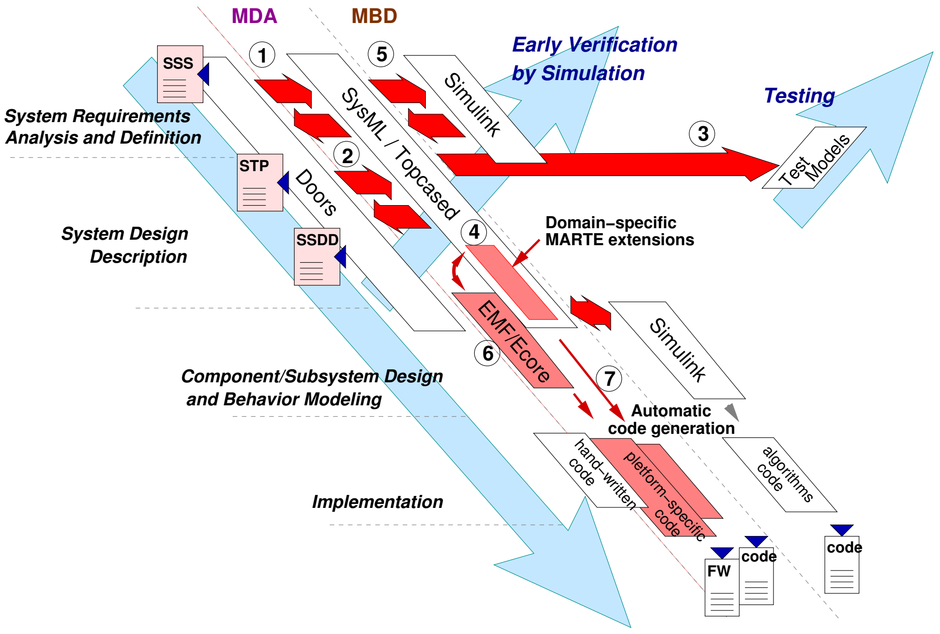

MDA
MBD
SSS
1
5
Early Verification by Simulation
Testing
System Requirements Analysis and Definition
SysML / Topcased
Simulink
2
3
STP
Doors
Test Models
System Design Description
SSDD
4
Domain-specific MARTE extensions
EMF/Ecore
Simulink
6
7
Component/Subsystem Design and Behavior Modeling
Automatic code generation
hand-written code
platform-specific code
algorithms code
Implementation
FW
code
code

Fig. 1. Early V&V, model transformations and automatic generation of implementations in the ELT process

System interfaces are described in additional documents: the *Interface Requirements Specification* (IRS)[24], and the *Interface Design Description* (IDD). The ELT process realizes a platform-based design approach [2] rendered here for the sake of simplicity as an early V&V model on top of a (conventional) V process. Automation is provided for the system-level testing stage, where the SysML interaction diagrams defined for the system and subsystem specifications are automatically processed to generate the models of the system- and subsystem-level tests that verify them (③ in the figure). In architecture design, the SysML models of the system and the subsystems are defined according to the Platform-based Design paradigm, separating the functional model from the model of the execution platform, including the physical architecture. A third model represents the deployment of the functional subsystems onto the computation and communication infrastructure and the HW devices. To define the execution platform and the mapping relationships between the functions and the platform (which defines the model of the software tasks and the network messages, among others) domain-specific SysML extensions are required. We found that the standard MARTE profile [10] is not completely adequate for our needs. We therefore defined our own domain-specific stereotypes as extensions to MARTE (step ④ in the process). Synchronous reactive behavioral models of algorithmic components are developed in **MATLAB/Simulink**: a Mathworks toolset comprising a graphical modeling language (Simulink); a scripting language (MATLAB [18]) and a set of simulation, analysis and optimization and synthesis tools. These models provide an early validation of the system functionality by simulation and are used to automatically derive a software or firmware implementation. The Simulink models can refine functional subsystems of the SysML architecture (in a top-down development) or provide building blocks for the definition of the system (in a bottom-up flow). To support both paths, we developed scripts and code generation templates that can be used to transform a Simulink subsystem (hierarchy) into a (set of) SysML block(s) and viceversa, preserving the flow specifications at the interface ports (step ⑤). The process uses code generation techniques or automatic generation of firmware implementations starting from the Simulink models for the behavioral part, and from the SysML architecture description for the implementation of the communication and synchronization functions and for the code framework of the software tasks ⑦. The synthesis of the communication functions over shared memory and serial links uses a definition of the message model based on an Ecore metamodel that has been defined *ad hoc* ⑥.

2.3 Structure of a Project

Figure 2 depicts the reference SysML project structure used to organize and relate the model elements used in the system design process. The project consists of six packages:

- a *SystemRequirements* package containing a SysML model of the system requirements imported from a DOORS SSS module by means of a RIF export;
- an IntefaceDataTypes package containing a SysML model defining the Data Types and Interfaces provided and required by the system and its parts;
- a *SystemFunctionalArchitecture* package containing a SysML model describing the functional architecture of the system, as a network of subsystems exchanging data signals;
- an *ExecutionPlatform* package containing a SysML model describing the execution platform in terms of the HW and basic software components, including boards, memories, processing units (cores), network connections, but also device drivers, operating system(s) and middleware. For this purpose, we extended the standard MARTE profile [10] for real-time and embedded systems, providing baseline concepts for representing HW/SW systems;
- a *Mapping* package containing a SysML model using an extension of the MARTE Mapping profile to specify how functional components and behaviors are mapped onto an execution platform, generating the software architecture of tasks and messages. To guarantee independence and reusability as well as visibility of the design entities involved in the mapping, the mapping model imports both the functional and platform models;
- a *Test* package containing a SysML model defining all the tests by means of which the system requirements shall be verified.

Separated from the project models we maintain a domain model which is shared among the different projects and contains domain specific meta-entities describing the information managed by the system. This organization enables the reuse

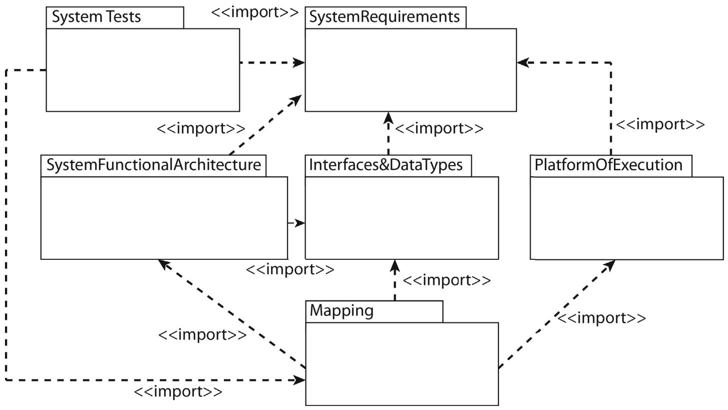


Fig. 2. The structure of SysML projects in ELT

of Interfaces and Data Types and according to the PBD paradigm, allows deploying (by mapping) the same functional model into different platforms of execution.

2.4 System Requirement Modeling and Management

Our process starts with the definition of the system requirements in DOORS. Requirements are expressed in natural language, which can be easily understood by customers and other stake-holders, but is subject to inconsistencies, omissions and duplication of information. To partially obviate to these problems, the SSS requirements are paired with a SysML (semi)formal description. Next, when the architecture models are defined, each model artefact must be traced to the requirements that originated it and also to the (subsystem-level) requirements that it defines.

To this end, we realized an Eclipse plugin for automating the exchange of information and the synchronization of requirements models and diagrams between DOORS and Topcased. The plugin exploits a standard XML format for requirements interchange called RIF (*Requirement Interchange Format* [25]). A metamodel for the RIF format is available in Ecore and used in our approach to automatically generate an Ecore model by importing the RIF XML exported from DOORS. As with any other Ecore model, the Eclipse modeling framework automatically generates an editor that can be used to modify the imported data.

In addition, we built a synchronization engine, based on correspondence rules between the DOORS RIF objects and their attributes (in a given module) and corresponding SysML elements and attributes in Topcased. The synchronization module is based on rules written as QVT Model to Model transformations [17] to synchronize the content of corresponding elements or automatically generate them when requested. On top of the synchronization component, a wizard allows the user to create correspondences between sets of requirements and sets of SysML elements. In this way, parts of a source DOORS module (i.e. paragraphs, interface requirements, functional requirements, parametric requirements) can be imported, or updated, into a new or already existing target SysML model. Among the predefined rulesets provided by the wizard, the user can take a DOORS module containing SSS requirements and automatically import it into a (newly generated) SysML project as a set of SysML requirements. These SysML requirements are used in the subsequent phase to define *Satisfy* relationships with the SysML model entities of the architecture design (SSDD).

2.5 Computational Independent Models

A *Computation Independent model* or CIM (also called domain model) is a conceptual model of the domain of interest (or problem domain) which describes the various entities, their attributes, roles and relationships, plus the constraints that govern the integrity of the model. Using Ecore, ELT defined an electronic warfare meta-model (details can be found in [20]) as a formal and structured representation of the electronic warfare concepts, ranging from the concept of *Platform* (i.e. aircraft, ships, tanks, etc.), to *Sensors* (i.e. Radars, ESMs, etc.), electromagnetic *Waveforms* (i.e. Radiofrequency, Pulse Repetition Interval, etc.) and *Countermeasures* (i.e. Jammers, Chaff, Flare, etc.). This Ecore model represents our Computational Independent Model and we want to keep it unique for all the system we design. Our first aim in designing a new system is thus verifying whether our domain model is expressive enough to cope with the system requirements. If the domain model does not contain concepts or entities' characteristics referenced in the system's requirements we extend it or adapt the requirements (in collaboration with the customer).

2.6 Defining the (Platform Independent) Functional Architecture

Once the system requirements are defined and the ontology of the domain entities is available in the CIM, we start the actual system design by defining the functional architecture of the system. To this end, we use SysML with the MARTE (Modeling and Analysis of Real Time Embedded Systems) profile [10]. The SysML functional model of the system is a network of subsystems. Each SysML *Block* represents a (sub)system functionality defined independently of the eventual implementation technology (i.e. Software, Firmware, etc.) or the HW upon which it will be executed (i.e. CPU, GPU, FPGA, etc.) according to the PBD paradigm. The *System Functional Architecture* package contains the SysML design of the architecture, consisting of Blocks, representing functional subsystems, ports (interaction points) and connectors among ports. As in any SysML model, standard (synchronous) invocation of services is modeled through UML *StandardPorts* each typed with a UML *Interface* providing *Operations*, each of which represents a behavior of the related component invoked synchronously (in a blocking fashion) with respect to the caller. Non-blocking communication may occur according to a discrete-event model (through signal events) or according to a stream of data values produced periodically accoding to a discrete-time base. In the case of (asynchronous) signal events, we use

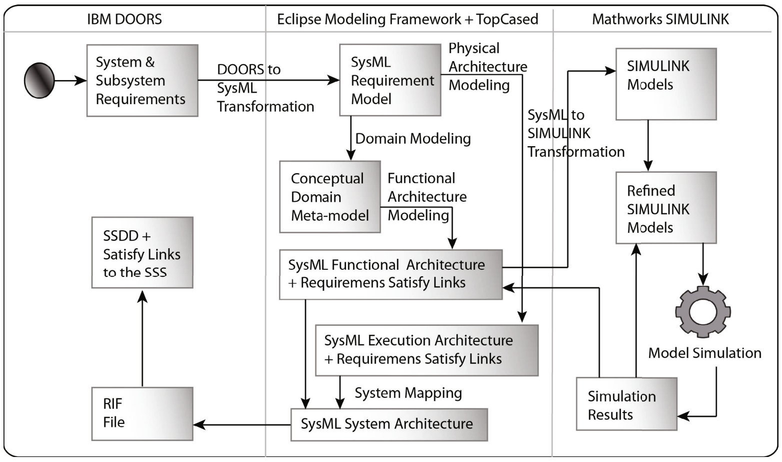

IBM DOORS
Eclipse Modeling Framework + TopCased
Mathworks SIMULINK
System & Subsystem Requirements
DOORS to SysML Transformation
SysML Requirement Model
Physical Architecture Modeling
SIMULINK Models
SysML to SIMULINK Transformation
Domain Modeling
Conceptual Domain Meta-model
Functional Architecture Modeling
Refined SIMULINK Models
SSDD + Satisfy Links to the SSS
SysML Functional Architecture + Requiremens Satisfy Links
SysML Execution Architecture + Requiremens Satisfy Links
Model Simulation
System Mapping
RIF File
SysML System Architecture
Simulation Results

Fig. 3. The system-level design flow

ports stereotyped in MARTE as *ClientServerPort* that allow transmission or reception of UML *Signals*. The corresponding port interfaces are stereotyped as *ClientServerSpecification*. A UML *Reception*, stereotyped as *ClientServerFeature* may be associated to each signal received by the port, specifying the behavior response to it. The case of communication through periodic data streams is of special interest, because it mirrors the communication semantics that is used in the Synchronous Reactive (SR) models produced by the Simulink tool. In this case, the behavior of a SR functional subsystem needs to be further restricted. Communication ports will be SysML flow ports stereotyped as *SRFlowPort*. Also, an SR subsystem is stereotyped as *SRSubsystem* and characterized by the realization of a standard runtime interface consisting of a single *Step* method. The *SRSubsystem* stereotype is associated with an execution period attribute. Interfaces and client server specifications are contained in the *Interface and Data Types* package so that they can used multiple times in different contexts. Data types are generated from the previously described domain model. This guarantees that the system relies on well structured and homogeneous data descriptions, each constrained within the value range prescribed by the SSS. The functional model of the system is part of the SSDD description and must be linked to the SSS requirements from which it originated. After being imported from DOORS, the SSS requirements are mirrored in the SysML tool as a set of SysML requirements. Each component, port, interface and signal of the SysML functional model is connected to the requirement it satisfies by means of the SysML *Satisfy* relationship. OCL scripts verify that each functional component satisfies at least one requirement. Each subsystem will define a set of derived requirements on its structure and behavior. Those requirements are linked with a *Trace* relatonship to the functional subsystem (or element, like a port or even a connection) that originates them. By following the chain SSS Requirement $\rightarrow$ *Satisfy* $\rightarrow$ SysML element $\rightarrow$ *Trace* $\rightarrow$ SSDD Requirement, the tool is capable of inferring a *Derive*

Relationship between the SSS and the SSDD requirements. The synchronization plugin with DOORS allows to define other associations (QVT transformations) for the automatic generation of the objects in a DOORS SSDD module starting from a set of SysML architecture design descriptions and requirements. Further, the tool generates a DOORS refinement link for each *Derive* relationships between SSS and SSDD requirements in Topcased. Figure 4 depicts a portion of a real functional architecture model extracted from an actual ELT system. The figure illustrates the kind of complexity in terms of interactions and interfaces that is typical of ELT models.

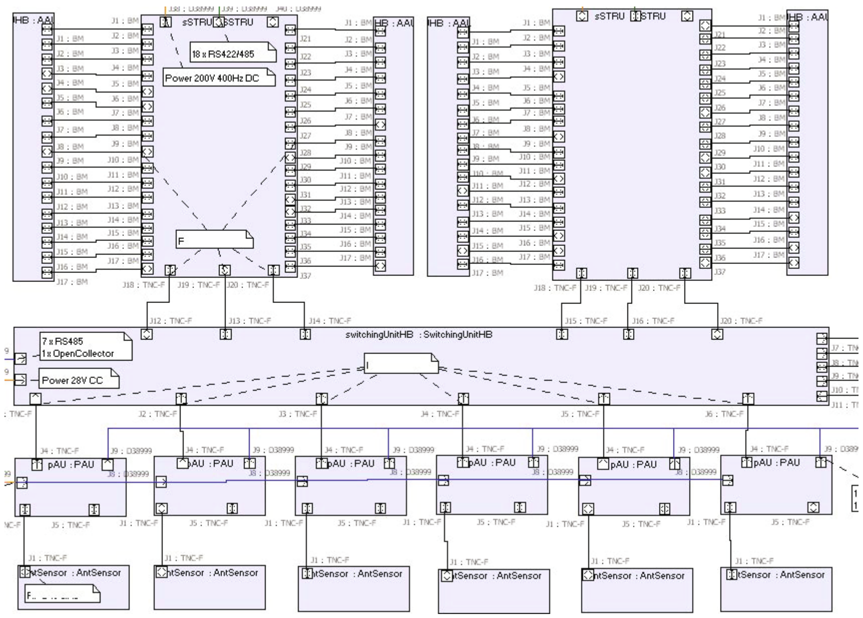


Fig. 4. A portion of the Internal Block Diagram of the EW Integrated System

Once the functional architecture is defined, a state diagram must be associated to every subsystem in the functional architecture model. For each state transition we define both triggers (e.g., the reception of a signal on a port) and guard conditions by means of OCL constraints. After, the behaviors associated with each of the subsystem states are defined.

The UML Superstructure [7] defines two kind of behaviors: *executing* and *emergent*. Interaction diagrams (i.e. sequence diagrams) are used to model emergent behaviors. Communication among active objects occurs through UML *Signals*. Active objects hide their *Operations*, which are invoked only upon the reception of a signal. This approach improves the separation of concerns. While modeling emergent behaviors we add time constraints for each request/response exchange or operation call by means of UML *DurationConstraints*.

Some executing behaviors are defined by activity diagrams and then implemented manually in C++. Other actions define behaviors that are imported from a Simulink model or for which an MBD development flow is going to be used. These are stereotyped as *Analytical* and must refer to a Step function of an *SRSubsystem* block. In case of a top-down process, the development flow makes use of transformations from the interface view of the SysML *SRSubsystem* block into the specification of a Simulink Subsystem, complete with its ports and datatype specification as Bus Objects. The Simulink subsystem is then further developed in the Mathworks environment. The transformation currently in use is an Acceleo script (described in [21]) that transforms the SysML block into a set of MATLAB scripts. More often, however, the functionality to be developed has already been prototyped in Simulink and a reverse transformation generates a SysML block. In this case, a MATLAB script generates an XML file compliant with an Ecore metamodel developed *ad hoc* for the representation of Simulink subsystems in EMF. A QVT model-to-model transformation then generates the SysML block from the Ecore model. Later, at code generation time, special care must be taken when generating the data implementations of the port interfaces and the calls to the Step operations of the *SRSubsystem* blocks.

2.7 Execution Platform Modeling and Mapping

The execution platform and the mapping models define the structure of the HW and SW architecture that supports the execution of the functional model. For both models we leveraged the standard definitions of the MARTE profile. However, it was apparent that MARTE is extensive and general and yet still lacks several features of interest.

The execution platform is defined in a package called *PlatformModels* with the same principles of hierarchical decomposition used in the functional model. Here, blocks represent hardware components at different levels of granularity, but also classes of basic software, including device drivers, middleware classes and operating system modules. The MARTE profile provides several concepts for the basic software classes, but is unfortunately not adequate for the definition of hardware components. For example, not a single stereotype is dedicated to the representation of physical network links (of any type), connectors or cables. Also, concepts like message frames (for Ethernet, Controller Area Network or other standards) and the placement of data signals onto frames are missing. For this reason, we had to define our own taxonomy of stereotyped definitions for most hardware components. Most of them were quite straightforward, others revealed minor complexities or subtleties for usability, such as for the definition of broadcast buses, which are derived by extension from the connector and block metaclasses to ensure the possibility of representing one-to-many connections (impossible with SysML connectors), but at the same time, allowing a more natural modeling of physical links with connectors whenever possible. The definition of stereotypes for other physical elements proved to be much more difficult, as is the case of connectors with multiple pins, a subset of which realizes a communication bus (such as, for example, an Ethernet link). In this case, the connector

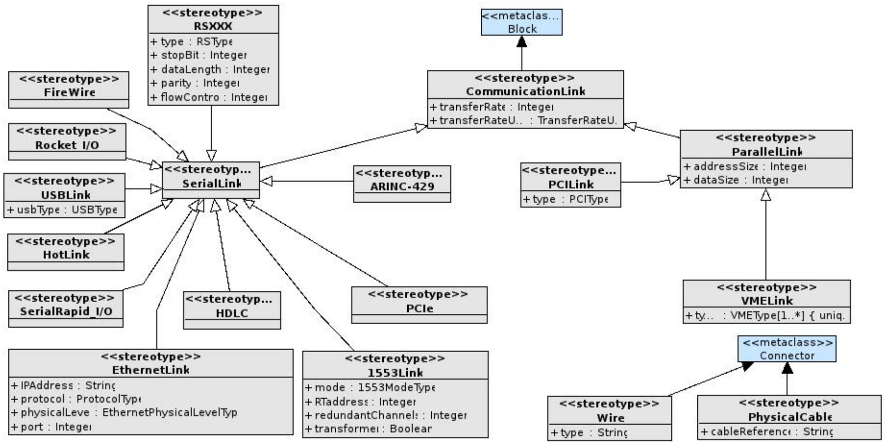


Fig. 5. A UML profile involved in physical communications

stereotype cannot be simply obtained by extending the port concept, because ports cannot contain other ports representing pins. Figure 5 shows an example of UML profile we have developed in order to model serial(e.g., RS232, RS422, ARINC-429, and USB) and parallel (e.g., PCI, and VME) busses and cables. The Mapping definitions are contained in the *MappingModels* package, in which the mapping of the functional subsystems onto the execution platform generates the SW architecture of tasks, messages and logical resources. For the definition of concurrent software we use the *SWConcurrency* package which defines the stereotype *SwConcurrentResource* to represent entities competing for computing resources and executing sequential instruction segments. The elements of this package provide an execution context (e.g., stack, interrupts enable/disable and registers) for an execution flow (sequence of actions).

2.8 Automatic Generation of Documentation, System and Unit Tests and Software/Firmware Implementations

A significant improvement in the productivity and quality of the process is obtained using the tools for the automatic generation of models, documents and implementations (code and firmware). ***Documentation*** Once the elements of the Functional Architecture and the Execution Platform models have been put in a mapping relationship, we can apply a transformation workflow aimed at automatically generating an SSDD document describing the system architecture. Most of the SSDD, SRS and SDD [24] contents are, in fact, already present in the defined functional and platform models and can be imagined as different views of the same models. The QVT transformations of the SysML-to-DOORS plugin generate a RIF file that is imported, generating DOORS modules for the SSDD. IRS and IDD specifications and the corresponding documents, each linked to the related SSS requirements, according to what specified in the SysML model with the *Derive* links.

Test Cases
QVT transformations have been defined to process the interaction diagrams describing the system-level behaviors and generate the sequence diagrams of the test suites that verify out of range and in range invocations for each constrained operation call. Additional sequence diagrams are generated for all the duration constraints aimed at testing out of time conditions. The models representing system and unit tests are generated from the constraints on the entities of the functional architecture and can be an input to further transformations aimed at the automatic generation of the related *System Test Plan* and *System Test Description* documents.

Implementation Generation
The automatic generation of the implementation of functionality is performed using Mathworks tools (Simulink Coder) to derive the FPGA (if firmware) or C (C++) code implementation (if software) of the behavior of some functional subsystems. The generated FPGA implementation communicates with the other subsystems using a set of registers and shared memory locations. The C or C++ generated code follows the conventions of the code generator (two functions for the subsystem initialization and termination and a function with a conventional name for the runtime evaluation of the block outputs given the inputs and the state). The Simulink Coder conventions also define the names of the variables implementing the interface ports. Other functional subsystems are developed manually by hand-written code or purposely designed HW or firmware.

For the infrastructure that provides communication and synchronization among blocks, we exploit the possibility of generating code using Acceleo transformation rules (this work has not been completed as yet). For the communications among subsystems implemented by hand-written C++ code, there are two options. In the case of local communication (detected from the mapping information of the SysML model), we defined a set of transformations generating boost [27] active objects, signals and state machines according to what specified in the platform independent architecture models. In this case, data structures are also automatically generated and OCL constraints turned into run-time checks on the specified value boundaries. When communication is remote (that is, when the mapping model places the two communicating subsystems on different nodes) and, in case the connecting network is of Ethernet type, we make use of a purposely developed Ecore model that defines mapping details, such as on what (TCP/UDP/IP) message the data signal is transmitted and with which offset and encoding, to generate automatically the network interface part of the communication.

For the communication between hand-written C++ code implementing functional subsystems and subsystems generated in SW from Simulink model, Acceleo scripts automatically generate the wrappers that provide the marshalling of parameters to the variables implementing the input ports and retrieving the data from the output port variables. The Step function implementing the subsystem runtime behavior is invoked in the context of a software thread executing at the appropriate rate.

Finally, for the case of communication between automatically generated firmware implementations and software subsystems, the read and write interface operating on the shared registers and memory locations is automatically generated using Acceleo based on the information provided in an additional Ecore mapping model, defining the position of data signals in memory and/or HW registers.

2.9 Dealing with Legacy Systems and Sub-systems

The described process is applicable to the design and development of any new system though particular attention has to be provided when dealing with already existing legacy components (either HW, SW or FW) for which no models are available. In this case our approach is to manage the legacy component as a black block and define modeling elements wrapping it. To this end we apply the *Facade* design pattern [47] each time the component does not provide a cohesive enough well structured architecture. Such solution can be permanent, for those component which are very stable and are not supposed to be extended/improved in the future, or temporary, for those components we think will need to be extended or improved in the future. In this second case we may in fact decide to start a reverse engineering process aimed at obtaining a set of SysML models describing the component itself. This decision is any time taken after an evaluation of the required extra cost compared to the return of investment that could derive.

3 A Case Study on an Electronic Warfare System

As an example of what can be achieved with our process we present results related to a project developed at ELT from 2010 to the end of 2011 called Electronic Warfare Manager (EWM [29]). The EWM realizes a Mission Computer for an electronic defence suite capable of gathering information from the available sensors (i.e. Radar Warning Receivers, Laser Warning Systems and Missile Warning Systems [28]), providing an integrated situation assessment that decides what of the available electronic countermeasures (i.e. Chaff, Jammers, etc.) to apply and managing their execution. From an industrial point of view, a fundamental requirement for the system is that it must ensure connectivity with different physical communication links on different platforms, according to the customer. Retargetability of communication interfaces was obtained by the automatic generation of IRS, IDD, and related C++ implementation from an abstract SysML model of the interfaces. By modeling each communication protocol as described in Section 2.4, more than 15K lines of code (LOC) and almost 200 pages of documentation (IRS and IDD) have been automatically generated. The code implementing these transformations is about 2,4KLOC implying that the effort has been significantly reduced. Also the remaining part of the system architecture has been modeled, as described in section 2.4, by means of SysML, MATLAB and Simulink with the automatic generation of the corresponding implementations for additional 25KLOC. Although it is in principle possible to generate almost all the required code, as of now we still have to add some glue code allowing interactions among generated components. For the EWM, this code was about 5KLOC so that the whole system is about 45KLOC, 90 percent of which has been automatically generated from models. In order to have an evaluation of the saved effort we used the Constructive Cost Model II (COCOMO II) [30] [32]. A web application provided by the Center of Systems and Software Engineering (CSSE) [31] estimates for a SW project of 40KLOC (the amount of code we automatically generated), leaving all the other COCOMO parameters as nominal, a development effort of 169.9 Person-months for a cost of 1868638$. This does not mean that 90% of this amount was actually saved, because the Inception and Elaboration phases, corresponding to the System Requirement Analysis and System Design Phases, are still (mostly) manually performed. However, the design step that is most affected by automatic code generation is the Construction phase, which is also the most expensive (estimated by the CSSE to be $1,420,166 for a duration of 12.5 months and 10.3 people involved). Accuracy of the CSSE-COCOMO estimates was confirmed from the fact that the Inception and Elaboration phase estimates turned out to be quite close to the actual effort and cost experienced at ELT.

4 Related Work

The amount of work related to our project is simply staggering. We provide some references with respect to and technologies used in the stages of the process, but there are surely many more that are omitted. The match of a functional and execution architecture is advocated by many in the academic community (examples are the Y-cycle [33] and the Platform-Based Design PBD [2]) and in the industrial domain (the AUTOSAR automotive standard [34] is probably the most relevant recent example) as a way of obtaining modularity and separation of concerns between functional specifications and their implementation on a target platform. The OMG and the MDE similarly propose a staged development in which a PIM is transformed into a Platform Specific Model (PSM) by means of a Platform Definition Model (PDM) [35]. The development of a platform model for (possibly large and distributed) embedded systems and the modeling of concurrent systems with resource managers (schedulers) requires domain-specific concepts. The OMG MARTE [10] standard is very general, rooted on UML/SyML and supported by several tools. MARTE has been applied to several use cases, most recently on automotive projects [37]. However, becasue of the complexity and the variety of modeling concepts it has to support, MARTE can still be considered an ongoing work, being constantly evaluated [36] and subject to future extensions. Several other domain-specific languages and architecture description languages of course exist, such as, for example EAST-AADL and the DoD Architectural Framework. Several other authors [38], [39] acknowledge that future trends in model engineering will encompass the definition of integrated design flows exploiting complementarities between UML or SysML and Matlab/Simulink, although the combination of the two models is affected by the fact that Simulink lacks a publicly accessible meta-model [38]. Work on the integration of UML and synchronous reactive languages [40] has been performed in the context of the Esterel language (supported by the commercial SCADE tool), for which transformation rules and specialized profiles have been proposed to ease integration with UML models [41]. With respect to the general subject of model-to-model transformations and heterogenous models integration, several approaches, methods, tools and case studies have been proposed. Some proposed methods, such as the GME framework [42] and Metropolis [43]) consist of the use of a general meta-model as an intermediate target for the model integration.Other groups and projects [44] have developed the concept of studying the conditions for the interface combatibility between etherogeneous models. Examples of formalisms developed to study the formal conditions for compatibility between different Models of Computation are the Interface Automata [45] and the Tagged Signal Language [46]. In this context our contribution is to provide an example of an actual industrial framework in which different tools and languages (i.e. DOORS, UML, SysML, MARTE, DSLs, SIMULINK, M2M and M2T Transformations,etc.) are integrated together into a single design and development workflow. Our contribution aims to show that there is not a "*ready to use*" model driven process suitable to any industry. It is instead necessary to design the process itself and properly tailor it around specific needs that could vary from a company to another. In our experience this process engineering activity can only be performed by a team with a very deep and wide knowledge of the existing technologies and methodologies together with a strong understanding of the company domain and needs.

5 Conclusions

In this paper, we presented an industrial flow and related tools featuring the integration of Model Driven Architecture and Model Based Design methodologies using a Platform-Based Design paradigm for the realization of military real-time embedded systems conforming to the MIL-STD-498 standard. The process is characterized by integration of heterogeneous languages, methods and tools, from requirements to implementation generation, in a flow in which the backbone is provided by the open source Eclipse Modeling Framework (EMF) and its metamodeling, model-to-model and model-to-code transformation capabilities. We provide system traceability from DOORS requirements to SysML design elements and system-level tests, and viceversa. In our process, the functional architecture is developed separately from the execution platform and later merged with it, according to the PBD paradigm. The development of the models for the execution platform revealed inadequacies and limitations of the standard MARTE profile, which was suitably extended and for which a new release that addresses the concerns regarding the communication modeling is strongly advocated. In our approach, almost all MIL-STD-498 compliant documentation is automatically generated from system models. Similarly, on selected case studies, about 90 percent of the target code and firmware implementations are generated from models with substantial savings. Behavioral code is generated from Simulink models, while infrastructure, communication code and tasking code is developed from SysML models and Ecore models, processed by Acceleo scripts.

References

1. Elettronica S.p.A.: http://www.elt-roma.com
2. Sangiovanni-Vincentelli, A.: Quo Vadis, SLD? Reasoning About the Trends and Challenges of System Level Design. Proceedings of the IEEE 95(3), 467–506 (2007)
3. The Object Management Group: http://www.omg.org
4. Mukerji, J., Miller, J.: Overview and Guide to OMG's Architecture, http://www.omg.org/cgi-bin/doc?omg/03-06-01
5. Paterno, F.: Model-Based Design and Evaluation of Interactive Applications. Springer, London (1999)
6. The Meta Object Facility (MOF): http://www.omg.org/spec/MOF/2.4.1
7. The UML Superstructure: http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/
8. The UML Infrastructure: http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF/
9. The System Modeling Language: http://www.sysml.org/docs/specs/OMGSysML-v1.1-08-11-01.pdf
10. Modeling Analysis of Real Time Embedded Systems (MARTE) profile: http://www.omg.org/spec/MARTE/1.0/PDF/
11. The Eclipse Modeling Framework: http://www.eclipse.org/modeling/emf/
12. Popp, P., Di Natale, M., Giusto, P., Kanajan, S., Pinello, C.: Towards a Methodology for the Quantitative Evaluation of Automotive Architectures. In: Proceedings of the Design Automation and Test in Europe Conference, Nice, April 15-18 (2007)
13. Zhu, Q., Yang, Y., Di Natale, M., Scholte, E., Sangiovanni-Vincentelli, A.: Optimizing the Software Architecture for Extensibility in Hard Real-time Distributed Systems. IEEE Transactions on Industrial Informatics 6(3) (2010)
14. TopCased: http://www.topcased.org
15. Acceleo: http://www.acceleo.org/pages/home/en
16. MOF Models to Text Transformation Language: http://www.omg.org/spec/MOFM2T/1.0/
17. Query View Transformation Language: http://www.omg.org/spec/QVT/1.0/
18. MATLAB: http://www.mathworks.it/products/matlab/
19. SIMULINK: http://www.mathworks.it/products/simulink/
20. Sindico, A., Tortora, S., Chiarini Petrelli, A., Fasano, M.V.: An Electronic Warfare Meta-Model for Network Centric Systems. In: Cognitive Information Processing, CIP (2010)
21. Sindico, A., Di Natale, M., Panci, G.: Integrating SysML With SIMULINK Using Open Source Model Transformations. In: SIMULTECH 2011, pp. 45–56 (2011)
22. IBM DOORS: http://www-01.ibm.com/software/awdtools/doors/
23. System Engineering Handbook: http://www.incose.org/ProductsPubs/products/sehandbook.aspx
24. The MIL-STD-498 Standard: http://www.letu.edu/people/jaytevis/Software-Engineering/MIL-STD-498/498-STD.pdf
25. The Requirement Interchange Format: http://www.omg.org/spec/ReqIF/1.0.1/
26. The Object Constraint Language: http://www.omg.org/spec/OCL/2.0/
27. Boost: http://www.boost.org/
28. Vakin, S.A., Shustov, L.N., Dunwell, R.H.: Fundamentals of Electronic Warfare. Artech House Radar Library (2001)
29. Tortora, S., Sindico, A., Severino, A.: A Data Fusion Architecture for an Electronic Warfare Multi-Sensor Suite. In: Cognitive Information Processing, CIP 2010 (2010)

30. Bohem, B., Clark, B., Horowitx, E., Westland, C., Madachy, R., Selby, R.: Cost models for future software life cycle processes: COCOMO 2.0. In: Annals of Software Engineering, vol. 1(1), pp. 57–94.
31. The Center of Systems and Software Engineering:
32. Boehm, B.: Software Engineering Economics. Prentice-Hall, Englewood Cliffs (1981) ISBN 0-13-822122-7
33. Kienhuis, B., Deprettere, E.F., van der Wolf, P., Vissers, K.A.: A methodology to design programmable embedded systems - the y-chart approach. In: Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation - SAMOS, pp. 18–37. Springer, London (2002)
34. Autosar, specifications 4.0 (2010), http://www.autosar.org/
35. Mellor, S.J., Kendall, S., Uhl, A., Weise, D.: MDA Distilled. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA (2004)
36. Koudri, A., Cuccuru, A., Gerard, S., Terrier, F.: Designing Heterogeneous Component Based Systems: Evaluation of MARTE Standard and Enhancement Proposal. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 243–257. Springer, Heidelberg (2011)
37. Wozniak, E., Mraidha, C., Gerard, S., Terrier, F.: A Guidance Framework for the Generation of Implementation Models in the Automotive Domain. In: EUROMICRO-SEAA 2011, pp. 468–476 (2011)
38. Vanderperren, Y., Dehaene, W.: From uml/sysml to matlab/simulink: current state and future perspectives. In: Proceedings of the Conference on Design, Automation and Test in Europe, DATE 2006, Leuven, Belgium, pp. 93–93 (2006)
39. D. B. F.I.T.T., Eda survey results (2005)
40. Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., de Simone, R.: The synchronous languages 12 years later. Proceedings of the IEEE 91(1) (January 2003)
41. Berry, G., Gonthier, G.: The synchronous programming language ESTEREL: Design, semantics, implementation. Science of Computer Programming 19(2) (1992)
42. Karsai, G., Maroti, M., Ledeczi, A., Gray, J., Sztipanovits, J.: Composition and cloning in modeling and meta-modeling. IEEE Transactions on Control System Technology (special issue on Computer Automated Multi-Paradigm Modeling) 12, 263–278 (2004)
43. Balarin, F., Lavagno, L., Passerone, C., Watanabe, Y.: Processes, interfaces and platforms. embedded software modeling in metropolis. In: Proceedings of the Second International Conference on Embedded Software. EMSOFT, pp. 407–416. Springer, London (2002)
44. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming Heterogeneity—the Ptolemy Approach. Proceedings of the IEEE 91(2) (January 2003)
45. de Alfaro, L., Henzinger, T.: Interface automata. In: Proceedings of the 8th European Software Engineering Conference, Vienna, Austria (2001)
46. Benveniste, A., Caillaud, B., Carloni, L.P., Sangiovanni-Vincentelli, A.: Tag Machines. In: Proceedings of the ACM International Conference on Embedded Software (EMSOFT 2005), Jersey City, NJ, USA (September 2005)
47. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)

Author Index