Antonio Vallecillo Juha-Pekka Tolvanen
Ekkart Kindler Harald Störrle
Dimitris Kolovos (Eds.)

LNCS 7349

Modelling Foundations and Applications

8th European Conference, ECMFA 2012
Kgs. Lyngby, Denmark, July 2012
Proceedings

Springer

Lecture Notes in Computer Science 7349

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Antonio Vallecillo Juha-Pekka Tolvanen
Ekkart Kindler Harald Störrle
Dimitris Kolovos (Eds.)

Modelling Foundations and Applications

8th European Conference, ECMFA 2012
Kgs. Lyngby, Denmark, July 2-5, 2012
Proceedings

Springer

Volume Editors

Antonio Vallecillo
Universidad de Málaga, ETSI Informática
Campus de Teatinos, Bulevar Louis Pasteur 35, 29071 Málaga, Spain
E-mail: av@lcc.uma.es

Juha-Pekka Tolvanen
MetaCase
Ylistönmäentie 31, 40500 Jyväskylä, Finland
E-mail: jpt@metacase.com

Ekkart Kindler
Harald Störrle
Technical University of Denmark
Department of Informatics and Mathematical Modelling
Richard Petersens Plads, 2800 Kgs. Lyngby, Denmark
E-mail: {eki, hsto}@imm.dtu.dk

Dimitris Kolovos
University of York, Department of Computer Science
Deramore Lane, York, YO10 5GH, United Kingdom
E-mail: d.kolovos@cs.york.ac.uk

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-31490-2 e-ISBN 978-3-642-31491-9
DOI 10.1007/978-3-642-31491-9
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2012940653

CR Subject Classification (1998): D.2.1-2, D.2.4-5, D.2.7, D.2.11, D.2, D.3, F.3, K.6

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 2012 European Conference on Modelling Foundations and Applications (ECMFA 2012) was held at the Technical University of Denmark (DTU), Kgs. Lyngby, Denmark, during July 2–5, 2012.

ECMFA is the key European conference aiming at advancing the techniques and furthering the underlying knowledge related to model-driven engineering. Model-driven engineering (MDE) is a software development approach based on the use of models for the specification, design, analysis, synthesis, deployment, testing and maintenance of complex software systems, aiming to produce high-quality systems at lower costs. In the past seven years, ECMFA has provided an ideal venue for interaction among researchers interested in MDE both from academia and industry. The eighth edition of the conference covered major advances in foundational research and industrial applications of MDE.

In 2012, the Program Committee received 106 abstracts, which finally materialized into 81 full paper submissions. From these, 20 Foundations track papers and 10 Applications track papers were accepted for presentation at the conference and publication in these proceedings. This indicates the level of competition that occurred during the selection process. The submission and the reviewing processes were administered by EasyChair, which greatly facilitated these tasks. Papers on all aspects of MDE were received, including topics such as architectural modeling and product lines, code generation, domain-specific modeling, metamodeling, model analysis and verification, model management, model transformation and simulation. The breadth of topics and the high quality of the results presented in these accepted papers demonstrate the maturity and vibrancy of the field.

The ECMFA 2012 keynote speakers were Henrik Lönn, from VOLVO Technology in Sweden, and Ed Seidewitz, from Model Driven Solutions in the USA. Abstracts of their talks are included in these proceedings. We thank them very much for accepting our invitation and for their enlightening talks.

We are grateful to our Program Committee members for providing their expertise and quality and timely reviews. Their helpful and constructive feedback to all authors is most appreciated. We thank the ECMFA Conference Steering Committee for their advice and help. We also thank our sponsors, both keynote speakers and all authors who submitted papers to ECMFA 2012. Alfred Hofmann and the Springer team were really helpful with the publication of this volume.

July 2012

Antonio Vallecillo
Juha-Pekka Tolvanen
Ekkart Kindler
Harald Störrle
Dimitris Kolovos

Organization

Program Committee

Jan Øyvind Aagedal	Norse Solutions
Vasco Amaral	FCT, Universidade Nova de Lisboa, Portugal
Terry Bailey	Vicinay Cadenas, S.A.
Stephen Barrett	Concordia University, Canada
Mariano Belaunde	Orange R&D
Reda Bendraou	INRIA Bretagne Atlantique Rennes, France
Jorn Bettin	SoftMetaWare
Xavier Blanc	Bordeaux 1 University, France
Behzad Bordbar	University of Birmingham, UK
Marco Brambilla	Politecnico di Milano, Italy
Jordi Cabot	INRIA-École des Mines de Nantes, France
Tony Clark	Middlesex University, UK
Benoit Combemale	IRIT CNRS
Diarmuid Corcoran	Ericsson AB
Zhen Ru Dai	University of Applied Science Hamburg, Germany
Juan Antonio De La Puente	Universidad Politécnica de Madrid, Spain
Zinovy Diskin	University of Waterloo, Canada
Gregor Engels	University of Paderborn, Germany
Anne Etien	University of Lille and INRIA Lille Nord-Europe, France
Stephan Flake	Orga Systems GmbH, Germany
Robert France	Colorado State University, USA
Mathias Fritzsche	SAP Research CEC Belfast, UK
Jesus Garcia-Molina	Universidad de Murcia, Spain
Sebastien Gerard	CEA, LIST
Marie-Pierre Gervais	LIP6 and Université de Paris 10 Nanterre, France
Martin Gogolla	University of Bremen, Germany
Jeff Gray	University of Alabama, USA
Esther Guerra	Universidad Autónoma de Madrid, Spain
Michael R. Hansen	Technical University of Denmark, Denmark
Reiko Heckel	University of Leicester, UK
Markus Heller	SAP Research Karlsruhe, SAP AG, Germany
Andreas Hoffmann	Fraunhofer, Germany
Teemu Kanstrén	VTT
Gabor Karsai	Vanderbilt University, USA

Thomas Kuehne	Victoria University of Wellington, New Zealand
Jochen Kuester	IBM Research
Vinay Kulkarni	Tata Research Development and Design Centre, India
Ivan Kurtev	University of Twente, The Netherlands
Roberto Erik Lopez-Herrejon	Institute for Systems Engineering and Automation, Johannes Kepler University, Austria
Dragan Milicev	University of Belgrade, Serbia
Parastoo Mohagheghi	SINTEF, Norway
Birger Møller-Pedersen	University of Oslo, Norway
Tor Neple	Norse Solutions AS
Alfonso Pierantonio	University of L'Aquila, Italy
Ivan Porres	Åbo Akademi University, Finland
Olli-Pekka Puolitaival	F-Secure Corporation
Arend Rensink	University of Twente, The Netherlands
Laurent Rioux	THALES R&T
Tom Ritter	Fraunhofer FOKUS, Germany
Louis Rose	The University of York, UK
Julia Rubin	IBM Research at Haifa, Israel
Bernhard Rumpe	RWTH Aachen University, Germany
Andrey Sadovykh	Softeam
Houari Sahraoui	DIRO, Université de Montréal, Canada
Bernhard Schaetz	TU München, Germany
Douglas Schmidt	Vanderbilt University
Andy Schürr	TU Darmstadt, Germany
Bran Selic	Malina Software Corp.
Renuka Sindhgatta	IBM Research - India
John Slaby	Raytheon Company
Jim Steel	The University of Queensland, Australia
Alin Stefanescu	University of Pitesti, Romania
Gabriele Taentzer	Philipps-Universität Marburg, Germany
Francois Terrier	CEA, LIST
Juha-Pekka Tolvanen	Metacase
Salvador Trujillo	IKERLAN Research Centre
Andreas Ulrich	Siemens AG
Antonio Vallecillo	University of Malaga, Spain
Ragnhild Van Der Straeten	Vrije Universiteit Brussel, Belgium
Pieter Van Gorp	Eindhoven University of Technology, The Netherlands
Marten J. Van Sinderen	University of Twente, The Netherlands
Hans Vangheluwe	McGill University
Daniel Varro	Budapest University of Technology and Economics, Hungary
Cristina Vicente-Chicote	Technical University of Cartagena, Spain

Markus Voelter	Independent
Michael Von Der Beeck	BMW Group
Edward Willink	Eclipse Modeling Project
Manuel Wimmer	Business Informatics Group, Vienna University of Technology, Austria
Tao Yue	Carleton University and Simula Research Laboratory
Gefei Zhang	arvato systems
Olaf Zimmermann	IBM Research GmbH
Steffen Zschaler	King's College London, UK

Additional Reviewers

Abbors, Fredrik
Al-Lail, Mustafa
Ali, Shaukat
Almeida, Marcos
Anjorin, Anthony
Aranega, Vincent
Bajwa, Imran
Bapodra, Mayur
Barat, Souvik
Barroca, Bruno
Baudry, Benoit
Blouin, Arnaud
Brucker, Achim D.
Brunelière, Hugo
Burgueño, Loli
Büttner, Fabian
Cadavid, Juan
Cichos, Harald
Criado, Javier
Cuccuru, Arnaud
Dang, Duc-Hanh
De Lara, Juan
De Mol, Maarten
Di Ruscio, Davide
Duddy, Keith
El Kouhen, Amine
Eramo, Romina
Espinazo-Pagán, Javier
Fatemi, Hassan
Fazal-Baqaie, Masud
Fritzsche, Mathias
Gerth, Christian
Haber, Arne
Hamann, Lars
Hesari, Shokoofeh
Horst, Andreas
Horváth, Ákos
Ingles-Romero, Juan F.
Iovino, Ludovico
Izsó, Benedek
Jalali, Arash
Jnidi, Rim
Khan, Tamim
Kuhlmann, Mirco
Lauder, Marius
Liu, Qichao
Look, Markus
Mallet, Frédéric
Monperrus, Martin
Noyrit, Florian
Pedro, Luís
Planas, Elena
Radermacher, Ansgar
Rath, Istvan
Rossini, Alessandro
Saller, Karsten
Sanchez, Oscar
Soltenborn, Christian
Strüber, Daniel
Sun, Wuliang
Truscan, Dragos
Wouters, Laurent
Wozniak, Ernest
Ziane, Mikal

Table of Contents

Executable UML: From Multi-domain to Multi-core

Ed Seidewitz

Vice President, Model Driven Architecture Services a Model Driven Solutions
(a division of Data Access Technologies, Inc.), United States
ed-s@modeldriven.com

Abstract. Modeling problem domains independently of technology domains is the basis for software that is adaptable to both changing business requirements and advancing technical platforms. Moreover, implementation-independent executable models allow problem-domain validation to be built right into agile conversations with customers. These validated models can then be compiled to a target implementation platform of choice.

But, unlike traditional programming, executable modeling abstracts behavior from the problem domain, rather than abstracting from hardware computational paradigms. In particular, executable models naturally embrace concurrency, because problem domain behavior is concurrent. And, as we move into an era of multiple cores, dealing with concurrency is rapidly moving from a peripheral to a central issue in mainstream programming.

Our programming languages today, on the other hand, are too platform specific, still based too much on, and abstracting too little from, traditional sequential, von Neumann hardware architectures. What we need is a way to model problem domains that can then be compiled to the highly concurrent multi-core platforms around the corner as easily as the traditional platforms of yesterday. This is exactly what executable modeling offers.

Work over the last few years has now provided new standards for precise execution semantics for a subset of UML and an associated action language. Taking advantage of these new standards, executable UML holds out the promise of addressing some fundamental issues for the next generation of programming - from multi-domain to multi-core.

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, p. 1, 2012.

Models Meeting Automotive Design Challenges

Henrik Lönn

Systems and Architecture, Department of Mechatronics & Software
Volvo Technology, Gothenburg, Sweden
henrik.lonn@volvo.com

Abstract. Automotive systems are increasingly complex and critical. Their development accounts for a considerable share of the budget, both in terms of cost and time. The development process is complex, involving multiple development teams in varying disciplines, roles, departments, companies and locations, each using their own tools and notations.

One contribution to meeting these challenges is to use a common ontology that integrates information according to recognized patterns, an Architecture Description Language. The purpose of EAST-ADL is to capture engineering information related to automotive electrical/electronic (E/E) system development, from early phase to final implementation. The system implementation is represented using AUTOSAR, i.e. EAST-ADL is complementary to AUTOSAR, adding information beyond the software architecture to serve engineering work already in early phases.

The EAST-ADL model has a core part representing the E/E system, which interfaces to an Environment model for near and far environment. Extensions for cross-cutting concerns or evolving modelling concepts annotate the core elements with these additional aspects. One of the Extensions concerns dependability and captures information related to safety. Another extension captures system timing using events, event chains and timing constraints.

The EAST-ADL system model is organized in 4 abstraction levels, see Figure 1, from the *Vehicle Levels* abstract and solution-independent feature models over the *Analysis Levels* hardware independent functional models and the *Design Levels* hardware-allocated functional architecture to the *Implementation Level* AUTOSAR software and hardware architecture.

Based on an agreed modelling approach such as AUTOSAR and EAST-ADL, research and development on modelling technology, tools and methodology for automotive EE system development can continue more efficiently. Such results will allow the multitude of company specific approaches to be leveraged and gradually replaced by off-the shelf solutions.

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 2–3, 2012.

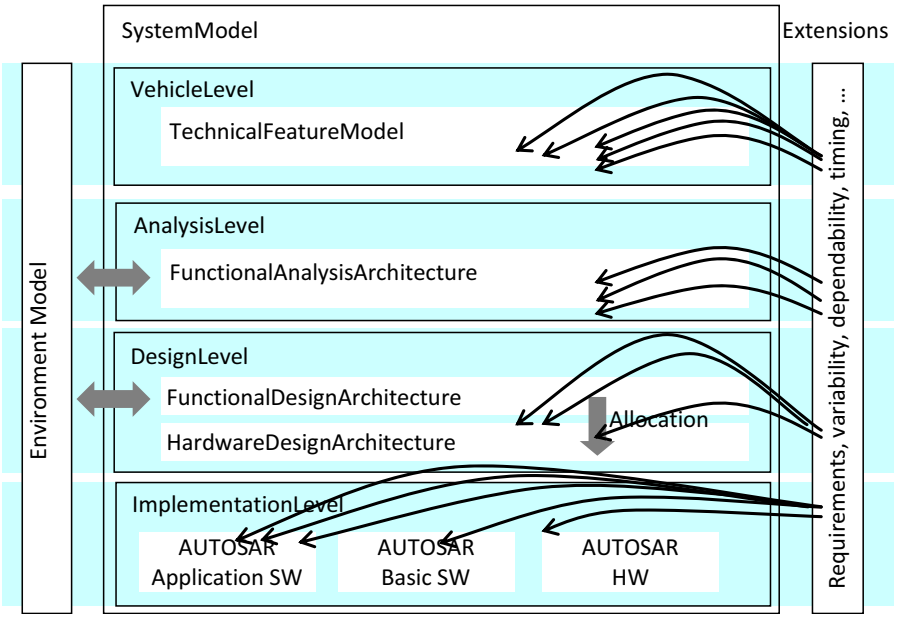


Fig. 1. EAST-ADL organization - core, plant and extensions

A Commutative Model Composition Operator to Support Software Adaptation*

Sébastien Mosser[1], Mireille Blay–Fornarino[2], and Laurence Duchien[3]

[1] SINTEF IKT, Oslo, Norway
sebastien.mosser@sintef.no
[2] I3S – UMR CNRS 7271 (formerly 6070), University Nice Sophia-Antipolis, France
blay@polytech.unice.fr
[3] INRIA Lille–Nord Europe, LIFL – UMR CNRS 8022, University of Lille 1, France
laurence.duchien@inria.fr

Abstract. The *adaptive software* paradigm supports the definition of software systems that are continuously adapted at run-time. An adaptation activates multiple features in the system, according to the current execution context (*e.g.*, CPU consumption, available bandwidth). However, the underlying approaches used to implement adaptation are ordered, *i.e.*, the order in which a set of features are turned on or off matters. Assuming feature definition as etched in stone, the identification of the *right* sequence is a difficult and time–consuming problem. We propose here a composition operator that intrinsically supports the commutativity of adaptations. Using this operator, one can minimize the number of ordered compositions in a system. It relies on an action–based approach, as this representation can support preexisting composition operators as well as our contribution in an uniform way. This approach is validated on the Service–Oriented Architecture domain, and is implemented using first–order logic.

1 Introduction

The *"adaptability"* of a software is defined through its capability to react to changes and consequently to adapt itself to new environments [24]. Adaptation is now considered as a first–class problem [19], and software must be developed with the ability of being adapted during their whole life–cycle, to properly support the emergence of new technologies and the obsolescence of legacy ones. Adaptation mechanisms strongly rely on composition operators to support the introduction (or removal) of new features inside adaptive systems [15]. For example, the detection of a sudden drop in network bandwidth turns on a `cache` feature, and thus triggers the composition of `cache` artifacts (*i.e.*, model elements)

* This work is partially funded by the EU Commission through the REMICS project (contract number 257793), the SINTEF strategic project MODERATES, the French Ministry of Higher Education and Research, Nord–Pas de Calais Regional Council and FEDER through the Contrat de Projets Etat Region Campus Intelligence Ambiante (CPER–CIA) 2007-2013.

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 4–19, 2012.

with the existing system. Existing approaches used to support such compositions rely on order–dependent operators, *e.g.*, aspect weaving [12] or functional composition [1]. Thus, the order in which features are turned on or off matters.

This order–dependency triggers several issues in the context of adaptive systems, as the designer has to explicitly control this order. The model elements associated to a feature F are composed with the existing system s as soon as the adaptation engine identifies a situation that requires F to be present in s. An immediate problem is the adaptation of unforeseen elements introduced by other features which lead to unexpected results (so-called *fragile point-cut* problem in the aspect-oriented literature [9]). Thus, the implementation of such feature assets is difficult: it must take into account the implementation of all the other feature assets to be sure that its composition produces the expected system.

In this paper, **we propose a new composition operator (called *parallel*, and denoted as ||) that allows designers to minimise the number of ordered compositions** (and the associated issues, *e.g.*, non–deterministic result if two compositions cannot commute). Using this operator, it is possible to reify that several features are turned on independently of each others, ensuring commutativity at the composition level, by construction. Such a property helps to tame the complexity of feature definition, guarantees the determinism of the computed result and also ensures the consistency of the composed system, whatever the order of composition used at the implementation level is.

2 Motivations and Challenges

Motivations. The starting point of this study is the modelling of a *Car Crash Crisis Management System* (CCCMS), started in 2010. This case study was designed as a prototypical usage of aspect–oriented modelling techniques [13], involving multiple concerns that had to be composed in a non–trivial way with respect to the requirements document. During the elaboration of our response to this case study [21], we encountered several situation where multiple and different concerns had to be composed on the same element in the original model. Actually, this situation happened 40 times in this case study, and up to 5 concerns were composed on these *shared join points* (SJP). Thus, up to $5! = 120$ combinations can be used if we consider these compositions as sequential. More critically, these sequences do not lead to the same result, as some of them cannot commute safely! The designer has to identify which order has to be used for each SJP.

The second step that triggers our research of a new operator to support composition is the study of dynamic adaptation in the context of business processes. Where the models handled by the CCCMS were "simply" design models (*i.e.*, static), we describe in [22] a process used to support the dynamic adaptation and un-adaptation of running business processes. According to a "*models@run.time*" point of view, the adaptation of a running system is assimilated to the composition of new model elements with the model associated to the running system, and the propagation of the adapted model at the run-time level. But contrarily to the CCCMS, in this case, there is no human-in-the-loop to control the order of compositions. Based on *Complex–Event Processing* (CEP) techniques, the adaptation engine automatically triggers

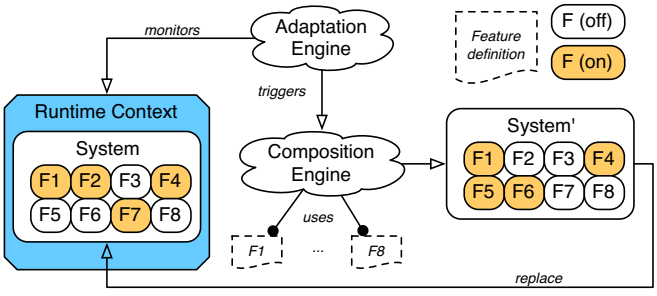


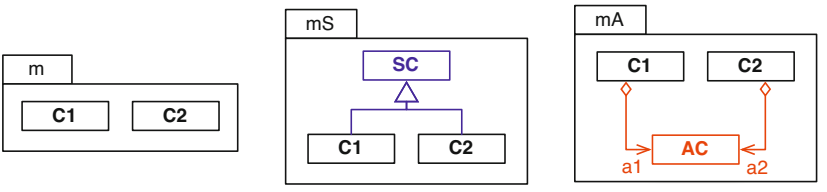
Fig. 1. Intrinsic relation between *adaptation* and *composition*

the needed composition, without any human intervention, as depicted in Fig. 1. We consider here a system s as the result of the composition of a set of features (here $s = \{F_1, F_2, F_4, F_7\}$). The adaptation engine monitors at run-time the execution context, and according to changes in this context, identify the new set of feature needed in the system ($s' = \{F_1, F_4, F_5, F_6\}$). It triggers the composition engine to properly compose all these features in order to build the adapted system s'. Then, the old system is replaced by the newly composed one, and the adaptation loop continue. As a consequence, the potential non-determinism of the composition process is a critical issue. The adaptation engine identifies a set of features needed in the system (*e.g.*, a cache, a local database, a low-energy consumption wifi driver) according to the current context, and if the composed system depends on the way these features are composed (*i.e.*, their order), the result of the adaptation process in not predictable. As a matter of fact, additional knowledge has to be *a-priori* stored in the adaptation engine to patch the composition directive generated by the CEP engine. This knowledge introduces ordering constraints needed to enact a correct sequence of compositions.

Running example. To illustrate our proposition, and for the sake of concision, we restrict the problem to its essence and use a simple model m to represent the system to be adapted. The associated class–diagram is depicted in Fig. 2(a). This model initially contains two classes C_1 and C_2. We also consider the two following adaptations:

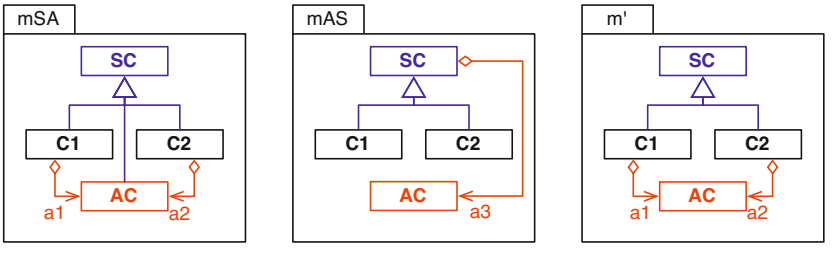
- S: This adaptation introduces a class SC in the given model, and adds an inheritance relation between all the top-level[1] classes and SC. It is a simplification of the modifications needed to introduce an *Observable* pattern into a model.
- A: This adaptation introduces a class AC in the given model, and adds an aggregation relation between all the top-level classes and AC. This adaptation can be used to add a persistence manager at the higher level of abstraction and then supports instance persistence through polymorphism.

[1] A *top–level* class is defined here as a class that does not inherits from another one, *i.e.*, at the top-level of the inheritance hierarchy.



(a) m (b) $m_S = S(m)$ (c) $m_A = A(m)$

Fig. 2. Feature composition: $F(\mu)$



(a) $m_{SA} = S(A(m))$ (b) $m_{AS} = A(S(m))$ (c) $m' = ?$

Fig. 3. Sequential composition ($S \bullet A \neq A \bullet S$)

Such adaptations represent *de facto* new features to be introduced (*i.e.*, composed) in the model. Batory *et al.* [2] use modern mathematics to model features and their introduction: *(i)* programs are constants, and *(ii)* features are functions that produce a program when applied to a program. Thus, we consider here a program tantamount a model, and we denote as $F(\mu)$ the fact that the model elements associated to the feature F are composed with the model μ (*i.e.*, F is a model transformation). We depict in Fig. 2(b) and Fig. 2(c) the two previously described features, separately composed with m.

According to this representation, the explicit ordering of feature introduction is modelled through functional composition: $(F \bullet G)(\mu) = F(G(\mu))$. In sequential composition, the commutativity of features depends on their internal definitions. As A and S both *(i)* modify all the available top-level classes and *(ii)* introduce a new one, their sequential composition cannot commute, as represented in Fig. 3(a) and Fig. 3(b). This issue highlights the fact that $\bullet$ is not commutative by essence: the order of the composition impacts the obtained model. As a consequence, this compositional model cannot be used to produce the model depicted in Fig. 3(c) without changing the implementation of A or S.

Challenges. An obvious solution is to consider that the features should not use quantified definitions (*i.e.*, avoid constructions like "for all top level classes do ...") but instantiated definitions instead (*i.e.*, use only constructions like "for C_1 and C_2, do ..."). Unfortunately, this approach scale with difficulty (in the Cccms case study, a feature had to be applied at 27 different places), and does

not allow one to reuse a feature from a system to another one (usually, these selectors are implemented as XPath expression to dynamically identify model elements). Thus, to produce the model m', where A and S are introduced independently of each others, we need to define an explicitly unordered composition operator, denoted as $||$. This operator is complementary to the $\bullet$ one, as it ensures commutativity of features composition when such a property is needed. Several challenges need to be faced to define $||$:

$\mathcal{C}_1$: *Adaptation re-usability.* To support the reuse of an adaptation, a designer must be able to define an adaptation independently of the concrete models element defined in the targeted system (*e.g.*, using quantifiers, "for all ...").

$\mathcal{C}_2$: *Adequacy with "usual" composition operator such as aspects weaving or features composition.* The key idea is not to reinvent the wheel. We aim to propose a new operator that complements the others when an ordering is not explicitly needed.

$\mathcal{C}_3$: *Adaptation Isolation & Determinism.* If at the requirements level two features are not expressed as joint (*i.e.*, there is no explicit ordering dependencies between them), the composition operator must be able to reflect this decision and consequently ensure a deterministic result.

$\mathcal{C}_4$: *Inconsistency detection.* Through usual composition operator, both $S \bullet A$ and $A \bullet S$ lead to consistent (but different) models after composition. The composition operator must be able to identify inconsistencies that can be introduced during the process, if any.

The contribution of this paper aims to tackle these challenges, through the definition of a *parallel* composition operator, denoted as $||$. We assume that the features used to implement adaptations are based on property selection (*e.g.*, "for all model elements like *this*, do *that*"), as this writing style supports feature re-usability into multiple systems ($\mathcal{C}_1$). On the one hand, if a designer knows the composition order associated to a given set of features, he/she can use existing composition operators to implement the composition ($\mathcal{C}_2$). On the other hand, when such an order is not explicit, the application of a feature F must not impact the application of feature F', for all input models ($\mathcal{C}_3$). Nevertheless, as such isolated composition may lead to inconsistencies, we provide an automated mechanism to identify inconsistencies in the composed result ($\mathcal{C}_4$).

3 An Action–Based Approach

Inspired by cutting–edges researches on the modelling research field (*e.g.*, Praxis), we use an action–based approach to represents models. According to this paradigm, *"Every model can be expressed as a sequence of elementary construction operations. The sequence of operations that produces a model is composed of the operations performed to define each model element."* [4]. This section formalises the action–orientation used to support the definition of both $||$ and $\bullet$ (formally defined in Sec. 4).

Formalising Actions. The PRAXIS method [4] defines four operations to model models, allowing one to *(i)* `create` a model element, *(ii)* `delete` a model element, *(iii)* set a property in a model element (`setProperty`) and finally *(iv)* set a reference from a model element to another one (`setReference`). We propose here a generalisation of the approach where the expressiveness is dedicated to the handling of attributed graphs. Consequently, we are not restricted to class-based models, and this definition works for any type of artifact that can be represented by a graph (the validation example relies on behavioural model initially modelled as business processes). That is, we consider a model as a set of model elements (*i.e.*, nodes), interconnected through relations (*i.e.*, edges). Sets are intrinsically unordered, and do not contain duplicates. Using first–order logic as underlying foundations, we define the following closed terms (*i.e.*, actions) to interact with a given model:

- $add_n(N, Kind)$: introduces a node N in the model. $Kind$ specializes the node (*e.g.*, a class, a UML annotation).
- $add_e(N, N', Kind)$: defines an edge from N to N' in the model. $Kind$ specializes the reified relation (*e.g.*, inheritance, aggregation).
- $del_e(N, N', Kind)$: deletes the edge from N to N', according to its $Kind$ (as several relations of different kinds may exist between two nodes).
- $del_n(N)$: deletes the node N in the model.

Models as Action Sets. We define a model as a tuple of four action sets (EQ. 1). A model $\mu \in \mathcal{M}$ is composed by *(i)* a set of node additions A_n, *(ii)* a set of edges additions A_e, *(iii)* a set of edge deletions D_e and finally *(iv)* a set of node deletions D_n. In our example, node kinds are restricted to $\{Cl\}$ (for "class"), and relation kinds are defined in $\{Ag, In\}$ (respectively "aggregation" and "inheritance"). For example, considering each class as a node, the model depicted in FIG. 2(a) is reified in EQ. 2.

$$\mu = (A_n, A_r, D_r, D_n) \in \mathcal{M} \tag{1}$$

$$m = (\{add_n(C_1, Cl), add_n(C_2, Cl)\}, \emptyset, \emptyset, \emptyset) \tag{2}$$

The union of two models is used to combine several models into a single one. It is defined as the distribution[2] of the usual set union operator into each contained set:

$$\mu = (A_n, A_e, D_e, D_n) \cup (A'_n, A'_e, D'_e, D'_n) = (A_n \cup A'_n, A_e \cup A'_e, D_e \cup D'_e, D_n \cup D'_n)$$

Relations with action sequences. We do not use plain action sequence representation to avoid permutations issues. Using such a representation, two different action sequences (s_0, s_1) that lead to the same model are considered as non–equal, where our set–driven representation (μ) ensures unicity:

$$s_0 = [add_n(a, Cl), add_n(b, class)], s_1 = [add_n(b, Cl), add_n(a, Cl)], s_0 \neq s_1$$
$$\mu = (\{add_n(a, Cl), add_n(b, Cl)\}, \emptyset, \emptyset, \emptyset)$$

[2] One can notice that such a distribution can also be used to implement others model combination operator (*e.g.*, intersection, difference).

The underlying idea is that a sequence of actions always respects the following steps: it *(i)* adds nodes, *(ii)* adds edges between these nodes, *(iii)* deletes existing edges and finally *(iv)* deletes isolated nodes. In a given step, the internal ordering does not matter (adding x before y is not relevant with regard to the final model). Thus, one can see our representation as a canonical form of an action sequence mandatory to build a given model: the division into four subsets supports this partial ordering.

Consistency. Using this representation, model consistency is ensured according to several logical rules. As the detection of inconsistencies in models is a dedicated research field [3], we always assume in this paper that the handled models are *consistent*. For a given model $\mu = (A_n, A_r, D_r, D_n)$, this property is ensured according to the following rules:

P_1: *"Related elements existence"*. An action that adds a relation between two elements (here classes) C and C' assumes that these two classes are added with the associated actions in A_n (Eq. 3).

P_2: *"Deletion of existing relations"*. An action that deletes a relation between two elements C and C' with a given $Kind$ assumes that this relation is added in A_e (Eq. 4).

P_3: *"Deletion of isolated elements"*. An action that deletes an element C assumes that all relations involving C are deleted in D_e (Eq. 5).

$$add_e(C, C', \_) \in A_e \Rightarrow add_n(C, Cl) \in A_n \wedge add_n(C', Cl) \in A_n \tag{3}$$

$$del_e(C, C', k) \in D_e \Rightarrow add_e(C, C', k) \in A_e \tag{4}$$

$$del_n(C) \in D_n \Rightarrow \begin{cases} \forall add_e(C, X, K_{cx}) \in A_e, \exists del_e(C, X, K_{cx}) \in D_e \\ \wedge\ \forall add_e(Y, C, K_{yc}) \in A_e, \ \exists del_e(Y, C, K_{yc}) \in D_e \end{cases} \tag{5}$$

4 Using Actions to Support || and •

In this section, we present how the model representation described in the previous section supports feature introduction, and the definition of both • and || operators.

Using Actions to Introduce Features. Using a functional approach, base models are considered as *constants* (*e.g.*, $\mu \in \mathcal{M}$), and features are defined as *functions* that map an input model μ into an enriched model μ'. Thus, introducing a feature F into a model μ means to use the latter as the input of the former: $\mu' = F(\mu)$. In our approach, we propose to consider F as a two steps function: *(i)* the copy of the input model μ into the output one and *(ii)* the generation of the actions necessary to modify μ and then produce the expected model as output, denoted as $\Delta_F(\mu)$. Our action–based representation of models allows the designer to represent these elements in an endogenous way, as both μ and $\Delta_F(\mu)$ are modelled as sets of actions. Thus, we obtain μ' as the following: $\mu' = F(\mu) = \mu \cup \Delta_F(\mu)$.

For example, we consider here the feature S described in the previous section. Assuming a function named *top* that returns the set of top-level classes discovered in its input, one can implement Δ_S as the following: for an input model μ, it *(i)* adds the SC class and then *(ii)* generates the addition of an inheritance relation between all top-level classes and SC.

$$\begin{aligned}\Delta_S : \mathcal{M} &\to \mathcal{M}\\ \mu &\mapsto (\{add_c(SC, Cl)\}, \{add_e(X, SC, In) | X \in top(\mu)\}, \emptyset, \emptyset)\end{aligned}$$

Thus, the introduction of S in m (Fig. 2(b)) is modelled as the following:

$$\begin{aligned}m &= (\{add_n(C_1, Cl), add_n(C_2, Cl)\}, \emptyset, \emptyset, \emptyset)\\ \Delta_S(m) &= (\{add_c(SC, Cl)\}, \{add_e(C_1, SC, In), add_e(C_2, SC, In)\}, \emptyset, \emptyset)\\ m_S &= S(m) \;=\; m \cup \Delta_S(m)\\ &= (\{add_c(SC, Cl), add_n(C_1, Cl), add_n(C_2, Cl)\},\\ &\quad \{add_e(C_1, SC, In), add_e(C_2, SC, In)\}, \emptyset, \emptyset)\end{aligned}$$

As said in the consistency paragraph, we assume to work with consistent models and features. Thus, the introduction of a feature into a model always leads to a *consistent* model: let μ a consistent model, and F a given feature. Even if $\Delta_F(\mu)$ may be inconsistent (*e.g.*, deleting a class that is added in μ and not in $\Delta_F(\mu)$, violating P_3), the result of its union with m is therefore consistent.

4.1 Sequential Composition: •

Using F and G as two features, and μ a model, we define the functional composition operator • as the following:

$$\begin{aligned}G(\mu) &= \mu \cup \Delta_G(\mu)\\ (F \bullet G)(\mu) &= F(G(\mu)) = G(\mu) \cup \Delta_F(G(\mu)) = \underbrace{\mu \cup \Delta_G(\mu)}_{G(\mu)} \cup \Delta_F(\underbrace{\mu \cup \Delta_G(\mu)}_{G(\mu)})\end{aligned}$$

The operator holds the following properties, and thus behaves as the "usual" operator:

- Identity: Let Id be the identity feature[3], and F a given feature. $F = F \bullet Id$.
- Idempotence: Let F be a feature. In the general case, $F(F(\mu)) \neq F(\mu)$
- Commutativity: this property cannot be ensured in the general case, and its implementation–dependent. It can be ensured if and only if the two functions F and G are orthogonal. In the general case, $F \bullet G(\mu) \neq G \bullet F(\mu)$.

Running example. We now consider Δ_A, the function used to implements the previously defined A feature:

$$\begin{aligned}\Delta_A : \mathcal{M} &\to \mathcal{M}\\ \mu &\mapsto (\{add_n(AC, class)\}, \{add_e(X, AC, Ag(\_)) | X \in top(\mu)\}, \emptyset, \emptyset)\end{aligned}$$

[3] $\forall \mu \in \mathcal{M},\ \Delta_{Id}(\mu) = (\emptyset, \emptyset, \emptyset, \emptyset) \Rightarrow Id(\mu) = \mu$.

With this function and the previously defined one Δ_S, one can build the model m_{SA} depicted in Fig. 3(a), which represents $(S \bullet A)(m)$.

$$
\begin{aligned}
\Delta_A(m) &= (\{add_n(AC, Cl)\}, \{add_e(C_1, AC, Ag(a_1)), add_e(C_2, AC, Ag(a_2))\}, \emptyset, \emptyset) \\
m_A = A(m) &= m \cup \Delta_A(m) \\
&= (\{add_n(AC, Cl), add_n(C_1, Cl), add_n(C_2, Cl)\}, \\
&\quad \{add_e(C_1, AC, Ag(a_1)), add_e(C_2, AC, Ag(a_2))\}, \emptyset, \emptyset) \\
top(m_A) &= \{AC, C_1, C_2\} \\
\Delta_S(m_a) &= (\{add_n(SC, Cl)\}, \\
&\quad \{add_e(C_1, SC, In), add_e(C_2, SC, In), add_e(AC, SC, In)\}, \emptyset, \emptyset) \\
m_{SA} = S(A(m)) &= m \cup \Delta_A(m) \cup \Delta_S(m \cup \Delta_A(m)) = m_A \cup \Delta_S(m_A) \\
&= (\{add_n(AC, Cl), add_n(C_1, Cl), add_n(C_2, Cl), add_n(SC, Cl)\}, \\
&\quad \{add_e(C_1, AC, Ag(a_1)), add_e(C_2, AC, Ag(a_2)), \\
&\quad\ add_e(C_1, SC, In), add_e(C_2, SC, In), add_e(AC, SC, In)\}, \emptyset, \emptyset)
\end{aligned}
$$

4.2 Parallel Composition: ||

Using F and G as two features, and μ a model, we define the parallel composition operator || as the following:

$$
\begin{aligned}
F(\mu) = \mu \cup \Delta_F(\mu),\ G(\mu) = \mu \cup \Delta_G(\mu) \\
(F||G)(\mu) = F(\mu) \cup G(\mu) = \mu \cup \Delta_F(\mu) \cup \Delta_G(\mu)
\end{aligned}
$$

As the || operator is defined over set union, it holds the following properties:

- Identity: considering Id as the identify feature and F as a given feature, $(F||Id)(\mu) = \mu \cup \Delta_F(\mu) \cup (\emptyset, \emptyset, \emptyset, \emptyset) = F(\mu)$.
- Idempotence: let F be a given feature. $F||F(\mu) = F(\mu) \cup F(\mu) = F(\mu)$.
- Commutativity: the operator relies on set union, which is commutative. $(F||G)(\mu) = (G||F)(\mu) = \mu \cup \Delta_F(\mu) \cup \Delta_G(\mu)$.

When applied to the previous example, one can build the model m' depicted in Fig. 3(c) as the following:

$$
\begin{aligned}
m' &= (A||S)(m) = m \cup \Delta_A(m) \cup \Delta_S(m) \\
&= (\{add_n(AC, Cl), add_n(C_1, Cl), add_n(C_2, Cl), add_n(SC, Cl)\}, \\
&\quad \{add_e(C_1, AC, Ag(a_1)), add_e(C_2, AC, Ag(a_2)), \\
&\quad\ add_e(C_1, SC, In), add_e(C_2, SC, In)\}, \emptyset, \emptyset)
\end{aligned}
$$

4.3 Impact on Model Consistency

The previously described definition of feature composition assumes their consistency: for a given model μ and any feature F, the composition of F with μ always leads to a consistent model (*i.e.*, a model that respects the P_i constraints).

•: the sequential composition operator is consistent by construction: it simply chains the compositions.
||: the parallel composition operator works on a different basis (*i.e.*, model union), and then may lead to inconsistent models.

Let F and G two consistent features. For a given model μ, we ensure by construction that both $F(\mu)$ and $G(\mu)$ are also consistent. But their parallel composition $\mu' = F(\mu) \cup G(\mu)$ may be inconsistent, according to the following rules:

P_1: *"Related elements existence"*. This property is violated if and only if a feature adds a relation that involves an element deleted by another feature.
P_2: *"Deletion of existing relations"*. This property can be violated if and only if a feature F deletes a relations added in another feature F'. Such a situation also implies a violation of P_1 (F' defines a relation between unknown elements).
P_3: *"Deletion of isolated elements"*. This property is violated since a feature deletes an element used by the other one in a newly added relation (see P_1).

In fact, the computation of an inconsistent resulting model (*after* the composition) identifies an issue in the features: they cannot be composed in parallel as is, as one relies on the other. It is then a typical use case for a sequential composition (•). It tackles challenge $\mathcal{C}_4$, as such erroneous situation can be automatically detected (*e.g.*, through the satisfaction of a logical predicate).

5 Implementation and Validation

In this section, we describe how the approach is implemented in a logical language, and emphasize the need for using the || operator in the context of a complex case study.

5.1 Implementation

We provide a reference implementation of the approach[4]. This framework supports the definition of features as Prolog predicates, and includes a *Domain–Specific Language* (DSL) to express compositions. This language is domain independant, as it relies on the action sequences previously defined, reifying models as attributed graphs. The engine compiles compositions expressed through the DSL into logical predicates (using ANTLR[5]), and supports their execution in a Prolog interpreter (SWI-Prolog[6]). At run-time, SWI-Prolog provides the JPL framework, which implements a bidirectional Java/Prolog bridge. Thus, the engine can be connected to any tool reachable through the Java language, *e.g.*, the *Eclipse Modeling Framework* (EMF). The implementation of the running example used in this paper is available in the code repository[7].

[4] http://www.gcoke.org
[5] http://www.antlr.org/ (version 3.3)
[6] http://www.swi-prolog.org/ (version 5.10.4)
[7] http://code.google.com/p/gcoke/source/browse/trunk/lines/ase_xp/

Listing 1.1. Ordered composition ($\bullet$): $m_{sa} \neq m_{as}$

```
1 composition ordered(m) {
2   a(model: m)    => (output: m_a);
3   s(model: m_a) => (output: m_sa);
4 } => (m_sa);
```

Listing 1.2. Parallel composition ($||$): $m' = (A||S)(m) = (S||A)(m)$

```
1 composition parallel(m)  {
2   s(model: m) => (output: m_p);
3   a(model: m) => (output: m_p);
4 } => (m_p);
```

Using the engine, one can express compositions using the DSL. We represent in Lst. 1.1 how the framework supports ordered compositions ($\bullet$). A composition is named (here `ordered`), and consumes models (here `m`) to produces new ones (here `m_sa`). The way such models are produced is represented as a set of composition directives: line 2 implements the application of the feature `a` using `m` as input `model`, and storing its `output` into `m_a`. A directive is triggered as soon as all its input artifacts are available (*i.e.*, existing or computed by another directive). The parallel composition operator is implemented as the absence of order between directives (Lst. 1.2, next page). In a composition named `parallel`, we only declare that `s` and `a` use the model `m` as input, and store their result in `m_prime` (lines 2 and 3). In front of such a declaration, the engine computes each set of actions independently, and will perform the union of the generated actions sequences before executing it. If an inconsistency is detected (which is not the case here), an error is raised to the designer.

5.2 Validation

We focus here on the Cccms case study, as it is the largest one we used to validate the $||$ operator. The case study was designed by Kienzle *et al.* [13] as a reference framework to compare different aspect–oriented modelling approaches. This example is a *real-life* example, involving thousands of model elements according to real-life business processes. In this context, the considered models to be composed reify business processes, *i.e.*, behavioural models. The business processes involved in the Cccms[8] are modelled as graphs, where nodes are activities and relations implement a partial order between these activities. The case study defines hundreds of activities scheduled by thousands of relations, which makes the example suitable for "real–life" complexity. We instantiated two variants of the requirements: *(i)* a system that only fits the *business* requirements and *(ii)* a system that includes several *non–functional* (NF) concerns. The final system

[8] http://www.adore-design.org/doku/examples/cccms/start

(including NF concerns) defines 146 compositions. As stated in the motivations of this paper, we identified up to 40 shared join points in this study ($\sim$ 27%). On these points, up to 5 concerns had to be composed, leading to 120 potential sequences of composition. This situation triggers a humongous amount of verifications to be checked on the composed system, which is modelled as a set of dense graphs (hundreds of nodes, thousands of relations). Thus, the execution of checkers to verify the consistency of the composed system costs a lot of resource and CPU–time, as the verifications rely on the systematic check of each path defined in a given graph (subject to combinatorial explosion).

While designing the Cccms, instead of systematically using the $\bullet$ operator and manage all the complexity by hand, we used the || operator to support the compositional approach. The requirement document stated that the features were supposed to be orthogonal, and as a consequence the || operator perfectly implements this intention. The inconsistency detection mechanism (applied on action sequence instead of large graphs) was then used to identify the situations where an order should be defined. Results are summarised in Tab. 5.2.

- The business version uses 24 features and defines 28 composition directives to build the complete system. It can then be considered as a large simplification of the expected system. In this version, only 2 composition directives where identified as conflicting, and actually had to be explicitly ordered (*i.e.*, implements a $\bullet$ composition). All the other compositions can be computed independently. This point illustrates that from a business point of view, the absence of ordering is really important. Applying these features as an ordered sequence can produce unexpected results, like the ones shown in Fig. 3. Through sequential composition, designers would had to *(i)* check the composed system to verify that the obtained result does not contain such feature interactions and/or *(ii)* avoid the usage of quantifiers to anticipate such situations.
- The introduction of NF concerns includes in our case five additional features, dealing with security, persistence and statistical logging. In this configuration, we use 146 composition directives to build the complete system (business + NF). Up to 73% of these directives were unordered in this case study. The others requires an order to meet the requirement specifications. For example, we had to introduce security features after all the others to secure the complete process. It is important to notice that this need was not explicitly documented in the requirements, but accurately detected by the inconsistency detection mechanism. This point highlights the complementarity of the sequential and parallel composition operators.

Table 1. Composition directives used in the Cccms

System	#Composition	#Ordered	#Parallel
Business	28	2 (7%)	26 (93%)
Business + NF	146	39 (27%)	107 (73%)

6 Related Works

Modern mathematics were proposed as a support of feature–oriented software development [2]. This algebraic representation allows the usage of equation optimisers to rewrite the compositions in an efficient way [16]. It is possible to reify the interaction of a feature and another one [17] using mathematical derivative function. Another lead is to use commuting diagrams [14] to explore the different composition orders. The way features are composed together can be constrained through the usage of *design constraint rules*, expressed as attributed grammars [18]. A "valid" composition is consequently identified as a word recognised by the design constraint grammar (identifying conflicting features upstream). The contribution of this paper complements these works, as it also reify compositions as mathematical expressions. The major difference with these works is the definition of a commutative and idempotent composition operator.

The opposite approach of the one described in this paper is to analyse the set of available features and to automatically identify the needed composition order, as implemented by the CAPucine framework [25]. Using CAPucine, a *Feature Diagram* (FD [8]) is used to express the business variability of a given system. Using an aspect–oriented modelling approach, features are bound to assets that implement aspect models: a fragment of model to be added (*i.e.*, advice) and a selector used to identify where this fragment should be added (*i.e.*, point-cut). CAPucine analyses the given elements according to two directions: from the FD to the models and *(i)* from the models to the FD. On the one hand, the latter analyses the set of selectors against the set of model fragments, and identifies hidden dependencies between features that were not expressed in the FD. On the other hand, the former verifies for each constraints expressed in the FD (*e.g.*, "F requires F'") if the implementation follows it (*i.e.*, the selector defined in F matches elements defined in the fragment of model associated to F'). These analysis are complementary to the parallel composition operator, as one can use it to automatically discriminate the features that requires a sequential ($\bullet$) composition and the features that rely on the parallel operator ($||$). Thus, it is possible to *(i) detect* hidden ordering with CAPucine and *(ii) ensure* that others features are composed in isolation.

Another lead to ensure commutative composition is followed by the model transformation community [5]. In this work, the key idea is to analyse the set of model elements impacted (*e.g.*, read, modified, deleted) by a given transformation τ, and then reason about these different sets to check if two transformations may commute. This reasoning capabilities are formalised using set theory, and dedicated to model–transformation. Such an analysis ensures the consistency of models after a parallel composition. Thus, this approach complements ours: *a posteriori* inconsistency detection can be avoided at run-time if commutativity safety can be proved. However, our composition operator *ensures* the parallel application of a set of features, by construction, whatever their definition.

Model weaving can also be considered as a way to support adaptation. This paradigm relies on aspect weaving at the model level. In this context, it is possible to use optimisation techniques to select the best model to be woven with

the current one [26]. But intrinsically, these approaches implements an aspect weaving algorithm, which is by nature not commutative. Our approach is thus complementary to these ones, as one can implement our || operator in such a framework and then support unordered composition.

More specifically, the MATA approach [27] supports the weaving of models aspects using a graph–based approach. This approach supports powerful conflict detection mechanisms, used to support the "safe" composition of models [23]. The underlying formal model associated to this detection is based on critical pair analysis [7]. Initially defined for term rewriting systems and then generalised to graph rewriting systems, critical pairs formalise the identification of a minimal example associated to a potentially conflicting situation. This notion supports the development of rule–based systems, identifying conflicting situations such as "the rule r will delete an element matched by the rule r'" or "the rule r generates a structure which is prohibited according to the existing preconditions". This work is complementary with the one presented in this paper, as it can be used to handle inconsistencies in a more detailed way.

7 Conclusions and Perspectives

In this paper, we introduced a new composition operator (denoted as ||), that enables the parallel composition of existing features. Using an action–based approach, we formally defined this new operator and the existing ones (*e.g.*, sequential), as well as its prototypical implementation using a logical language. We identified four challenges, accurately tackled by the approach. The operator supports feature re-usability ($\mathcal{C}_1$), and complements the existing ones (to be used when an order is needed, $\mathcal{C}_2$). It also ensures determinism in the composition ($\mathcal{C}_3$), as the composition order does not matter when || is used. Finally, inconsistency detection mechanisms are provided to ensure the safety of the parallel composition ($\mathcal{C}_4$). The operator was validated in the context of SOA business processes, illustrating how it scales in front of large systems.

Immediate perspectives of this work are to apply the operator to multiple application domains. We plan to focus on the two following research fields, which highly rely on compositions to support their adaptation: *(i)* Cloud–computing and *(ii)* Internet of Things. For the former, it is known that the design of efficient distributed systems is a tedious task. The use of composition algorithms to support their adaptation according to a step-wise approach tames such a complexity, and ensure properties in the composed result (difficult to be checked by humans). In the context of the REMICS[9] project, we are dealing with the migration of legacy systems into cloud-based application. In this context, the need for adaptation is double: *(i)* models of legacy applications have to be adapted *w.r.t* models of clouds to enact a cloud version of the application, and *(ii)* at run-time, run-time models have to be adapted to accurately use the power of the cloud (*e.g.*, "elasticity", which refers to an unlimited resource provisioning capability). The Internet of Things domains is driven by the multiplication of embedded

[9] http://remics.eu/, EU FP7, STREP.

devices (*e.g.*, sensors, smartphone, Pda, tablet PC). Intrinsically, the Internet of Things aims to compose multiple devices into an autonomic entity, able to reconfigure itself at run-time [6], according to changes in its environment (*e.g.*, a more accurate display device is discovered, and the application is reconfigured to broadcast the main content to this new device) [20]. These two application domains will support large-scale experimentation of the || operator, based on real case studies provided by industrial partners.

References

1. Batory, D., Sarvela, J.N., Rauschmayer, A.: Scaling Step-Wise Refinement. IEEE Transactions on Software Engineering 30, 2004 (2004)
2. Batory, D.S.: Using Modern Mathematics as an FOSD Modeling Language. In: Smaragdakis, Y., Siek, J.G. (eds.) GPCE, pp. 35–44. ACM (2008)
3. Blanc, X., Mougenot, A., Mounier, I., Mens, T.: Incremental Detection of Model Inconsistencies Based on Model Operations. In: van Eck, P., Gordijn, J., Wieringa, R. (eds.) CAiSE 2009. LNCS, vol. 5565, pp. 32–46. Springer, Heidelberg (2009)
4. 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)
5. Etien, A., Muller, A., Legrand, T., Blanc, X.: Combining Independent Model Transformations. In: Shin, S.Y., Ossowski, S., Schumacher, M., Palakal, M.J., Hung, C.C. (eds.) SAC, pp. 2237–2243. ACM (2010)
6. Fleurey, F., Morin, B., Solberg, A.: A Model-driven Approach to Develop Adaptive Firmwares. In: Giese, H., Cheng, B.H.C. (eds.) SEAMS, pp. 168–177. ACM (2011)
7. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of Typed Attributed Graph Transformation Systems. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 161–176. Springer, Heidelberg (2002)
8. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-Oriented Domain Analysis (FODA) - Feasibility Study. Tech. rep., The Software Engineering Institute (1990), http://www.sei.cmu.edu/reports/90tr021.pdf
9. Kastner, C., Apel, S., Batory, D.: A Case Study Implementing Features Using AspectJ. In: Proceedings of the 11th Int. Software Product Line Conference, pp. 223–232. IEEE Computer Society, Washington, DC (2007)
10. Katz, S., Mezini, M., Kienzle, J. (eds.): Transactions on Aspect-Oriented Software Development VII - A Common Case Study for Aspect-Oriented Modeling. LNCS, vol. 6210. Springer, Heidelberg (2010)
11. Katz, S., Ossher, H., France, R., Jézéquel, J.-M. (eds.): Transactions on Aspect-Oriented Software Development VI, Special Issue on Aspects and Model-Driven Engineering. LNCS, vol. 5560. Springer, Heidelberg (2009)
12. 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. 327–353. Springer, Heidelberg (2001)
13. Kienzle, J., Guelfi, N., Mustafiz, S.: Crisis Management Systems: A Case Study for Aspect-Oriented Modeling. In: T. Aspect-Oriented Soft. Dev. [10], pp. 1–22
14. Kim, C.H.P., Kästner, C., Batory, D.: On the Modularity of Feature Interactions. In: Procs of the 7th Int. Conf. on Generative Programming and Component Engineering, pp. 23–34. ACM, New York (2008)

15. Kniesel, G.: Type-Safe Delegation for Run-Time Component Adaptation. In: Guerraoui, R. (ed.) ECOOP 1999. LNCS, vol. 1628, pp. 351–366. Springer, Heidelberg (1999), doi:10.1007/3-540-48743-3_16
16. Liu, J., Batory, D.: Automatic Remodularization and Optimized Synthesis of Product-Families. In: Karsai, G., Visser, E. (eds.) GPCE 2004. LNCS, vol. 3286, pp. 379–395. Springer, Heidelberg (2004)
17. Liu, J., Batory, D.S., Nedunuri, S.: Modeling Interactions in Feature Oriented Software Designs. In: Reiff-Marganiec, S., Ryan, M. (eds.) FIW, pp. 178–197. IOS Press (2005)
18. McAllester, D.: Variational Attribute Grammars for Computer Aided Design (Release 3.0). Tech. rep. MIT (1994)
19. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C.: Composing Adaptive Software. Computer 37, 56–64 (2004)
20. Morin, B., Barais, O., Jezequel, J.M., Fleurey, F., Solberg, A.: Models@ Run.time to Support Dynamic Adaptation. Computer 42(10), 44–51 (2009)
21. Mosser, S., Blay-Fornarino, M., France, R.: Workflow Design Using Fragment Composition – Crisis Management System Design through ADORE. In: T. Aspect-Oriented Software Development [10], pp. 200–233
22. Mosser, S., Hermosillo, G., Le Meur, A.F., Seinturier, L., Duchien, L.: Undoing Event-Driven Adaptation of Business Processes. In: 8th International Conference on Services Computing (SCC 2011), pp. 1–8. IEEE, Washington DC (2011)
23. Mussbacher, G., Whittle, J., Amyot, D.: Semantic-Based Interaction Detection in Aspect-Oriented Scenarios. In: RE, pp. 203–212. IEEE Computer Society (2009)
24. Oreizy, P., Medvidovic, N., Taylor, R.N.: Runtime Software Adaptation: Framework, Approaches, and Styles. In: Companion of the 30th Int. Conf. on Software Engineering, ICSE Companion 2008, pp. 899–910. ACM, New York (2008)
25. Parra, C., Cleve, A., Blanc, X., Duchien, L.: Feature-Based Composition of Software Architectures. In: Babar, M.A., Gorton, I. (eds.) ECSA 2010. LNCS, vol. 6285, pp. 230–245. Springer, Heidelberg (2010)
26. White, J., Gray, J., Schmidt, D.C.: Constraint-Based Model Weaving. In: T. Aspect-Oriented Software Development VI [11], pp. 153–190
27. Whittle, J., Jayaraman, P.K., Elkhodary, A.M., Moreira, A., Araújo, J.: MATA: A Unified Approach for Composing UML Aspect Models Based on Graph Transformation. In: T. Aspect-Oriented Software Development VI [11], pp. 191–237

Comparative Study of Model-Based and Multi-Domain System Engineering Approaches for Industrial Settings

Anjelika Votintseva, Petra Witschel, Nikolaus Regnat, and Philipp Emanuel Stelzig

Siemens AG, Otto-Hahn-Ring 6, Munich, Germany
{anjelika.votintseva,petra.witschel,
nikolaus.regnat,philipp.stelzig}@siemens.com

Abstract. A typical approach for the development of multi-domain systems often carries the risk of high non-conformance costs and time-consuming re-engineering due to the lack of interoperability between different domains. In its research project "Mechatronic Design", the Siemens AG develops an integrated, model-based and simulation-focused process to perform a frontloading engineering approach for multi-domain systems.

The paper presents two use cases from this project as two implementation approaches to system modeling and simulation being synchronized at early design phases. Both use cases utilize the standardized system modeling language SysML and the multi-domain simulation language Modelica. One use case evaluates the standardized OMG SysML4Modelica profile for transformation between SysML and Modelica. The other use case uses a Modelica independent and proprietary profile aiming at more flexible usage. For both approaches, advantages and disadvantages are identified and compared. Depending on the project objectives, the general suitability of the approaches is also judged.

Keywords: model-based system engineering, simulation, multi-domain systems, SysML, Modelica, comparative study, industrial use cases.

1 Introduction

Model-based systems engineering (MBSE) is the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases [1]. It is especially challenging to manage the development of multi-domain systems. In the present context, by multi-domain systems we understand those involving both multiple technical disciplines (like mechanics, electrics, software) and different engineering methods (e.g. in this context, architecture description, system simulation). MBSE enables an integrated process to develop constituent parts of such systems in an integrated way. In addition to descriptive models, using executable models allows system architects to run simulations. Within the internal company-wide research project of Siemens AG called "Lighthouse Project Mechatronic Design" (LHP MDE), an integrated, model-based and simulation-focused process is under development to perform a frontloading engineering approach for multi-domain products.

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 20–31, 2012.

Our paper exemplifies how system modeling and system simulation can be used and synchronized at early design phases for concept evaluation and functional design. With the design of an electrical car (eCar) and the development of medical equipment (Artis Floorstand), two use cases of LHP MDE are examined in more details, where different development groups follow different approaches. For description of the system model, in the both use cases the standardized system description language SysML [2] has been used which is based on the unified modeling language UML [3]. SysML is currently becoming an industrial standard in the domain of system modeling as it allows consistent structuring of complex systems with easily understandable graphical visualization. The quantitative analysis has in both cases been performed with the Modelica language [4]. Modelica is very suitable for multi-domain simulations, as it allows to model – in an object-oriented way – any system that can be described by differential algebraic equations. It is possible to create Modelica models graphically and textually, what makes it particularly well-suited for automatic transformations. SysML and Modelica are increasingly used in different projects at Siemens AG.

Research is already performed to combine UML-based system description with Modelica-based simulation. In [5] a UML Profile called ModelicaML is presented which enables integrated modeling and simulation of system requirements and design. Another profile named SysML4Modelica [6] concentrates on the combination between SysML and Modelica. Problems in developing complex multi-domain systems are tackled in further works. For example, Model Integrated Mechatronics (MIM) [7] is an architecture that promotes model integration for different kinds of artifacts allowing concurrent engineering of mechanical, electronic and software components. It simplifies the integrated development process by using the construct of Mechatronic Components. The Functional (Digital) Mockup [8] approach is synergistic design synchronization, model execution and analysis, providing a tight integration of mechanics with electronics and software and a smooth integration of dependability predictions during the early development phases.

This paper is structured as follows. In Section 2, the two application areas eCar and Artis Floorstand are introduced together with their development processes. Section 3 contains implementation details. The comparison of the proposed approaches is performed in Section 4. Section 5 concludes and gives an outlook for future works.

2 Application Areas and Proposed Development Processes

This section provides an overview of the selected use cases – eCar and Artis Floorstand. The development processes proposed within these use cases are outlined together with exemplary workflows justifying the chosen ways of tool integration.

2.1 Use Case eCar

The use case eCar is related to the development of a hypothetical electrical vehicle [9]. In our sample model we concentrate on the level of functional architecture and abstract logical design to show how architectural decisions at early development stages can be analyzed and compared. As an example of an early decision, we consider

the question if a concept with one or two electric motors is more advantageous. In one concept, a single electrical motor, connected via a mechanical differential to the front wheels of the vehicle, is used. In the other concept, two independent motors are individually attached to the front wheels. This choice of concepts visualized in Fig. 1 (left-hand) has impact on a variety of non-functional requirements like efficiency, battery range, drive comfort and costs. One focus of our works was the evaluation of battery consumption under control of a new SW component, an adaptive cruise control (ACC). The car in front was assumed to drive according to the New European Drive Cycle (NEDC) shown in Fig. 1 (right-hand). This example demonstrates the integration of software and environment models in modern multi-domain systems.

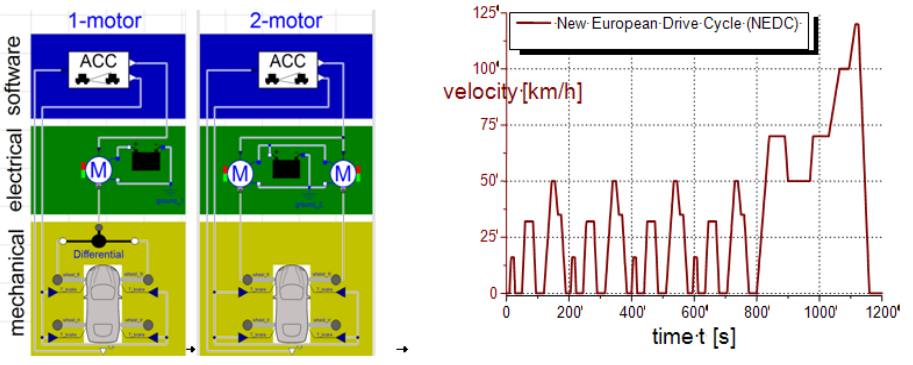


Fig. 1. Abstract models for the sample multi-domain system eCar and an input drive cycle

In this use case, the standardized SysML4Modelica profile [6] was evaluated. The specification of the profile describes how to represent elements of Modelica within SysML. Using this approach, a system architect can select an appropriate level of detail, which is then transformed automatically into Modelica.

Development Process

A typical eCar development process contains such phases as requirements, functional architecture, logical architecture, physical and software design, integration and testing activities. A generic system model can be configured into different product configurations containing instances of physical elements and software implementations, as well as other product parameters. After a product configuration is specified within SysML, some parts of the model required for the further analysis are decorated with the domain-specific stereotypes and tags (in this case from the SysML4Modelica profile). This can be applied to whole components or just to some model elements (blocks or their properties). After that, Modelica simulation models are generated automatically for different product configurations. Components from software design are inserted into UML projects where they are developed further. Selected results from the domain-specific analysis are transferred back to the system model, e.g. as product parameters or component properties.

Exemplary Workflow

In the use case eCar, we have assumed a workflow starting from SysML, where the different kinds and levels of the system architecture are developed and different kinds of analysis are managed (e.g., validation activities are specified and mapped to tools, dependencies are defined between artifacts).

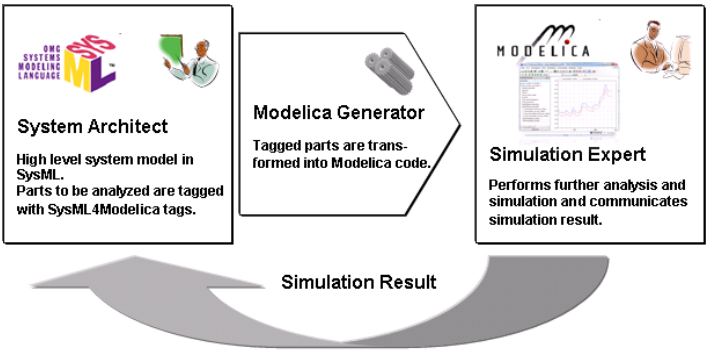


Fig. 2. Exemplary system engineering workflow with feedback in the use case eCar

In the use case eCar, a system architect performs the following steps (see Fig. 2):

1. develop a high level architecture in SysML,
2. add details to the components,
3. select a part of the system to be analyzed with additional tools by marking these elements with specific stereotypes (SysML4Modelica),
4. (automatically) transform this part of the system to the target language and pass the resulting model to the simulation experts for the further analysis.

The simulation experts then delivers the results of the analysis back to the system architect, who updates the system with the parameters from the simulation results.

2.2 Use Case Artis Floorstand

The use case Artis Floorstand refers to a particular configuration from the Artis product family of C-arm systems produced by Siemens Healthcare which are most notably used in angiography as carriers for imaging devices. A characteristic feature of these systems is that the mechanical structure is mainly fixed and further development often concentrates on improving the performance or studying other non-functional aspects. Typical challenges arising in the development are e.g. reaching higher rotation speed (e.g. from 20°/s to 45°/s) to allow a faster imaging process and reduce patient exposure to radiation, or better positioning accuracy (e.g. a maximum tolerance of ±2° for rotational movements instead of ±5°) for higher imaging quality. To this end, the influence of certain design decisions has to be analyzed as early as possible. For instance, how big are the resulting torques when accelerating the C-arm to higher rotation speeds? Can these torques be generated by the electric motors used so far? What is the benefit of more costly, yet more precise gears? How does smaller clearance or

higher stiffness in the gears affect positioning accuracy? During the early development stages all architectural decisions need to consider such non-functional aspects and therefore a tradeoff analysis has to be made, with multi-body simulations being most suitable for a first quantitative analysis.

Traditionally, parts of this development process were based on textual system specifications. However, with increasing system complexity, this approach was found to be unsatisfactory. Thus, the main aim of our work was to show the benefit of SysML models against pure textual specifications and evaluate how these SysML models can be used for early system simulations to support design decisions with a high-level tradeoff analysis.

For multi-body simulation aspects Modelica was used but as the evaluation of different simulation tools and languages was still ongoing it was decided that the SysML4Modelica profile will not be applied. Instead, we tried to include information needed for the simulation purposes in the SysML model in a way that allows a simple export into the simulation tool of choice. Most importantly however, we found that information being specific to any particular physical or mathematical model or simulation environment would clutter the SysML model with information that is of no use for the system architect. Also, the system architect cannot be expected to be an expert in physical or mathematical modeling.

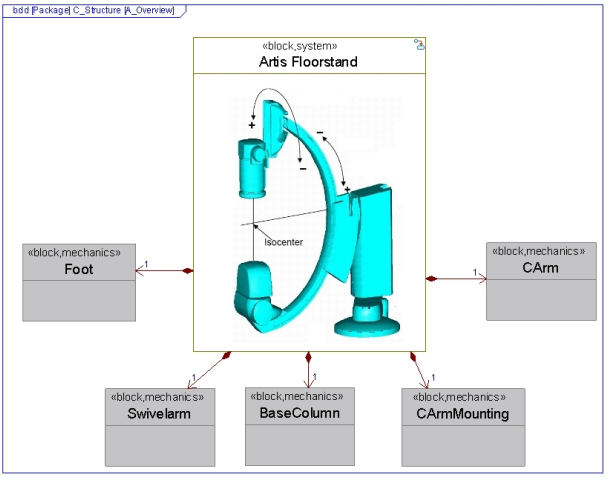


Fig. 3. High-level SysML representation for Artis Floorstand

Development Process

The selected development process for this use case contains the phases: requirements, identification of key hazards, description of the system functionalities and structure, and finally system simulation to verify the requirements. As part of the system structure definition, different views (mechanical, electrical, etc.) are created.

For this use case with a higher level of complexity in the system specifications, it appears more suitable to transfer only a “skeleton” of a simulation model to the

simulation expert out of the system model. This skeleton should reflect the system's physical structure, and be connected to the essential design parameters stored in the SysML model. The simulation expert extends the skeleton to a complete simulation model, exploiting his expertise in physical and mathematical modeling, to cope with the simulation requirements.

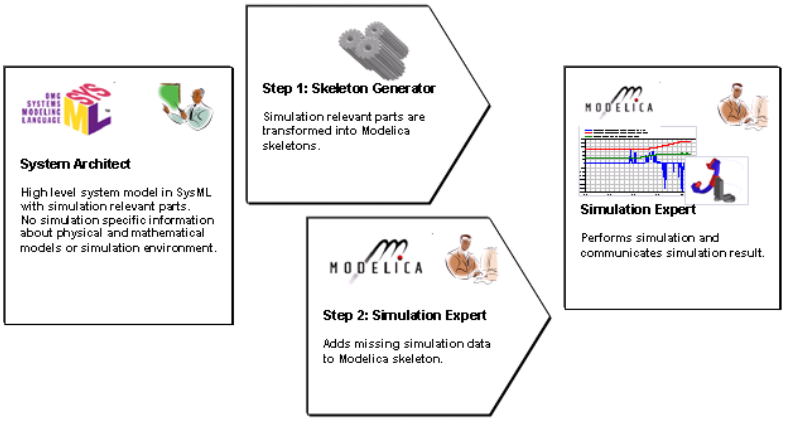


Fig. 4. Exemplary workflow "From SysML to simulation" in the use case of Artis Floorstand

Exemplary Workflow

A system architect of this use case performs the following steps (shown in Fig. 4):

1. develop the high-level architecture in SysML,
2. add simulation relevant data and mark them using dedicated stereotypes,
3. (automatically) create a Modelica skeleton model for the respective SysML components, as well as their connections.

This relieves the simulation expert from the burden of getting the simulation relevant information from a text based document or by interviewing the system architect. The simulation expert then proceeds as follows:

1. open the Modelica skeleton model (he got from the system architect),
2. add missing data to the skeleton, e.g. by editing the connect-equations and connectors,
3. perform the simulation with the essential design parameters being taken from the SysML model,
4. communicate the simulation results and findings to the system architect.

Given the completed simulation model, the system architect then has the possibility to study the system performance under variations of the essential design parameters (trade-off simulations) over which the Modelica model remains coupled to the SysML model. In particular, this can be done autonomously by the simulation expert.

3 Implementation Details

This section presents details how the development processes are realized in the different use cases. Aspects of system modeling over different development cycles are regarded in more details. Benefits from reuse and refinement of models of different abstraction levels are discussed.

3.1 Use Case eCar

In the eCar use case, during the first stage of system development, a low-fidelity model of the system interfaces is constructed. In this stage, the models focus on the analysis of basic information and energy flow. As interfaces are defined in SysML, system simulations are continuously used to evaluate if the interfaces contain all relevant information and reflect the natural technical variables used by the domain experts to design the components. As the design process continues, individual component models are iteratively enriched by more details. In the late stages of a multi-domain development process, the system simulation is used for testing of hardware components. SysML information is used as a central hub for keeping the various component and test revisions synchronized with the system simulation. In this example, physical design refers to mechanical and electronic components modeled with a Modelica tool. Software design addresses software architecture developed with a UML tool.

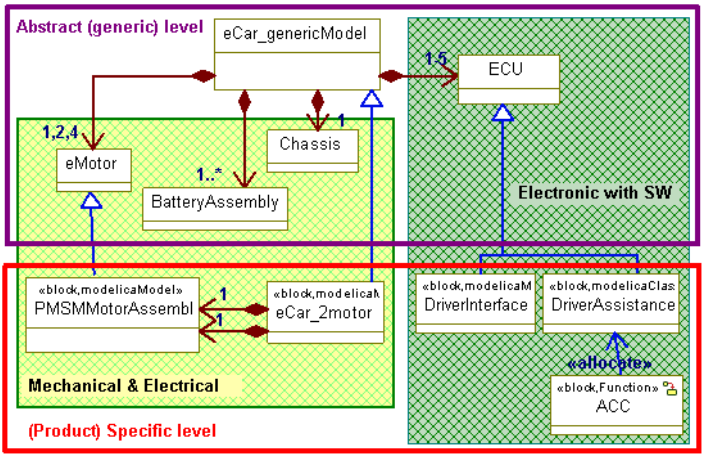


Fig. 5. Introducing a new function (ACC) into existing architecture

Fig. 5 shows an abstract representation of a part of the logical architecture with a sample software component for the ACC function intended to be mapped to one of the electronic control units (ECUs). It also shows some examples for variants of physical constituents of an electrical car and their ECUs as well as variants of software algorithms and drive cycles (inputs for ACC sensors) specified via the generalization relation. To configure specific products, one of the variants for each constituent is

selected and connections between these specialized blocks are defined within the SysML internal block diagrams. A variant PMSMMotorAssembly of an electromotor, BatteryAssembly and Chassis from the physical design, as well as product configurations for 1-motor and 2-motor and all their elements (blocks, connectors etc.) are marked with SysML4Modelica stereotypes. These elements are translated automatically into Modelica models. The ECU with ACC software is connected to "physical parts" as an instance of the abstract interface block (ACC) containing the reference to the output of the functionality evaluation for a specific drive cycle. Fig. 1 (left-hand) shows the simulation models generated out of the product configurations in SysML.

3.2 Use Case Artis Floorstand

A major insight of the use case Artis Floorstand is that different levels of abstraction of simulation models should be kept well separated from the system description. If the system description is refined, also the simulation models have to be refined accordingly. In the approach of this use case, the reuse of physical and mathematical models depends on the nature of the refinement. If the system architect refines the SysML model in the sense that block diagrams are added or deleted, reuse of the physical and mathematical models added to the Modelica skeleton by the simulation expert is ensured through the use of preserved areas in the generated Modelica code skeleton.

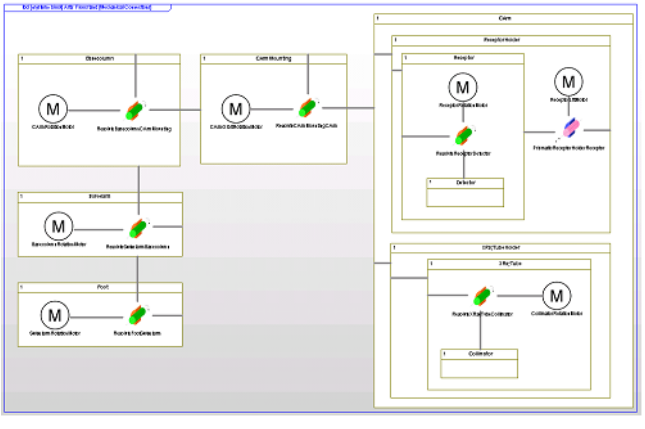


Fig. 6. SysML Internal Block diagram showing mechanical connections

Moreover, physical and mathematical models added by the simulation expert can be reused for subsequent studies if stored appropriately (e.g. in libraries). Otherwise, the reuse of simulation models has to be judged by the simulation expert. Especially in the case when the structure of the SysML model is changed (e.g., a block is divided into several parts), it is not clear whether the corresponding physical and mathematical models are still compatible with the new system structure. A major advantage of the strict separation of system description and physical and mathematical modeling is that the simulation expert has all the freedom to exploit his expertise, while consistency with the system description is ensured by automatically generated

Modelica skeletons. The same goes for the essential design parameters that are included in the system description. However, the simulation expert has to use these parameters manually, which is acceptable as long as the number of such parameters is small.

The following elements store the simulation relevant data within the SysML model:

- Internal block diagrams were used to model specific views of the system like the mechanical view (shown in Fig. 6).
- Block properties not only for models of the system parts but also as specific mechanical joints (e.g. revolute or prismatic). Also dedicated stereotypes and icons were used to make such elements distinguishable on diagrams.
- Mechanical connections modeled via associations marked with a dedicated stereotype «mechanics» which also included specific color setting to make them easily visible on diagrams.
- Attributes for block properties added to store additional information (e.g. weight, size) for system parts.

All these elements and diagrams are of interest to the system architect but are also relevant for simulation purposes; they are used to generate Modelica skeletons but could also be easily used as input for any other simulation language.

4 Comparison of the Proposed Approaches

In this section, advantages and shortcomings of the proposed approaches as well as challenges regarding their suitability in industrial settings are discussed. The following benefits are found to be common for the both approaches.

- SysML models allow for consistent structuring of complex technical systems and an easy-to-understand visualization of the structure and behavior.
- Major improvements are observed in the collaboration between the requirements engineering and the system engineering departments. This stems from the fact that requirements may be connected more easily to the components or subsystems.
- The generation of Modelica code (skeletons) out of a SysML model is another major improvement since the simulation expert is provided with a consistent structure of the system and no longer has to infer the structure himself.

This makes the collaboration between developers of different phases formal and with less or no communication errors.

The differences of the approaches are described in the following subsection and summarized in Table 1.

4.1 Advantages and Disadvantages of both Approaches

The approach used in the use case eCar is seen to have advantages concerning information distribution and collocation thanks to a SysML-based overall system description structured into development phases, levels of abstraction/details, and considered

aspects/analysis. Especially, the possibility of early evaluation of the system design through the combination of SysML animation and Modelica simulation in a frontloading approach saves development costs and time. However the synchronization between SysML and Modelica seems to be very challenging, because the current version of the SysML4Modelica profile allows transforming the complete Modelica models into SysML. This level of detail in the SysML4Modelica profile is not relevant for the high-level system architecture. It cannot be assumed that a system architect has the detailed knowledge of the semantics (and usage) of Modelica specific elements, nor of physical or mathematical modeling. In the case when the SysML model is intended to play the role of a container of the complete information, this will make the system description too complex and difficult to manage.

The reasonable usage of the standardized transformation SysML4Modelica is that each system architect is free to choose the level of details for the intended transformation, which he wants and is able to capture in the system description. It is not required that he uses the complete SysML4Modelica profile. Similarly, in this way the transformation is not restricted to a limited set of predefined interfaces, but can be refined at any time of the development.

An essential drawback of this approach is that the synchronization interfaces between SysML and Modelica, while defined with the help of stereotypes in SysML, are not distinguishable within Modelica anymore. This makes the automation of the feedback step (when results of the simulation are fed into the system description) challenging and still not standardized. The “feedback”-interfaces must be distinguishable within Modelica models to avoid that the whole Modelica model is mapped to the SysML model. Another drawback of this approach can be the limitations for the simulation experts when the generated simulation models need to be modified while keeping them compliant with the original model in SysML. Moreover this standardized profile can only be used only with Modelica-based simulation environment.

To showcase the approach of Artis Floorstand, a prototype model transformation script was implemented that allowed generation of Modelica skeletons based on specifically stereotyped SysML elements. This was justified by the request that the system descriptions shall not be contaminated with simulation-specific data, owing to the fact that the roles of system architect and simulation expert are normally assigned to two different people or departments. A big advantage of the approach is therefore that the persons working on the architecture or the simulation only need to know their own language (e.g. SysML or Modelica) and respective tools but not both. The simulation expert retains the full flexibility in setting up physical and mathematical models for the system, both of different levels of abstraction and different levels of detail, while the SysML model is not overloaded with simulation-specific data. In addition, consistency between the structure of the SysML-based system description and the simulation models is ensured through the automatic generation of Modelica code skeletons. Another collaborative benefit is that a system architect can reuse the simulation models established by the simulation expert independently from the expert’s support for trade-off analyses.

One drawback of the concept is the effort needed to identify the necessary information that is relevant to connect it to the simulation (most notably Modelica connector

classes). This task would remain highly project specific, but also strongly tied to the simulation that should be performed. The concept may therefore only be feasible for projects or problem classes where the simulation goal is clearly identified as both the SysML model as well as the transformation rules needs to be tailored to fit this goal.

Table 1 summarizes the comparison of the considered approaches.

Table 1. Assessment of comparison aspects for the two approaches

	Approaches used in	
Aspects for comparison	*eCar*	*Artis Floorstand*
Complexity within SysML representation	probably high	always low
Completeness of generated simulation models	complete is possible	only code frames
Flexibility of changing simulation models	low	high
Selection of the interfaces to simulation	flexible	fixed
Flexible selection of simulation tools	fixed language	possible
Synchronization efforts between formalisms	high	low
Usability	may be complex	easy
Area of application	universal	project specific

4.2 Challenges in Industrial Practice

The both procedures for the use cases described above carries specific challenges especially in an industrial setting:

- It has to be decided which information should be modeled within the system description language and the simulation model. A system architect must be able to identify the goal of the simulation at different development phases and specify simulation relevant attributes in a non intrusive way. Thus, a trade-off between the following aspects is the most challenging:
  - A system architect should not be challenged with loads of simulation specific elements within his SysML model.
  - The SysML model needs to contain enough data that allows the generation of a meaningful simulation model.
- As most multi-domain systems are designed in increasingly large teams, interfaces are often set up at early design stages, and later can be modified corresponding to new requirements or other changes in the environment. This implies rework on the existing models. On the other hand the cooperation between different stakeholders must be as easy as possible to provide a fast reply to the external changes.
- The multi-domain design community is more and more following iterative design processes with a need of iteration results (e.g. from the component level) to be fed back to the system description. Therefore, the model transformation tools need to have two-way capabilities also handling conflicts during synchronization.

5 Conclusion and Outlook

In this paper two multi-domain systems engineering approaches, integrating model-based system description and simulation, have been compared. Each of the two approaches shows advantages that justify its application in the specific project setting but also reveals some weaknesses that have to be minimized. As a result of this case study we see the following focus for the future work. The challenges of synchronization between SysML and Modelica need to be explored in more detail. The possibility for the automatic updates of the system models with the results of the simulation should be investigated. And we plan to evaluate the usage of the SysML4Modelica profile in other use cases with the aim to identify the extent to which simulation details can be effectively managed within the SysML model.

References

1. International Council on Systems Engineering (INCOSE): Systems Engineering Vision 2020, Document No. INCOSE-TP-2004-004-02, Version 2.03 (2007), http://www.incose.org/ProductsPubs/pdf/SEVision2020_20071003_v2_03.pdf
2. Object Management Group: OMG Systems Modeling Language (OMG SysML), V.1.2, OMG formal specification formal/2010-06-02 (June 2010), http://www.sysml.org/docs/specs/OMGSysML-v1.2-10-06-02.pdf
3. Object Management Group: Unified Modeling Language: Superstructure, V.2.3, formal/2010-05-05 (2010), http://www.omg.org/spec/UML/2.3/
4. Fritzson, P.A.: Principles of object-oriented modeling and simulation with Modelica 2.1. John Wiley & Sons, Inc. (2004)
5. Schamai, W., Fritzson, P.: Paredis, C., Pop, P.: Towards Unified System Modeling and Simulation with ModelicaML: Modeling of Executable Behavior Using Graphical Notations. In: Proc. of the 7th Modelica Conference, Como, Italy, September 20-22 (2009)
6. Paredis, C., et al.: An Overview of the SysML-Modelica Transformation Specification. In: Proc. of the 20th Anniversary INCOSE Int. Symp., Chicago, IL, July 12-15 (2010)
7. Thramboulidis, K.: Model Integrated Mechatronics – Towards a new paradigm in the development of manufacturing systems. IEEE Transactions on Industrial Informatics 1(1) (2005)
8. Enge-Rosenblatt, O., et al.: Functional Digital Mock-Up and the Functional Mock-up Interface – Two Complementary Approaches for a Comprehensive Investigation of Heterogeneous Systems. In: Proc. of the 8th Int. Modelica Conference (2011)
9. Votintseva, A., Witschel, P., Goedecke, A.: Analysis of a Complex System for Electrical Mobility Using a Model-Based Engineering Approach Focusing on Simulation. Procedia Computer Science 6, 57–62 (2011)

Strengthening SAT-Based Validation of UML/OCL Models by Representing Collections as Relations

Mirco Kuhlmann and Martin Gogolla

University of Bremen, Computer Science Department
Database Systems Group, D-28334 Bremen
{mk,gogolla}@informatik.uni-bremen.de

Abstract. Collections, i. e., sets, bags, ordered sets and sequences, play a central role in UML and OCL models. Essential OCL operations like role navigation, object selection by stating properties and the first order logic universal and existential quantifiers base upon or result in collections. In this paper, we show a uniform representation of flat and nested, but typed OCL collections as well as strings in form of flat, untyped relations, i. e., sets of tuples, respecting the OCL particularities for nesting, undefinedness and emptiness. Transforming collections and strings into relations is particularly needed in the context of automatic model validation on the basis of a UML and OCL model transformation into relational logic.

1 Introduction

Models are a central means to master complex systems. Thus, for developing systems, building precise models is a main concern. Naturally, the examination of the validity of complex systems must be supported via tracing and checking model properties.

We employ the Unified Modeling Language (UML) and its accompanying textual constraint and query language OCL (Object Constraint Language) for the description of models. For automatically analyzing and validating models, we utilize *relational logic*. Relational logic is efficiently implemented in Alloy [11] and its interface Kodkod [19] which transforms relational models into boolean satisfiability (SAT) problems. As a consequence, our task consists in transforming our source languages UML and OCL as well as the considered model properties into structures and formulas of the target language relational logic. This way, we enable SAT-based validation of UML/OCL models. We have started to implement the transformation from UML/OCL to relational logic in a so-called model validator [13] which has been integrated into our UML-based Specification Environment (USE) [8].

In this paper, we focus on a vital aspect of UML/OCL models, namely the handling of OCL collection kinds (set, bag, ordered set, and sequence)[1] and

[1] One collection kind (e. g., set) can be manifested in different concrete collection types (e. g., Set(Integer) and Set(Bag(String))).

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 32–48, 2012.

strings. OCL collections and collection operations play a central role in the language. They are crucial for building precise UML/OCL models which can be successfully analyzed and checked. For instance, the evaluation of existentially and universally quantified formulas is based upon collections of values like in `Person.allInstances->exists(p|p.age<18)`. Another naturally used operation is role navigation which results in collection values, e. g., when the allowed states of a structural model given in form of a class diagram have to be restricted and the restriction involves two classes and an association navigation path between the classes, the association path will be evaluated in OCL through a collection expression. The following example ensures a minimum salary by collecting the employees of all companies using navigation: `Company.allInstances.employee->forAll(e| e.salary>3000)`.

The example model shown as a UML class diagram in Fig. 1 emphasizes the use of strings and different types of collections. A university is located in a specific state encoded by a two character string, e. g., 'DK' or 'US'. A person may have several postal and e-mail addresses and may be enrolled in a university. While postal addresses are always unordered (Set(String)), the e-mail addresses of a person can be prioritized by using an ordered set of addresses, or be retained without any prioritization by using a set. The abstract type Collection(String) allows for determining the concrete type (OrderedSet(String) or Set(String)) at runtime.

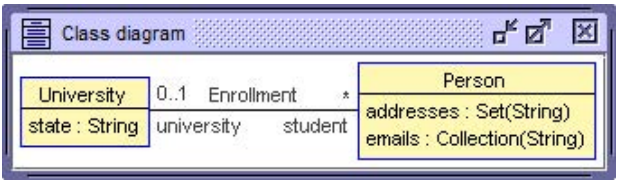


Fig. 1. Example UML Class Diagram with String and Collection Type Attributes

The following example OCL invariant further constrains the model. It requires each person who is a student at a university to have a postal address in the same state the university is located. For this purpose, it checks whether the string representing the university's location also occurs at any position in at least one postal address string of the student.

```
context Person inv AccessibleStudents:
  self.university.isDefined implies
    self.addresses->exists(a|
      Set{1..a.size}->exists(i|a.substring(i,i+1)=self.university.state))
```

However, when comparing UML and OCL, our source languages for describing models, and relational logic, our direct target language utilized for automatic model validation, we observe an impedance mismatch: (a) OCL offers four collection kinds whereas relational logic and its implementation Alloy directly support only relations, i. e., sets of flat tuples; *"other structures (such as lists and*

sequences) are not built into Alloy the way sets and relations are" (see p. 158 in [11]) (b) OCL is a typed language whereas plain relational logic is untyped. This means that the OCL type system has to be represented in relational logic and the missing collection kinds have to be encoded as sets. (c) The lack of *"higher-order relations"* implies that *"collections of collections"* which often occur in UML/OCL models are not directly supported in Alloy [1]. Consequently, the challenge is to respect all involved OCL particularities in the translation which means that nested collections as well as OCL type rules, the undefined value, and empty collections deserve special attention.

We present a uniform transformation which respects all language inherent differences between UML/OCL collections and flat relations of relational logic. Furthermore, we enable the representation of structured string values in relational logic. A comprehensive representation of UML/OCL collections and strings in relational logic is the premise for the translation of collection and string operations and, hence, a comprehensive approach to automatic UML/OCL model validation utilizing Kodkod and SAT solving. However, there is currently no other SAT-based approach which supports models like the example depicted in Fig. 1 or the related OCL constraint.

The rest of this paper is structured as follows. Section 2 associates the content of this paper with the SAT-based model validation context. The central Sect. 3 will show how OCL collections and strings are represented as relations. First, we introduce the approach considering exemplary transformations. Then, we illustrate the underlying transformation algorithms. After a discussion of performance implications in Sect. 4 and related work Sect. 5, we conclude with Sect. 6.

2 Model Validation via SAT Solving: Context

Our validation approach bases upon checking model properties by inspecting the properties of model instances (snapshots), e. g., the existence or non-existence of specific snapshots allows conclusions about the model itself. As shown in Fig. 2, the USE *model validator* allows developers to automatically analyze properties of their UML/OCL models by translating them into relational structures, i. e., bounded relations and relational formulas, which can be handled by the model finder Kodkod. In addition to a model, the properties under consideration, usually given in form of OCL expressions, as well as user-configurations with respect to the search space are transformed and handed over to Kodkod.

Kodkod in turn employs SAT solvers to find a solution, i. e., proper instantiations of specified relations, fulfilling the given formulas. Found SAT instances are therefore translated back into instances of the specified relations. In the end, the model validator transforms the relational instances into instances of the UML/OCL model and visually presents the found solution in form of an object diagram to the developer.

Since UML/OCL collections and strings values play a central role in precisely specified models, corresponding validation approaches must support the four collection kinds and their peculiarities as well as strings in order to provide a

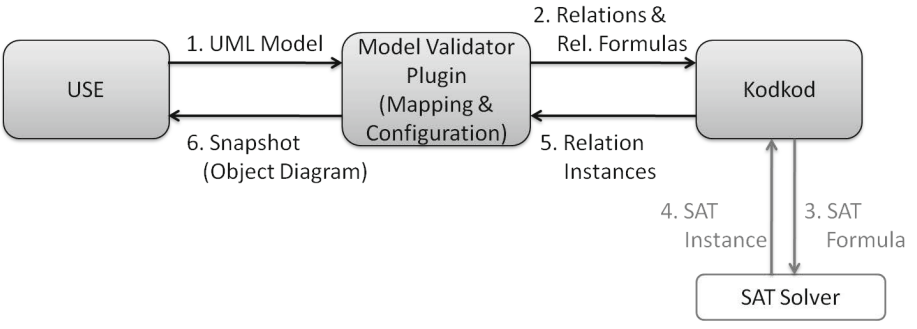


Fig. 2. Transformation process involving the USE model validator

comprehensive validation platform. The transformation algorithm discussed in this paper has been implemented in the model validator enabling the definition of meaningful OCL constraints on the one hand, and user-defined properties which are to be inspected on the other hand.

3 Transforming Collections and Strings into Relations

Relational logic describes formulas whose evaluation is based on flat relations with different arities, i. e., sets of tuples with atomic components, since relational logic forbids nested relations. Beside boolean and integer operations, relational logic naturally supports set operations like union and set comprehension. A central operation is the relational join for accessing specific components (i. e., columns) of tuples and for connecting tuples of different relations.

Relations generally have the same properties as OCL sets which are unordered and do not allow duplicate elements.[2] Thus, there is a straightforward way to translating non-nested sets into unary relations, e. g., `Set{2,1,3}` can be represented by the relation `[[3],[1],[2]]`. On the other hand, the following characteristics of OCL collections must be respected:

- Bags and sequences require the support of duplicate elements.
- Ordered sets and sequences require the support of ordered elements.
- All collection kinds require the support of nested collections.

A universally applicable transformation must cover all of these properties.

3.1 The Basic Idea

In this subsection we consider the first two named properties (support of duplicate and ordered elements), nested collections, and strings as well as the handling of undefined and empty values in greater detail. The comparability of collection and string values must be preserved by their relational representation. This essential aspect is discussed at the end of this subsection.

[2] Henceforth, the term 'set' refers to an OCL set.

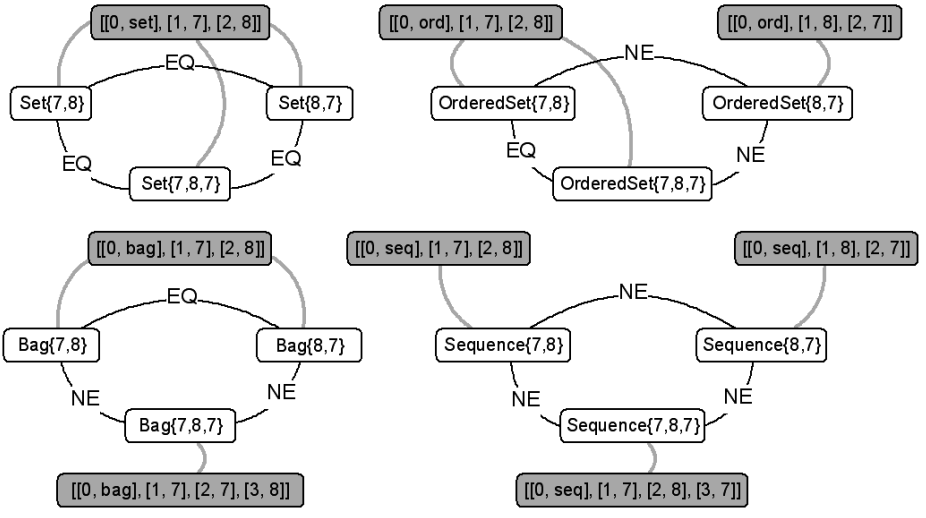


Fig. 3. Distinction of OCL collections and the corresponding translation into relations

Handling of Flat Collections. In Fig. 3 we illustrate the properties of the four OCL collection kinds based on three concrete literals, in each case. The OCL literals – depicted in white boxes – involve duplicate elements and elements in a particular order. Two literals are equal (EQ), i. e., they represent the same value, or are not equal (NE). For instance, while `Bag{7,8}` equals `Bag{8,7}`, the collection value `OrderedSet{7,8}` does not equal `OrderedSet{8,7}`.

Collection literals describing the same value should naturally yield the same (identifying) relational representation which are shown in grey boxes and by grey connecting lines. The translation assigns an index $1 \leq i \leq n$ to each element of a collection with n elements, determining an explicit element position. See, for example, the relational representation of the value `Sequence{7,8,7}`. The corresponding relational representation relates the index 1 to the first element (`7`), index 2 to the second element (`8`) and index 3 to the last element (`7`).

The depicted, grey relations reveal four distinctive features of the transformation which do not directly result from the collection properties, but from explicit design decisions:

- The elements of a set or bag are indexed in the respective relational representation, although sets and bags are intrinsically unordered and sets do not include duplicate elements.
- m duplicates of element e in a bag occur m times in the respective relational representation.[3]
- The elements of sets and bags are sorted in the resulting relations based on the natural order of integer values. For example, considering the literals `Set{7,8}` and `Set{8,7}`, the integer value `7` always precedes the value `8` in the relational representation of both sets.

[3] An alternative would be tuples counting element occurrences (e. g., $[e,m]$).

– The special index 0 indicates a typing tuple that determines the collection kind a relation represents (set, bag, ord, seq).

The first two features (an order for sets resp. bags and retention of explicit duplicates) directly follow from our intention to define a uniform transformation with no exceptional cases and resulting case distinctions, clearly simplifying (a) the representation of nested collections, and (b) the translation of OCL collection operations into relational logic. The last two aspects (sorting elements and explicit typing) allow us to compare OCL collection values in relational logic through an explicit sorting of their elements, as we will discuss at the end of this subsection.

Handling of Nested Collections and Strings. In the case of nested collections, the elements of a collection are in turn collections. In order to encapsulate the individual collections, i. e., to determine which value belongs to which collection, there must be an additional indicator. Following our uniform translation, a natural way to representing collection type elements is the use of a new index column, as shown in the following example:

```
Set{7}                --> [[  0,set  ],   the relation represents a set
                           [  1,7    ]]  the first element is 7
Set{8,9}              --> [[  0,set  ],   the relation represents a set
                           [  1,8    ],  the first element is 8
                           [  2,9    ]]  the second element is 9
Sequence{Set{7},
         Set{8,9}}    --> [[0,seq,seq],   the relation represents a sequence[4]
                           [1,0,set  ],   the first element is a set
                           [1,1,7    ],   the first element of the set is 7
                           [2,0,set  ],   the second element is a set
                           [2,1,8    ],   the first element of the set is 8
                           [2,2,9    ]]   the second element of the set is 9
```

The support of OCL collections also allows for representing string values. While strings may be seen as atomic values (e. g., `'Ada' --> [[Ada]]`), it is often necessary to consider a string as a value with an inner structure. Thus, because there is a need in OCL for manipulating and querying strings, they are treated like sequences of characters and are identified by a respective string typing tuple (e. g., `'Ada' --> [[0,str],[1,A],[2,d],[3,a]]`). A set of strings can thus be seen as sequences of values nested in a set:

```
Set{'Ada','Bob'} --> [[0,set,set],
                      [1,0,str], [1,1,A], [1,2,d], [1,3,a],
                      [2,0,str], [2,1,B], [2,2,o], [2,3,b]]
```

[4] Since all tuples of a relation must have the same arity, we use the collection kind indicator (e. g., seq) to *fill* typing tuples until they yield the required number of components. Multiple indicators in one typing tuple, thus, have no special meaning.

In OCL, sets, bags, sequences and ordered sets are specializations of *Collection*, and all basic types are subtypes of *OclAny*. Thus, for example, we can create collections including elements of type Collection(OclAny):

```
Set{Sequence{5,6,5},Set{'Ada',7,'Bob',8}} =
Set{Set{7,8,'Ada','Bob'},Sequence{5,6,5}} -->
[[0,set,set,set],
 [1,0,set,set],
 [1,1,1,7],
 [1,2,1,8],
 [1,3,0,str],[1,3,1,A],[1,3,2,d],[1,3,3,a],
 [1,4,0,str],[1,4,1,B],[1,4,2,o],[1,4,3,b],
 [2,0,seq,seq],
 [2,1,1,5],
 [2,2,1,6],
 [2,3,1,5]]
```

If string and non-string basic types are mixed, the non-string basic type values are brought into the complex string representation by handling them as if they were strings of length one with an absent typing tuple, e. g., the integer value `7` is represented as `[1,7]` instead of `[7]`.

The translation result of the previous example is a relation that represents a set including collections of sequences. Each additional nesting level adds a further index column to the relation. The fourth column determines the character or integer value. The third column determines the position of the characters in a string. The second column determines the position of a string within a collection. The first column determines the position of the collection in the outer set.

For instance, the tuple `[1,3,2,d]` determines `'d'` to be the second character of the third element (`'Ada'`) in the first element (`Set{7,8,'Ada','Bob'}`) of `Set{Set{7,8,'Ada','Bob'},Sequence{5,6,5}}`.

Undefined and Empty Collections. Empty collections are naturally represented by the absence of further tuples besides the typing tuple. Undefined (`un`) collections on the other hand yield a characteristic relational representation. This representation allows us to identify at which nesting level an undefined value occurs (c. f. the three different levels in the following example). Furthermore, undefined values are not accompanied by typing tuples, since the information which concrete type an undefined value represents is irrelevant.

```
Set{Undefined, Set{}, Set{Undefined, Set{}, Set{Undefined}}} -->
[[0,set,set,set],    the relation represents a set
 [1,un,un,un],       the first element is an undefined collection
 [2,0,set,set],      the second element is an empty set
 [3,0,set,set],      the third element is a set
 [3,1,un,un],        its first element is an undefined collection
 [3,2,0,set],        its second element is an empty set
 [3,3,0,set],        its third element is a set
 [3,3,1,un]]         which includes an undefined value
```

Making Ordered Relations Comparable. In Fig. 3 we depicted the equality and inequality of specific collection literals. Since sets and bags are intrinsically unordered, we obtain the properties: `Set{7,8}=Set{8,7}` and `Bag{7,8}=Bag{8,7}`. Accordingly, an equality check regarding the relational representation of both sets and both bags, respectively, must evaluate to true. We can achieve a general valid comparability at the relational level (a) by sorting the elements of the relational representations of sets and bags on demand (e. g., in the case of an equality check, or the casting operation `Set::asSequence()`), or (b) by sorting the elements already during the creation process. We applied the latter strategy which results in a unique representation of equal collection values through direct sorting:

`Set{7,8} --> [[0,set],[1,7],[2,8]] <-- Set{8,7}` and
`Bag{7,8} --> [[0,bag],[1,7],[2,8]] <-- Bag{8,7}`

SAT-based validation implies bounded search spaces, i. e., at the UML level, a limited set of covered model instances. Hence, the set of participating values (boolean, integer, enumeration, character, or object type)[5] is finite. This allows us to create a total order on all available values and to define a sorting algorithm with respect to the precedence of these values. Within the previous example of nested collections with mixed basic type values, the literals `Set{7,8,'Ada','Bob'}` and `Set{'Ada',7,'Bob',8}` yield the same relational representation after sorting the elements (integer precedes string). The example shows three further properties of the sorting algorithm:

- The sorting of sets and bags has a recursive nature, i. e., sorting is applied at each nesting level. Before an outer set or bag can be sorted, its elements must have been sorted.
- Sequences, ordered sets and strings are never sorted, since the order of their elements (or characters, respectively) is significant (e. g., `Sequence{5,6,5}<> Sequence{5,5,6}`). If they, however, include set or bag valued elements, these sets and bags have to be sorted (e. g., `Sequence{Set{8,7},Set{2,1}}= Sequence{Set{7,8},Set{1,2}}`).
- Beside basic values, also strings and collections obtain an explicit precedence, based on the collection kind, number of elements (or characters, respectively), and precedence of their elements (or characters), e. g., `'Bo' < 'Ada'`, `'Ada' < 'Bob'`, `Set{7} < Bag{7}`, `Set{7} < Set{1,2}`, and `Set{1,2} < Set{7,8}`.

The need for typing tuples directly follows from the need for comparability. Since the values `Bag{7,8} --> [[0,bag],[1,7],[2,8]]` and `Set{7,8} --> [[0,set],[1,7],[2,8]]` are not equal, a relational representation without typing tuples would lead to an invalid conclusion for equality:

`Set{7,8} --> [[1,7],[2,8]] <-- Bag{7,8}` $\lightning$

[5] The character type includes the alphabetic characters which are needed to create string values. The basic predefined type Real is currently not supported.

3.2 Realization of the Transformation Algorithms

In this section, we explain the details of translating UML/OCL collections into relations by considering the relevant transformation algorithms. Since the algorithm for constructing strings at the relational level is a special case of the creation of flat sequences, we focus on the general handling of collections.

First, we consider the core algorithm describing the creation of collection values. Then, we go into details of sorting collections which is needed in the context of sets and bags.

The Collection Creation Algorithm. The algorithm for creating UML/OCL collections in their relational representation includes two main aspects: (a) Each given element which should be included into the collection and is already available in a relational representation is incrementally indexed and added to the resulting relation. Duplicate elements are discarded if the resulting relation should represent a set or ordered set. (b) Relations representing sets or bags are sorted in the end. In this case, the following sorting algorithms become relevant. For details see Algorithm 1 in the appendix.

The Collection Sorting Algorithms. The central sorting algorithm includes the main activities for sorting relations representing sets or bags. It takes all possible pairs of elements existing in the given relation and determines which element precedes the other. The number of predecessors an element possesses then determines its new position in the sorted relation. The element without predecessors obtains the first position (index 1), and an element with x predecessors becomes the $x + 1$th element in the sorted relation. For details see Algorithm 2 in the appendix.

The precedence of two complex, collection-valued elements is determined by a further algorithm which respects the following precedence rules: undefined collections precede sets, sets precede sequences, sequences precede bags, bags precede ordered sets; in the case of collections of the same kind, the number of elements within these collections becomes relevant; if the numbers are identical, the precedence of the elements within the two considered collections must be recursively determined. For details see Algorithm 3 in the appendix.

The recursive calculation of element precedences may end at different levels of nested values. For example, consider the following pairs of collections:

A: `Set{Set{Set{7}}}`, `Bag{Set{Set{7}}}`
B: `Set{Set{Set{7}}}`, `Set{Set{Bag{7}}}`
C: `Set{Set{Set{7}}}`, `Set{Set{Set{8}}}`

In each case the left side precedes the right side. While in pair A the precedence can be directly determined (sets precede bags), pair B demands a nested (recursive) comparison until the elements `Set{7}` and `Bag{7}` at the second nesting level are reached. In the case of pair C, the final level of recursion is reached, i. e., the level at which only simple values occur (`7` and `8`).

As mentioned before, each validation task given to the USE model validator describes a *finite* user-defined *universe* of simple values (i. e., boolean, integer, enumeration, character, or object type). As a consequence, the precedence of these simple values can always be specified via a total order relation.[6] Given such a total order relation and two simple values, the precedence of both values can directly be calculated. For details see Algorithm 4 in the appendix.

4 Discussion

A bounded search space of the model validator (resp. Kodkod) requires bounded relations and thus bounded collection representations. Kodkod considers the set of all available (user-defined) simple values as a universe of atoms. Relations are bounded to a set of possible tuples by determining a set of possible atoms (a domain) for each column of the relation tuples. For instance, a relation that represents the type Set(Set(Boolean)) yields tuples of the form:

[$index_1$, $index_2$, $value$], with
$index_1 \in Domain_1 = \{0, 1, ..., x, \texttt{un}\}$, where x is a user defined maximum number,
$index_2 \in Domain_2 = \{0, 1, ..., x, \texttt{un}, \texttt{set}\}$, and
$value \in Domain_3 = \{\texttt{true}, \texttt{false}, \texttt{un}, \texttt{set}\}$.

There are $|Domain_1| * |Domain_2| * |Domain_3|$ possible tuples which can be included by an instance of the considered relation. As we have explained before, each nesting level of collections adds one additional column to the respective relation, increasing its arity by one. A nesting depth of n implies a relation of arity $n+2$ (or $n+3$ if strings are involved). Consequently, each additional nesting level considerably increases the search space and correspondingly reduces the SAT solving performance. Furthermore, Kodkod limits the maximum arity of involved relations und thus the maximum nesting depth: $|universe|^{max\_arity} < 2^{31} - 1$. Future work will comprise the optimization of the search space by bounding the possible tuples to OCL collection specific patterns.

There is also potential for optimization with respect our representation of OCL collections as flat relations. However, our aim is to present a universally defined and applicable approach in this paper. A concrete implementation can naturally realize several optimizations like discarding typing tuples, if the collection types can be statically determined, i. e., *Collection* is not involved, or using a simple representation of OCL sets and strings in form of unary relations, if only non-nested collections and no complex string values are needed. That is, while our approach supports all OCL collections structures, it can be thinned out as required.

In order to inspect the performance implications in the context of a complete implementation of our approach, let us consider the class diagram shown in Fig. 4

[6] In the case of values which do no yield a natural order, the model validator explicitly induces one, e. g., the order of objects is determined by the order the corresponding object identifiers are declared within the model validator, independent of the classes they instantiate.

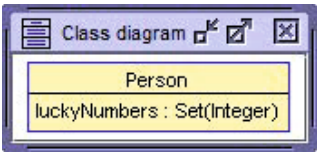


Fig. 4. UML Class with a Set-valued Attribute

which models persons with a set of lucky numbers, as well as the following OCL invariant which demands that people have unique sets of lucky numbers. Please note that in our approach for transforming UML and OCL models into relational models, classes are translated into unary relations, holding atoms which represent object identifiers. Attributes are translated into relations connecting object identifiers with attribute values, plus relational constraints ensuring attribute values of the specified type. In the case of collection-valued attributes, an object is related to each individual tuple of the corresponding collection value. An example instance of the attribute relation `Person_luckyNumbers` is shown at the end of this section.

```
context p:Person
  inv uniqueLuckyNumbersSets:
    Person.allInstances->forAll(p1,p2|
      p1.luckyNumbers=p2.luckyNumbers implies p1=p2)
```

`-->` (sketch of a translation into relational logic)

```
(all p1:Person, p2:Person |
  p1.Person_luckyNumbers=p2.Person_luckyNumbers => p1=p2)
```

First, we use the model validator to automatically translate this UML and OCL model into a relational model, and initiate a search for valid instances in the context of 4, 8, and 12 person objects. Then, we repeat this procedure for nested attribute types. Table 1 reveals the corresponding search times. The second column yields the results for a simple set representation using unary relations, e. g., `Set{7,8} --> [[7],[8]]` instead of the complex representation discussed in this paper, e. g., `Set{7,8} --> [[0,set],[1,7],[2,8]]`.

Table 1. Comparison of SAT Solving Performance regarding different Nesting Levels

#Persons	Set(Int) (simple)	Set(Int)	Set(Set(Int))	Set(Set(Set(Int)))
4	62 ms	437 ms	2200 ms	14955 ms
8	109 ms	764 ms	5132 ms	62540 ms
12	140 ms	1326 ms	16497 ms	140522 ms

In the context of type Set(Set(Integer)) and 4 required person objects, we, for example, obtain the following class and attribute relation instances as a result which are automatically transformed by the model validator into the object diagram shown in Fig. 5.

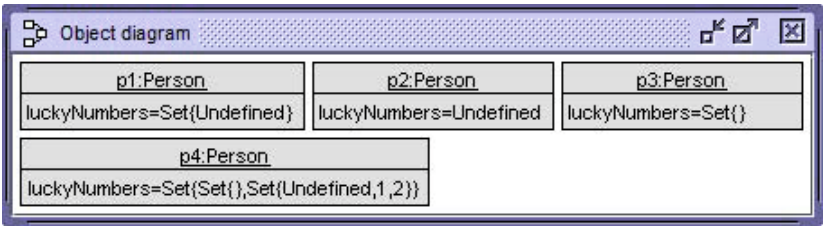


Fig. 5. Solution in the context of 4 Objects and Type Set(Set(Integer))

```
Person=[[p1],[p2],[p3],[p4]]
Person_luckyNumbers=[
  [p1,0,set,set,set],[p1,1,un,un,un],
  [p2,un,un,un,un],
  [p3,0,set,set,set],
  [p4,0,set,set,set],
    [p4,1,0,set,set],
    [p4,2,0,set,set],[p4,2,1,un,un],[p4,2,2,1,1],[p4,2,3,1,2]]
```

5 Related Work

Our paper has connections to many related works. The collection kinds set, bag and list (sequence) are considered in the context of functional programming in [10,23] whereas our approach is designed for object-oriented design and modeling. The Object Query Language OQL [7] uses three (set, bag, list) of the four OCL collections in the same way as they are employed in OCL but without defining a formal semantics. [16] makes a proposal to complete the OCL collections in a lattice-like style leading to union and intersection types. The work concentrates on the OCL collection kind set.

General type and container constructors similar to sets or bags are considered for database design using ER modeling in [9]. [4] represents the OCL standard collections with an extended OCL metamodel allowing for practical tool support with the aim of code generation. [6] studies fundamental properties of OCL collections in order to establish a new generalization hierarchy and focusses of the relationship between sets and ordered sets. [22] proposes a unified description of OCL collection types and OCL basic data types. [14] translates OCL into Maude and represents OCL collections by introducing new algebraic sorts without considering the complete OCL type system. A mapping of non-nested OCL collections and strings into bit-vector logic is done in [17]. In [5] the authors describe a staged encoding of OCL strings that performs reasoning on string equalities and string lengths before fully instantiating the string.

Our approach is based on relational logic which is implemented in the powerful Alloy system described in [11]. Alloy supports non-nested sets and sequences modeled as functions mapping integer (indices) to the sequence elements. The

UML2Alloy approach presented in [1] tackles the translation of UML and OCL concepts into Alloy. The authors sketch the possibility to describe sequences, bags and ordered sets via Alloy structures, but do not discuss further details like the preservation of collection comparability. While the representation of nested collections in Alloy is not possible, because of the lack of higher-order relations and restrictions with respect to available Alloy structures, the Alloy interface Kodkod [19] which we utilize in our approach allows users to handle plain relations with arbitrary contents.

Alloy and Kodkod are used for many purposes. [3] translates conceptual models described in OntoUML for validation purposes into Alloy. In [12] modeling languages and their formal semantics, in [21] enterprise architecture models based on ontologies are specified and analyzed with Alloy. Kodkod has been utilized for executing declarative specifications in case of runtime exceptions in Java programs [15], reasoning about memory models [20], or generating counterexamples for Isabelle/HOL a proof assistant for higher-order logic (Nitpick) [2]. [18] use Kodkod for checking the consistency of models described with basic UML concepts.

6 Conclusion

We have discussed a uniform representation of strings and nested, typed collections in form of flat, untyped sets respecting the OCL particularities for nesting, undefinedness and emptiness. Collections are a central modeling feature in UML and OCL for model inspection and model validation and verification. We have successfully implemented this approach in our model validator and applied it in several middle-sized examples.

As future work, we want to check the approach with larger case studies. In particular, we have to check whether efficiency improvement may be made by factoring out type information from the collection instances. A small benchmark for checking collection and string values could be developed. With respect to OCL, one might propose an OCL simplification based on the experience with the difficult handling of undefinedness in order to shorten the gap between the source and target languages.

Furthermore, the concepts for sorting collections at the relational level which we have discussed in this paper can be reused for standardizing corresponding OCL *sort* operations. Such operations could deterministically lead from unordered collections to sorted collections, e. g.,

`Set{1,2,3}->sort = OrderedSet{1,2,3} = Set{2,3,1}->sort`,
`Bag{1,2,2,3}->sort = Sequence{1,2,2,3} = Bag{2,1,2,3}->sort`, and
`Set{OrderedSet{2,1}, OrderedSet{1,2}}->sort =`
`OrderedSet{OrderedSet{1,2}, OrderedSet{2,1}}`.

However, also ordered collections (which are not necessarily sorted) need sometimes to be sorted, so that we propose using this sort operation for all four collection kinds.

References

1. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from UML to Alloy. SoSyM 9(1), 69–86 (2010)
2. 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)
3. 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)
4. Bräuer, M., Demuth, B.: Model-Level Integration of the OCL Standard Library Using a Pivot Model with Generics Support. ECEASST 9 (2008)
5. Büttner, F., Cabot, J.: Lightweight String Reasoning for OCL. In: Vallecillo, A., et al. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 240–254. Springer, Heidelberg (2012)
6. Büttner, F., Gogolla, M., Hamann, L., Kuhlmann, M., Lindow, A.: On Better Understanding OCL Collections *or* An OCL Ordered Set Is Not an OCL Set. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 276–290. Springer, Heidelberg (2010)
7. Cattell, R.G.G., Barry, D.K.: The Object Data Standard: ODMG 3.0. Morgan Kaufmann (2000)
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. Hartmann, S., Link, S.: Collection Type Constructors in Entity-Relationship Modeling. In: Parent, C., Schewe, K.-D., Storey, V.C., Thalheim, B. (eds.) ER 2007. LNCS, vol. 4801, pp. 307–322. Springer, Heidelberg (2007)
10. Hoogendijk, P.F., Backhouse, R.C.: Relational Programming Laws in the Tree, List, Bag, Set Hierarchy. Sci. Comput. Program. 22(1-2), 67–105 (1994)
11. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006)
12. Kelsen, P., Ma, Q.: A Lightweight Approach for Defining the Formal Semantics of a Modeling Language. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 690–704. Springer, Heidelberg (2008)
13. 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)
14. Roldan, M., Duran, F.: Dynamic Validation of OCL Constraints with mOdCL. ECEASST 44 (2011)
15. Samimi, H., Aung, E.D., Millstein, T.: Falling Back on Executable Specifications. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 552–576. Springer, Heidelberg (2010)
16. Schürr, A.: A New Type Checking Approach for OCL Version 2.0? In: Clark, A., Warmer, J. (eds.) Object Modeling with the OCL. LNCS, vol. 2263, pp. 21–41. Springer, Heidelberg (2002)
17. Soeken, M., Wille, R., Drechsler, R.: Encoding OCL Data Types for SAT-Based Verification of UML/OCL Models. In: Gogolla, M., Wolff, B. (eds.) TAP 2011. LNCS, vol. 6706, pp. 152–170. Springer, Heidelberg (2011)
18. Van Der Straeten, R., Pinna Puissant, J., Mens, T.: Assessing the Kodkod Model Finder for Resolving Model Inconsistencies. In: France, R.B., Kuester, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 69–84. Springer, Heidelberg (2011)

19. 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)
20. Torlak, E., Vaziri, M., Dolby, J.: MemSAT: checking axiomatic specifications of memory models. SIGPLAN Not. 45, 341–350 (2010), http://doi.acm.org/10.1145/1809028.1806635
21. Wegmann, A., Le, L.-S., Hussami, L., Beyer, D.: A Tool for Verified Design using Alloy for Specification and CrocoPat for Verification. In: Jackson, D., Zave, P. (eds.) Proc. First Alloy Workshop (2006)
22. Willink, E.D.: Modeling the OCL Standard Library. ECEASST 44 (2011)
23. Wong, L.: Polymorphic Queries Across Sets, Bags, and Lists. SIGPLAN Notices 30(4), 39–44 (1995)

A Collection Creation and Sorting Algorithms

The presented algorithms abstract from complex language characteristics of relational logic and are designed to clarify the overall activities for creating and sorting UML/OCL collections at the relational level.

```
collectionCreation(colKind, ... elements)
input: the required collection kind (colKind ∈ {set,bag,ord,seq}),
       a list of elements already available in relational representation
output: a collection of kind colKind including the properly ordered elements
  newCol ← []
  index ← 1
  for each e in elements do
    if colKind is bag or sequence or e does not already exist in newCol then
      indexed_e ← add index as first component to each tuple of e
      newCol ← newCol ∪ indexed_e
      index ← index + 1
    end
  end
  newCol ← newCol ∪ typing tuple
  if newCol is set or bag then
    return complexSort(newCol)
  else
    return newCol
  end
```

Algorithm 1: Creating relations for representing UML/OCL collections

```
complexSort(col)
input: an unsorted relation col representing a set or bag
output: a relation with sorted elements
  predecessorMap ← empty map
  for each e1, e2 in col do
    if complexPredecessor(e1,e2) = e1
      or (complexPredecessor(e1,e2) = Undefined
          and original e1 position < original e2 position) then
      predecessorMap.add(e1,e2)
    end
  end
  positionMap ← empty map
  for each e in col do
    positionMap.add(e, |predecessorMap(e)|+1)
    modified_e ← e with topmost index replaced by positionMap(e)
    col ← col with e replaced by modified_e
  end
  return col
```

Algorithm 2: Sorting relations representing sets or bags

```
complexPredecessor(e1, e2)
input: two complex elements
output: the preceding element
  if e1 is a singleton, i. e., a relation including just a simple value then
    return simplePredecessor(e1, e2)
  else
    if e1 is undefined and e2 is undefined then return Undefined end
    if e1 is undefined and e2 is not undefined then return e1 end
    if e1 is not undefined and e2 is undefined then return e2 end

    if e1 is a set and e2 is not a set then return e1 end
    if e1 is not a set and e2 is a set then return e2 end

    if e1 is a sequence and e2 is not a sequence then return e1 end
    if e1 is not a sequence and e2 is a sequence then return e2 end

    if e1 is a bag and e2 is not a bag then return e1 end
    if e1 is not a bag and e2 is a bag then return e2 end

    if e1 has less elements than e2 then return e1 end
    if e2 has less elements than e1 then return e2 end
    relevantElement ← null
    for each position i in e1 and e2 do
      e1_elem ← element at position i of e1
      e2_elem ← element at position i of e2
      if complexPredecessor(e1_elem, e2_elem) = Undefined then
        continue
      else
        if complexPredecessor(e1_elem, e2_elem) = e1_elem then
          return e1
        else
          return e2
  end end end end
  return Undefined
```

Algorithm 3: Determining the precedence of elements

```
simplePredecessor(TO, e1, e2)
input: a binary relation TO specifying a total order for all simple values
       so that if [x,y] ∈ TO, x is the direct predecessor of y,
       two simple elements
output: the preceding element
  if e1 = e2 then
    return Undefined
  else if [e1,e2] ∈ closure(TO) then
    return e1
  else
    return e2
  end
```

Algorithm 4: Determining the precedence of simple values

Model Interchange Testing: A Process and a Case Study

Maged Elaasar[1] and Yvan Labiche[2]

[1] IBM Canada Ltd, Rational Software, Ottawa Lab
770 Palladium Dr., Kanata, ON. K2V 1C8, Canada
melaasar@ca.ibm.com
[2] Carleton University, Department of Systems and Computer Engineering
1125 Colonel By Drive, Ottawa, ON K1S5B6, Canada
labiche@sce.carleton.ca

Abstract. Modeling standards by the Object Management Group (OMG) enable the interchange of models between tools. In practice, the success of such interchange has been severely limited due to ambiguities and inconsistencies in the standards and lack of rigorous testing of tools' interchange capabilities. This motivated a number of OMG members, including tool vendors and users, to form a Model Interchange Working Group (MIWG) to test and improve model interchange between tools. In this paper, we report on the activities of the MIWG, presenting its testing process and highlighting its design decisions and challenges. We also report on a case study where the MIWG has used its process to test the interchange of UML and SysML models. We make observations, present statistics and discuss lessons learned. We conclude that the MIWG has indeed defined a rigorous, effective and semi-automated process for model interchange testing, which has resulted in more reliable interchange of models between participating tools.

Keywords: Model, Interchange, MOF, UML, SysML, XMI, OCL.

1 Introduction

Model Driven Architecture (MDA) [1] is an approach to the development of systems advocated by the Object Management Group (OMG). MDA encourages the specification of system functionality as models. A model organizes information based on a metamodel. A metamodel describes the concepts and relationships of a modeling language, like UML [2] and BPMN [3]. A metamodel is itself organized based on a modeling language called MOF [4]. The OMG standardizes many modeling languages, including the aforementioned. One of the main challenges of the OMG has been representing models in a machine-independent format that allows their interchange between modeling tools. This led to the definition of XMI [5], a standard for representing MOF-based models in XML.

Many modeling tools have implemented XMI since its introduction. However, the success of XMI as an interchange format has been severely limited (Section 6) for two main reasons. First, XMI defines a set of complex mapping rules between MOF and XML (e.g., properties whose values are the default values are not serialized) that

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 49–61, 2012.

have not been consistently implemented by tools. Second, in an attempt to be flexible, XMI provides a number of mapping options (e.g., a property may be serialized as an XML element or an XML attribute.), making it harder for tools that implement different options to interchange models.

However, other important reasons for the limited success of model interchange has in fact little to do with XMI itself. First, modeling language specifications do not clearly define how their diagrammatic notation maps to their metamodel. This often leads to tools representing models (typically defined through diagrams) differently using the metamodel. Second, for logistical and/or competitive reasons, tool vendors are not necessarily motivated to improve their model interchange support.

This dismal state of model interchange had long motivated the OMG to seek a solution. An early attempt to define a test suite for interoperability certification was not successful due to lack of support and resources. More recently, a number of OMG members formed a Model Interchange Working Group (MIWG) with the goal of testing and improving interoperability between tools. The group had more success due to the following reasons: (i) market pressure from large users who use multiple tools (e.g., the US Department of Defense—DoD); (ii) involvement of major tool vendors who could instigate change; (iii) involvement of experts as neutral parties to interpret and fix the standards; (iv) available resources and technology to automate some of the testing activities; (v) agreement among the parties involved to carry out the testing activities in private and to control the publicity of the results so as to be able to share proprietary information.

In this paper, we report and reflect on the activities of the MIWG. In particular, we present a rigorous model interchange testing process defined by the MIWG. The process allows the definition and execution of test cases (creating models, exporting them, then importing them) using participating tools. We discuss the process's design decisions, challenges and tool support. In addition, we report on a case study where the process was used to test model interchange between tools supporting two standard modeling languages, namely UML and SysML [6] (a profile of UML). We describe the executed test cases and highlight interesting results and lessons learned.

The rest of this paper is structured as follows: Section 2 describes the entire model interchange testing process; strategies that were used to improve the scalability of the testing process are described in Section 3; Section 4 reports on a case study involving the interchange testing of UML and SysML models; a discussion of the results and outstanding issues is provided in Section 5; Section 6 highlights related work. Finally, conclusions and future work are discussed in Section 7.

2 Model Interchange Testing Process

The first task of the MIWG was to define a process for model interchange testing between tools of a given modeling language. The focus was on testing the interchange of models, which are instances of MOF-based metamodels, hence interchangeable with XMI. Out of scope was the interchange of the diagrammatic notation, which is

still defined informally with text and pictures (a problem that the OMG is trying to address with the new Diagram Definition specification [7]).

The MIWG testing process, depicted in Figure 1 with a UML activity diagram, involves defining and executing a number of model interchange test cases. Each test case is designed to test an area of a modeling language. The area under test could be large (e.g., UML Sequence Diagrams) or small (e.g., specific types of Actions in UML Activity Diagrams). Test case execution entails defining a reference model, exporting it from one tool and importing it into another. The incremental testing process involves four roles (represented by the activity diagram's swim lanes in Figure 1). These roles can be played by one or more of the parties who participate in the process. As indicated in the diagram, the roles are *MIWG*, which defines test cases, *producer* and *consumer* who use their tools to export/import models, thereby executing test cases, and *implementer* who resolves issues detected by the process. Each role and the activities it performs are further discussed below.

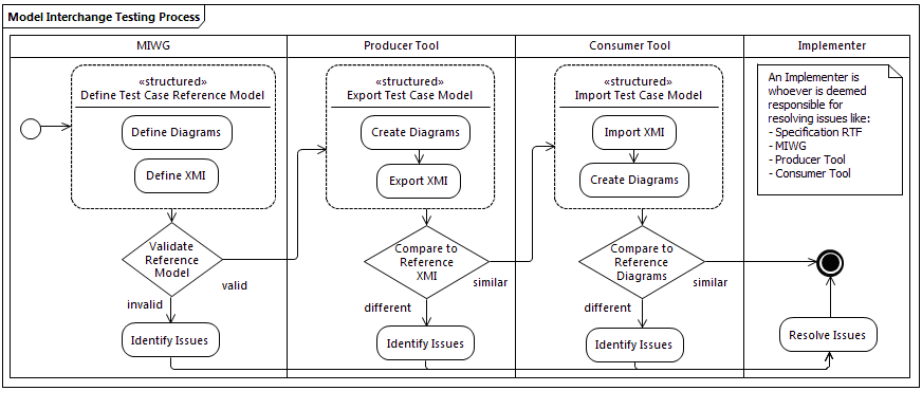


Fig. 1. The MIWG testing process (for one test case)

The process is initiated by the MIWG defining test cases that cover areas of a modeling language for which the MIWG feels it important that tools provide accurate interchange. Then, for each test case, a small exemplary reference model that covers the tested area is created. A reference model, typically defined by an expert who is familiar with the syntax of the related area, is specified with one or more diagram images representing the concrete syntax of the model, and an XMI file representing the abstract syntax of the model. Since it would be tedious to produce these artifacts manually, one of the participating tools is used to produce drafts. If necessary, the MIWG may edit these drafts manually using a simple text or image editor. As a byproduct of this activity, the MIWG sets guidelines for test case definition to avoid ambiguities (e.g., a missing visibility symbol in a UML diagram means an element's visibility is not set vs. being notationally elided). These guidelines are documented either on the diagrams (as notes) or on the MIWG's wiki [8] (when applicable to several diagrams). A reference model is only declared ready after having been validated by the MIWG. Identified issues are resolved by an implementer who can

either be the MIWG (for issues with test cases) or a relevant standard's revision task force (RTF) (for issues with that standard). Notice that the reference model serves as a test oracle [14], which is an artifact that determines the expected representation of a model exported from a producer tool when a test case is executed.

Once its reference model is ready, execution of a test case starts. The first step is for each participating vendor (referred to as a producer in Figure 1) to manually recreate (as opposed to import) the reference model using capabilities of his/her tool. If the tool does not support some of or all the areas being tested, the producer can provide documentation about those limitations, along with any possible workarounds. Then, the producer exports the model as XMI and the diagrams as images. After that, the producer compares the exported XMI to the reference one, identifies issues and analyzes them. Depending on the root causes of the issues, they get resolved by an implementer who can be the producer (for differences with the reference model), the MIWG or a relevant standard RTF.

Once the exported artifacts by a producer are available, each of the other participants (referred to as a consumer in Figure 1) proceeds to the importing step. In this step, a consumer creates a model by importing the exported XMI of the producer then recreating similar diagrams by dragging imported model elements onto them. Notice that diagrams are either not included in the XMI exports or included in a proprietary format within XMI:extension tags, and therefore not necessarily imported by the consumer tool, thus the need to manually creating the diagrams. If the consumer tool does not support some of the areas being tested, it can provide documentation about those limitations, along with any possible workarounds. After that, the consumer exports the diagrams as images for comparison with the reference images. Although not required by the process, exporting the imported model to XMI for comparison with the reference XMI can be done. During the comparison, issues are identified, analyzed, and eventually resolved by an implementer who can be the consumer (for issues related to importing the XMI file or differences with the reference diagrams not caused by the producer), the producer, the MIWG, or a relevant standard RTF.

Based on the description of the MIWG testing process given above, and assuming N tools are involved and T test case specifications (i.e., reference models) are created, we note that the process involves N exports for each of the T test cases, followed by N-1 imports for each export. This gives the process a linear scalability of [T.N] on export, but a polynomial scalability of [T.N.(N-1)] on import. We also note that the process is parameterized by the following parameters: the standards being tested, the participant tools and the test cases. Since some or all of these parameters may be subject to revision during a testing period, partly to address identified issues, the testing process may need to be re-executed multiple times, hindering scalability. Finally, we note that some of the activities of the process, like issue identification and analysis, are done manually, which makes them tedious and error-prone. In the following section, we discuss how the MIWG improved the scalability of its process.

3 Improving Scalability of the Testing Process

Although the MIWG testing process (as defined in Section 2) was found to be reasonably effective (Section 4) in resolving the most egregious problems of model interchange (e.g., inconsistent XMI support), its execution required excessive effort. The polynomial scalability on import (a tedious activity that includes manually creating and comparing diagrams) was prohibitive. The consensus of the group was that testing N-1 imports per export may no longer be necessary once the main roadblocks of model interchange were resolved. Instead, the group agreed that validating the exported models by comparing them to the reference models, along with testing the import of the reference models only, may suffice going forward.

The activity diagram in Figure 2 shows the revised MIWG testing process. Notice that after validation of the reference model by the MIWG, both producers and consumers can proceed with their activities and that the import activity involves only the reference model (not the exports by the other N-1 producers). As a result, the process goes from N-1 to 1 import per consumer, giving the revised process a linear scalability of [T.N] on import. However, the process may reveal issues where exports deviate from reference models regardless of whether consumer tools can handle the deviation. Therefore, the revised process provides weaker, less direct evidence that tools can interchange models. In contrast, using the original process, all N participants might incorrectly, though consistently, export and import a model and therefore no interchange problem would be exposed among those participants.

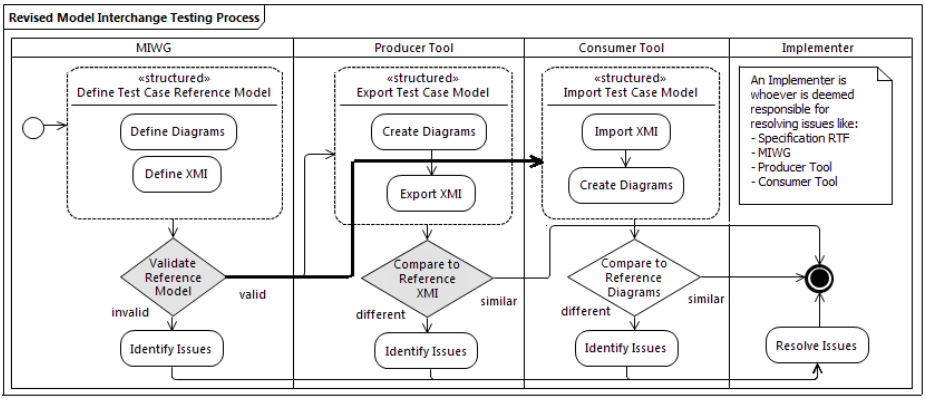


Fig. 2. The revised MIWG testing process (for one test case)

Another change in the revised process, highlighted by grey diamonds in Figure 2, is automating the validation of XMI files and their comparison to the reference XMI. The MIWG developed a web-based tool called the Validator [9] (hosted by the US National Institute of Standards and Technology) to automate this task. The tool (Figure 3) allows uploading an exported XMI file and comparing it to the corresponding reference XMI. The result is a detailed report outlining: (a) compliance issues with the relevant metamodel (found by checking type, multiplicity, and OCL

constraints) and XMI (found by checking the XMI rules) and (b) differences with the reference model (found by structurally comparing the two models).

In fact, comparing the two models was initially attempted at the text level by converting both files to Canonical XMI [11], a dialect of XMI contributed by the MIWG. Recall from Section 1 that XMI has many options, increasing both its flexibility and complexity. The Canonical dialect of XMI eliminates these options by fixing their values (e.g., a root XMI element is always required, all properties must be specified as XML elements except those in the XMI namespace, like xmi:id, and uuids are mandatory). Once in that dialect, files could be compared side by side to show text deltas. However, comparison by this method reports low level deltas (e.g., line x has changed) that are hard to analyze. Consequently, the group switched to structural comparison, where models are treated as graphs of elements and compared hierarchically. The elements in both graphs are matched by their type, qualified names and/or other characteristics. This method reports higher level deltas (e.g. attribute x has a different value) that are much easier to understand and analyze.

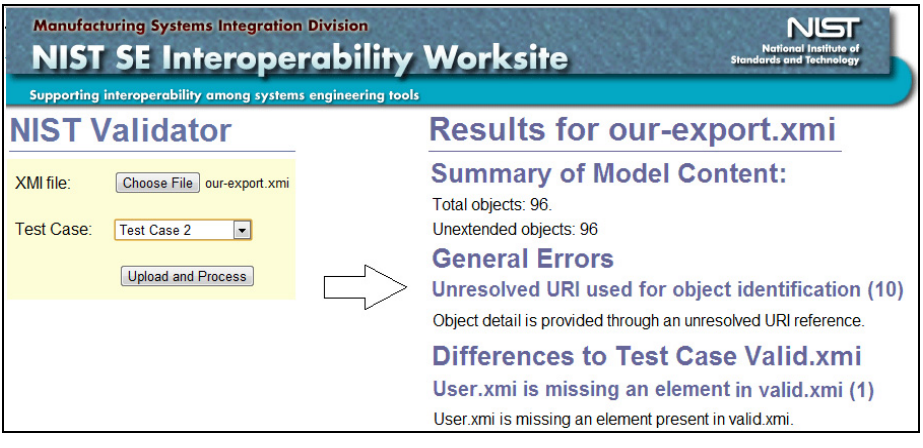


Fig. 3. The Validator Tool: the our-export.xmi file was uploaded and tested against test case 2 (left); validation results (excerpt) are reported (right)

4 Case Study: Interchange of UML and SysML Models

The MIWG validated its testing process by interchanging UML 2.3 and SysML 1.2 models using XMI 2.1. The goal was to assess whether the process was effective in testing and improving interoperability between participating tools through the identification and resolution of model interchange issues. We describe the setup and execution of this case study. We also report on and analyze its results.

4.1 Case Study Setup

The MIWG chose UML and SysML (a profile of UML) mainly because of market pressure from influential users (like DoD). Moreover, both languages are large, complex and popular making them good candidates for testing model interchange and validating the testing process. Table 1 shows the tools that participated in this case study, along with their vendors and latest tested versions.

The case study involved defining a test suite consisting of 16 test cases, summarized in Table 2. Twelve of these were for UML and five were for SysML (TC12 pertained to both UML and SysML). Notice that SysML tests alone would not have sufficed since not all of UML is used by SysML Most test cases were defined with multiple diagrams. They were also designed to be small in size (to permit easy checking) yet maximize coverage of their language areas. The interested reader is referred to [8] for more details on the test cases and their actual coverage of UML and SysML model elements.

For each test case, the MIWG chose a tool with which to define a reference model. The chosen tool was typically one that supported most (if not all) of the required notation and XMI syntax for the test case. However, for practical reasons, the choice of the tool was more often motivated by the fair distribution of workload between the MIWG members. The gaps in support and the (notation and XMI) imperfections were overcome with manual edits using an image or a text editor. The effort required to produce the reference model varied with the complexity of the test case and the group's level of expertise with the relevant areas of the specifications. It mainly involved manual inspection of the diagrams and the XMI and checking their validity against the specifications. In some cases, this was partially automated with the model validation features of some tools. As the group gained experience, defining valid reference models became easier.

In addition, during reference model definition, some information in the XMI was found to be not representable in the notation (e.g., the visibility of UML classes or which library of primitive types was used). In this case, notes were added to the diagrams to document them. (A note was used rather than a UML comment since the latter would appear in the XMI.) In some cases, the MIWG documented guidelines that applied to all test cases of the case study on its wiki [8].

Table 1. The tools participating in the case study

Vendor	Tool	Version
Atego	Artisan® Studio	7.2m
IBM	RSx	8.0.3
IBM/Sodius	IBM Rhapsody	7.6.x
No Magic	MagicDraw	17.0
SOFTEAM	Modelio	2.4.19
Sparx Systems	Enterprise Architect	9.1

Table 2. Test cases of UML and SysML

UML Test Cases		**SysML Test Cases**
TC1–Simple Class Model	TC7–State Machines	TC10–Blocks
TC2–Advanced Class Model	TC8–Use Cases	TC 11-Requirements
TC3–Profile Definition & App.	TC9–Interactions	TC12a–Activity Swim Lanes
TC4–Simple Activity Model	TC12b–Activity Swim Lanes	TC14-Parametrics
TC5–Advanced Activity Model	TC13–Instance Specifications	TC16-Allocations
TC6–Composite Structure	TC15–Structured Activity Nodes	

4.2 Case Study Execution

The case study was carried out over the course of 30 months and in two separate phases. In the first phase, which spanned the first 21 months, the MIWG executed the original testing process as defined in Section 2. Based on 16 test cases and 6 participating tools, a total of 96 (16x6) exports and 480 (16x6x5) imports were run, at least once each. Some had to be rerun multiple times as standards, test cases and/or tools were being revised. This alerted the group to the process's scalability problems. Hence, in the second phase, which spanned the next 9 months, the MIWG decided to switch to the revised process as defined in Section 3. In contrast to the first phase, a total of 192 (16x2x6) imports had to be run only in this phase.

4.3 Case Study Results

Before discussing the results, it is important to note that no tool-specific results will be disclosed, in accordance to an agreement among the MIWG members. In fact, the MIWG originally tried to keep a tool capability matrix but that quickly proved problematic because it was (i) commercially sensitive and (ii) hard to maintain, especially since tools continuously undergo revision that may positively or negatively impact their model interchange support. Instead, the MIWG agreed to give the general public the ability to assess tools on their own at any time. Specifically, the MIWG made the exported test case models from each tool available [8]. These models can be compared to the reference models using the Validator tool [9], which was used during the revised process in this case study. These assessments may be used for evaluating the interchange capabilities/limitations of a particular tool, as part of a tool selection process, or to set expectations for the use of a tool in a project.

In the remainder of this section, we report on the overall results of this case study. During the first phase, the exported and imported models were validated and compared to the reference models manually by the participants. This helped uncover major issues, especially with the tools' support of the UML metamodel and SysML profile, which hindered the successful interchange of models. For example, it showed that tools represented models with similar diagrams differently using the metamodel and/or profile. It also showed that tools were not consistently using the

official URLs. Some effort was spent at the beginning to address these issues to unblock interchange.

Additionally, it was observed that some tools exported extra model elements (e.g., container elements) that were not required by test cases. Although legal according to the UML/SysML standard, such elements were sometimes not expected by consumer tools. Similarly, some producer tools exported non-standard information (e.g., diagrams) in the XMI files. This was legal when such information was enclosed in XMI: extension elements to allow a consumer tool to ignore it. The MIWG proposed that producer tools use the XMI:exporter tag to specify their tool name. This can help consumer tools recognize and optionally process these extensions if they can.

Furthermore, we observed that the case study achieved good test coverage of UML and SysML standards (59% of UML metaclasses and 55% of SysML stereotypes). However, we also observed that tools were not consistent in their support of some areas of the standards. This could be attributed to ambiguities in these standards and/or to bugs in the tools. For example, tools did not have consistent default values for the multiplicity of a UML typed element or for the visibility of a UML packageable element. The MIWG clarified to vendors that the former should be *1..1* and the latter should be *public*. Another example was the names of SysML stereotypes that were sometimes different in XMI from those shown in the graphical notation when applied to UML elements.

During the second phase of the case study, consumers manually analyzed and reported on their imports of reference models. On the other hand, producers reported on their exports using the Validator tool, which allowed them to: (a) analyze their conformance to the (XMI, MOF and modeling language) standards and (b) compare them to the reference models. Figure 4 shows the results of the conformance analysis. The lines represent the average number of (all occurrences of all) issues/bugs reported by tools for their first (dashed line) and their last (solid line) export of each test case (on average vendors exported each test case 3-4 times due to bug fixes and/or changes in reference models). The average number of issues across all test cases was 65 at the start and 52 at the end (a drop of 20%). However, the number increased for some test cases (e.g., 9) due to ambiguities in related areas of UML (e.g., sequence diagrams) that made most tools inconsistent with it. Best results were achieved for Class and Instance diagrams (test cases 1, 2 and 11) and SysML requirements (test case 13).

On the other hand, when comparing the exported models to the reference ones (Figure 5), we observe that the average number of differences across all test cases was 26 at the start and 24 at the end (a drop of 8%). (The lines in Figure 5 represent the average number of differences reported by tools for each test case.) However, we note that differences did not always decrease (e.g., test case 4, 10 and 15). While this might be due to human error while recreating the reference models (e.g., naming elements differently), it could also be a symptom of the complexity of some areas of the standards (e.g., advanced activities), which causes implementation difficulties.

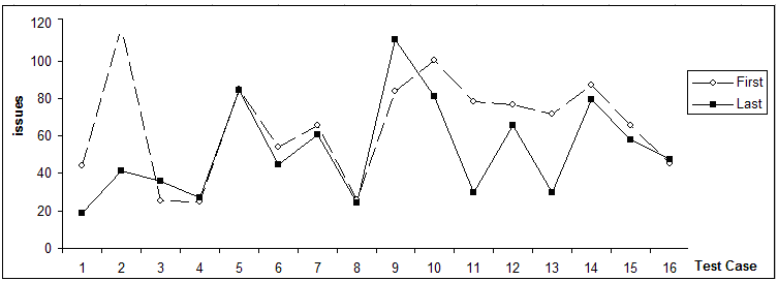


Fig. 4. Issues related to conformance to standards before and after case study

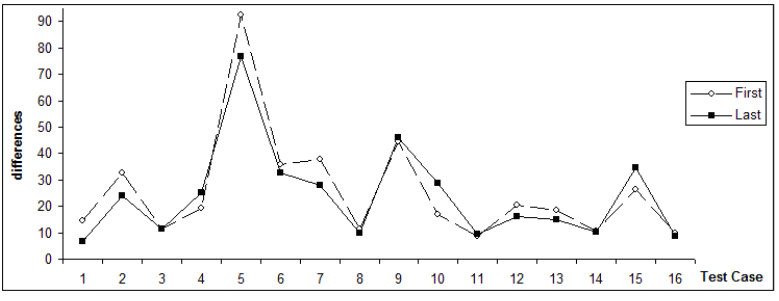


Fig. 5. Differences to reference models before and after case study

5 Discussion

The case study, presented in Section 4, shows that the MIWG testing process is effective for assessing model interchange between tools. It also shows that tool vendors can work together to improve model interchange. Versions of participating UML and SysML tools, available now on the market, are more interoperable than they were when the MIWG initiated its work, although there is certainly room for further improvement. Interoperability can directly benefit users as it may significantly improve quality and productivity and increase the investment in models.

The MIWG finds that XMI is an adequate model interchange mechanism. Nonetheless, the group admits that XMI 2.1 is a complex standard, though much of the complexity has been reduced in XMI 2.4 with the definition of Canonical XMI, as a result of this case study. The group also finds that most of the remaining interchange issues relate to either ambiguities/inconsistencies in the UML/SysML standards or bugs in the tools' support of these standards, rather than with the tools' XMI support. For example, a handful of issues have been logged against UML sequence diagrams that the standard does not fully define how they map to the UML metamodel. Another issue is against SysML, where one of the UML constraints (on property *Property::partWithPort*) does not apply in the context of SysML.

One of the challenges of model interchange that has surfaced in this case study is that tools differ in the versions of the standards they support. For example, while most participating tools have moved to UML 2.3, some have not, while others have moved

to 2.4 already. Those that did not move still managed to participate by manually editing their models. However, the authors[1] recommend that the testing process only moves to a newer version of a standard when a significant number of participating tools have moved to it. The authors also recommend that standards remain backward compatible in their minor revisions to motivate vendors to adopt them quicker.

Regarding the MIWG process, the authors note that the revised process is still quite labor intensive. Specifically, it requires expertise in defining and validating the reference models. Fortunately, once those models are defined, they can be reused by new tools. The process also requires the MIWG to manually analyze issues reported by the Validator tool to identify root causes. Furthermore, by not importing the exported models directly, the revised process provides a weaker proof of interchange. It may also expose bugs in producer tools that could have been tolerated by importer tools in the original less scalable process. Therefore, the authors recommend that the original process still be initially executed, at least once per test case, before switching to the more scalable revised process. Finally, the process focuses on testing the syntactical interchange of models only. It does not include testing the semantics, which relate to how models are used (e.g., executed).

6 Related Works

Although everyone who is working, or has worked, in a MDA context has anecdotal evidence that there are issues with model interchange, there has been very little published work that systematically studies those issues.

Alanen et al. [14] performed a simple case study whereby they created a simple class diagram (not focusing on specific features of the modeling language) using six different UML modeling tools. They then exported the diagram to XMI and manually compared, using a text editor, the six generated XMI files. They observed some discrepancies in the files and concluded (like the MIWG) that one of the main issues when practicing model interchange was the different versions of XMI and UML standards supported by tools. They advocated the need for a compliance test suite for checking XMI compatibility, which is an outcome of the MIWG work.

Persson et al. studied XMI compatibility between six commercial and three open source modeling tools [15], using one model (class diagram). They showed that XMI-based model interchange between UML modeling tools was weakly supported in practice. In a different publication [16], they experimented with two industrial size models, one commercial tool for creating the models and three open source tools to interchange with. The experimental procedure was the following, for each of the two models: (1) create the model using the commercial tool; (2) export the model to XMI and check conformance to the XMI standard using available automated XML checkers; (3) import the XMI using each of the three open source software tools (some imports failed even after changing the input XMI file to something expected by the tool) and visually check the diagram obtained; (4) export from the open source software, possibly fixing the generated XMI file; (5) import (back) into the

[1] Some comments made by the authors of this paper are not necessarily the ones of the MIWG.

commercial tool; and (6) visually compare the obtained diagram with the one they started the process with. They reported that manually fixing the XMI files was necessary so they conformed to the XMI standard, and sometimes fixes were insufficient to successfully complete the process. They concluded that model interchange based on XMI was not mature enough to be used in an industrial setting. They conducted a similar study in 2006 [17], using a different set of producer/consumer tools and a much simpler class diagram model (without focusing on important features of the language), and confirmed the limitations of the export/import, although they noticed an improvement when using the new XMI 2.0.

More recently, Eichelberger et al. [18] report on a comprehensive survey of the compliance of current modeling tools to the UML standard, focusing on a large set (476) of UML modeling features. With respect to XMI, they studied the structural compliance of exported XMI files to the XMI standard. They report that only four out of 68 tools have an acceptable level of compliance. With respect to XMI, they report that 47% of the tools do not pass the structural XMI validity test, 3% (i.e., two) tools, pass the XMI validity test, while the remaining 50% offer no XMI at all. They did not however try to import the XMI files into other tools.

In summary, some have attempted to study model interchange with XMI before. They differ from the process we discuss in this paper in one or more of the following ways: (i) They did not specifically and systematically define targeted reference models (they instead used available models); (ii) They did not necessarily systematically investigate pairs of producer/consumer tools; (iii) They did not involve the vendors of the producer/consumer tools to investigate and fix the root causes of encountered issues; (iv) They did not try to automate parts of the process.

7 Conclusion and Future Work

Modeling standards enable automated exchange of modeling information among tools. Due to errors and ambiguities with those standards and the lack of rigorous testing between tools, the full benefit of model interchange could not be realized. The MIWG has defined and validated a rigorous incremental model interchange testing process. The process has been fine tuned to scale with the number of participating tools. It has also been used in a case study to assess UML and SysML model interchange between six tools. A suite of 16 test cases has been defined and executed. The case study has led to improving interoperability between these tools as well as these tools' conformance to the standards (by 20%). The MIWG has also made the process public and partially repeatable to allow interested communities to assess the interchange capabilities and limitations of participating tools for their purposes.

Going forward, the MIWG plans to test other model interchange situations, like importing XMI files conforming to older versions of the standards or importing fragmented models with cross references. Another area of future work is round trip (export-import-change-export-import) testing. Yet another area is testing diagram interchange (when language-specific diagram definition standards become available

and implemented). Finally, newer versions of UML/SysML/XMI specifications will be tested, in addition to other standards (e.g., UPDM [12] and SoaML [13] profiles).

Acknowledgements. The authors would like to thank the entire MIWG team for the great work this paper reports on. The authors would also like to particularly acknowledge the following MIWG members: Peter Denno (NIST) and Pete Rivett (Adaptive), for reviewing this paper.

References

1. Model Driven Architecture, http://en.wikipedia.org/wiki/Model-driven_architecture
2. Unified Modeling Language, Superstructure v2.4.1., http://www.omg.org/spec/UML/2.4.1
3. Business Process Model and Notation v2.0., http://www.omg.org/spec/BPMN/2.0/
4. Meta Object Facility Core v2.4.1., http://www.omg.org/spec/MOF/2.4.1/
5. MOF 2 XMI Mapping v2.4.1., http://www.omg.org/spec/XMI/2.4.1/
6. Systems Modeling Language, v1.2., http://www.omg.org/spec/SysML/1.2/
7. Diagram Definition v1.0 FTF Beta 2., http://www.omg.org/spec/DD/1.0/Beta2/
8. MIWG Wiki, http://www.omgwiki.org/model-interchange
9. NIST Validator, http://syseng.nist.gov/se-interop/sysml/validator
10. OMG Object Constraint Language v.2.3.1., http://www.omg.org/spec/OCL/2.3.1/
11. Canonical XMI, FTF Beta 1, http://www.omg.org/cgi-bin/doc?ptc/12-01-01
12. Unified Profile for DoDAF and MODAF v2.0., http://www.omg.org/spec/UPDM/2.0/
13. Service Oriented Architecture Modeling Lang. v1.0., http://www.omg.org/spec/SoaML/1.0/
14. Alanen, M., Porres, I.: Model Interchange Using OMG Standards. In: Proc. of the 31st EUROMICRO Conf. on Soft. Eng. and Advanced Apps., pp. 450–459 (September 2005)
15. Persson, A., Gustavsson, H., Lings, B., Lundell, B., Mattsson, A., Ärlig, U.: OSS tools in a heterogeneous environment for embedded systems modelling: an analysis of adoptions of XMI. In: Proc. of the 5th Workshop on Open Source Software Engineering (May 2005)
16. Persson, A., Gustavsson, H., Lings, B., Lundell, B., Mattsson, A., Ärlig, U.: Adopting Open Source Development Tools in a Commercial Production Environment—Are we Locked-in? In: Proc. of 10th EMMSAD (June 2005)
17. Lundell, B., Lings, B., Persson, A., Mattsson, A.: UML Model Interchange in Heterogeneous Tool Environments: An Analysis of Adoptions of XMI 2. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 619–630. Springer, Heidelberg (2006)
18. Eichelberger, H., Eldogan, Y., Schmid, K.: A Comprehensive Survey of UML Compliance in Current Modelling Tools. In: SE 2009. LNI, vol. 143, pp. 39–50 (2009)

An Internal Domain-Specific Language for Constructing OPC UA Queries and Event Filters

Thomas Goldschmidt and Wolfgang Mahnke

ABB Corporate Research Germany,
Industrial Software Systems Program
{thomas.goldschmidt,wolfgang.mahnke}@de.abb.com

Abstract. The OPC Unified Architecture (OPC UA) is becoming more and more important for industrial automation products. The development of OPC UA components is currently supported by the use of SDKs for OPC UA. However, these SDKs provide only low level support for creating OPC UA based applications. This leads to higher development efforts. The domain-specific metamodel defined by OPC UA defines serves as a good basis for creating domain-specific languages on a higher abstraction level. This has the potential of reducing development efforts. In this paper, we focus on the event filter and query part of OPC UA. Current SDKs only provide interfaces for constructing an object tree for these queries and event filters programmatically. Creating and maintaining these object structures is tedious and error prone. Therefore, we introduce an internal DSL approach for constructing OPC UA queries and event filters based on the OPC UA information model and the Language Integrated Queries (LINQ) feature available in .Net.

1 Introduction

The OPC Unified Architecture (OPC UA) is becoming more and more important for industrial automation products. It is a central component to modern industrial applications. Classic OPC (OLE for Process Contol) defined an industry-wide adopted set of standards for accessing and distributing data in industrial systems. With the more recent OPC UA, new facilities like a unified address space model, service oriented interfaces and an extensible metamodel have been introduced. This allows OPC UA to be used from small embedded systems, industrial controllers, Distributed Control Systems (DCS) up to Manufacturing Execution Systems (MES) and Enterprise Resource Planning (ERP) systems.

The development of OPC UA components is currently supported by the use of special software development kits (SDKs) for OPC UA. These SDKs provide means for developing code that deals with the creation and navigation in the OPC UA information model, registration of monitors for value changes and calling of defined methods as well as connection and session handling. However, one of the largest drawbacks of working directly with these SDKs is that they

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 62–73, 2012.

often focus on dealing with technical interfaces rather than providing an easier programming model for developers. The OPC specification already proposes that higher level languages such as a graphical information modeling language [1] may be used to efficiently develop OPC UA applications. However, apart from the already mentioned graphical modeling of information models including code generation from it [2,3,4] no DSL support is provided by currently available OPC UA SDKs. More specifically, there is currently no support for defining OPC UA queries and event filters on a higher level of abstraction. Developers have to create them using object structures programmatically. This code, that accesses elements from the address space, has to deal with node identifiers or browse names that are mostly passed as string variables. During design time it is therefore not checked if the specific nodes and/or variables that code accesses actually exist in the used address space. Furthermore, the code is very lengthy and not clearly structured. This leads to error prone implementations and thus additional test effort while developing these OPC UA components.

In our previous work [5] we analyzed the different use cases in OPC UA development which may be improved by developing and using a DSL for them. One of the use cases where we identified the largest impact at a relatively low additional development effort was the creation of DSL for query and event filter creation. Therefore, we decided to build an internal DSL for this use case. Internal DSLs [6] are based on the syntax of a host language and add additional language constructs to that language which are in a compile step mapped towards structures of the host language. The advantage of such internal DSL is that existing IDEs can be reused without extensions and programmers do not need to learn a new syntax. A language feature of C#/.Net that is designed to be used in such a way is the Language Integrated Queries (LINQ) [7] feature. It provides a concrete syntax that is similar to SQL and uses a closure mechanism underneath. A query specified in LINQ is internally translated into an expression tree based on such a closure.

The contribution of this paper is an internal domain-specific language that facilitates LINQ to specify OPC UA queries and event filters. We provide a mapping specification of LINQ constructs to those used on OPC UA queries and event filters. Based on a generated object model of the target information model developers get code completion and error recognition for their specified queries. Furthermore, we demonstrate how the LINQtoOPCUA generator, which we developed, facilitates LINQ expression trees to instantiate the abstract syntax objects of OPC UA queries and event filters.

The remainder of this paper is structured as follows. A short introduction to OPC UA including its event filters and queries is given in Section 2. The query/event filter DSL that we designed is introduced in Section 3. A critical discussion of our DSL is presented in Section 4. Section 5 concludes this paper and outlines future work.

2 OPC Unified Architecture

OPC UA provides a secure, reliable, high-performing communication infrastructure to exchange different types of data in industrial automation. That includes

current data like measurements (e.g. from a temperature sensor) and setpoints (e.g. for defining the desired level of a tank), events (e.g. device lost connection) and alarms for abnormal conditions (e.g. a boiler reached a critical level). In addition, it provides the history of current data (e.g. the temperature trend the last day or the last ten years) and of events (what events of a certain type occurred the last five days). In order to provide semantic with the data, also meta data is exchanged in terms of an information model. In Figure 1 depicts an example of an OPC UA address space.

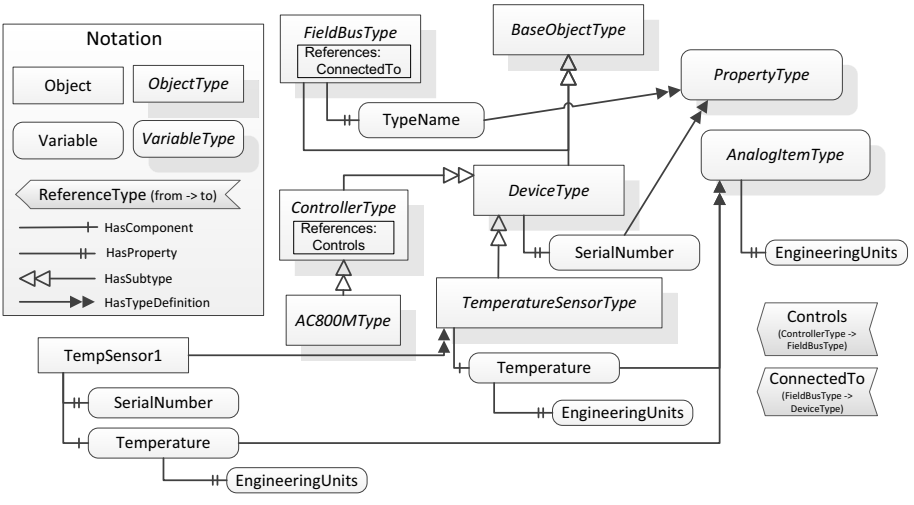


Fig. 1. Example of an Address Space in OPC UA

On the right hand of Figure 1 the type system is shown, with object types in a type hierarchy. For example, the *DeviceType* is an abstract object type representing all kinds of devices. It defines a variable called *SerialNumber*. A subtype *TemperatureSensorType* adds the *Temperature* variable, including the *EngineeringUnits*. Variables are typed as well, like the *Temperature* of type *AnalogItemType* defined by the OPC Foundation. This type adds a property to the variable containing the *EngineeringUnits*. On the left hand an instance of the *TemperatureSensorType*, *TempSensor1*, is shown. The instances contain the concrete values, like the temperature measured by *TempSensor1*.

OPC UA is based on a client server model where the client asks for data and the server delivers the data. The client has the option to read and write the data, but also to subscribe to data changes or event notifications. In addition, the client can browse the address space of the server and read the meta data information. For large and complex address spaces the client also has the capability to query the address space for information, for example asking for all temperature sensors that are currently measuring a temperature larger than 25 °C.

Clients can subscribe to events by subscribing to objects marked as event notifiers, which already provides some filtering based on how the objects are

structured in the address space. In addition, they can specify an event filter to define the event fields they want to receive as well as define the events they are interested in. To provide information about the structure of events supported by the server, the server provides an event type hierarchy. It uses the concept of abstract object types to define the event type hierarchy and variables to define the fields of events. In case of alarms concrete object types are used, as alarms can be represented in the address space in order to configure alarms, for example specifying the limit for an alarm. Figure 2 shows an example of an event type hierarchy. The *BaseEventType* is defined by the OPC UA specification [1], and the *DeviceStatusType* is a subtype providing the *HealthStatus* of a device. Clients can, for example, define an event filter for the *HealthStatus* reaching a critical level.

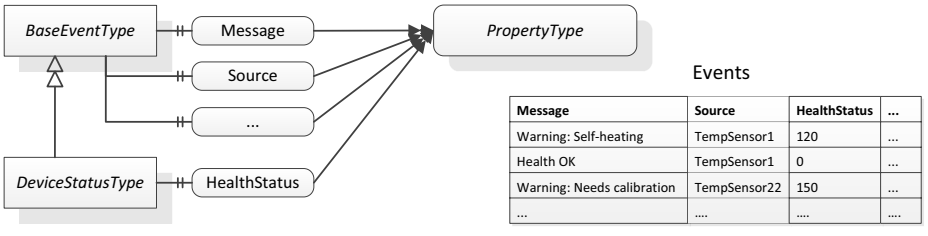


Fig. 2. Example of an Event Type Hierarchy in OPC UA

2.1 Event Filter Creation

Event filters in OPC UA are used in the creation of event subscriptions to define which kinds of events are relevant for a certain subscription. The specification [8] only defines the service interface used for registering such filters. Current OPC UA SDKs implement this service and provide a class which resembles the abstract syntax tree of the expressions. The specification informatively provides examples for concrete syntaxes (graphical, table-based or text based) for event filters. As OPC servers can be relatively thin and should not bother with extensive expression parsing, the OPC Foundation intentionally used the abstract syntax on the interface for these filters. However, users should not directly deal with the abstract syntax, but should use a nicely designed concrete syntax for defining these filters. Currently none of the SDKs implements a concrete syntax for event filters.

2.2 Query Creation

Similar to the event filtering mechanism the OPC UA specification defines query services on an abstract interface level. Concrete syntaxes are informatively proposed, but there is currently no implementation of any of these syntaxes by existing SDKs.

In contrast to the event filters, for which handlers are already implemented in the SDK and OPC UA stack, queries are mostly routed down to the server

implementation. On this layer the abstract syntax should then be mapped down to whatever query mechanism the underlying data source uses. This could for example be an SQL like language. In this case this use case is also relevant for server implementations.

2.3 Queries and Event Filters Metamodel

The OPC UA specification [8] already proposes that there should be a concrete syntax for event filters and queries. Both languages provide query like constructs, such as *select* and *where* clauses. OPC UA SDKs provide means for programmatically creating select, when and where clauses. However, writing the code that creates them causes a lot of overhead as each single select, where clause and all operands have to be instantiated and parametrized and then plugged together.

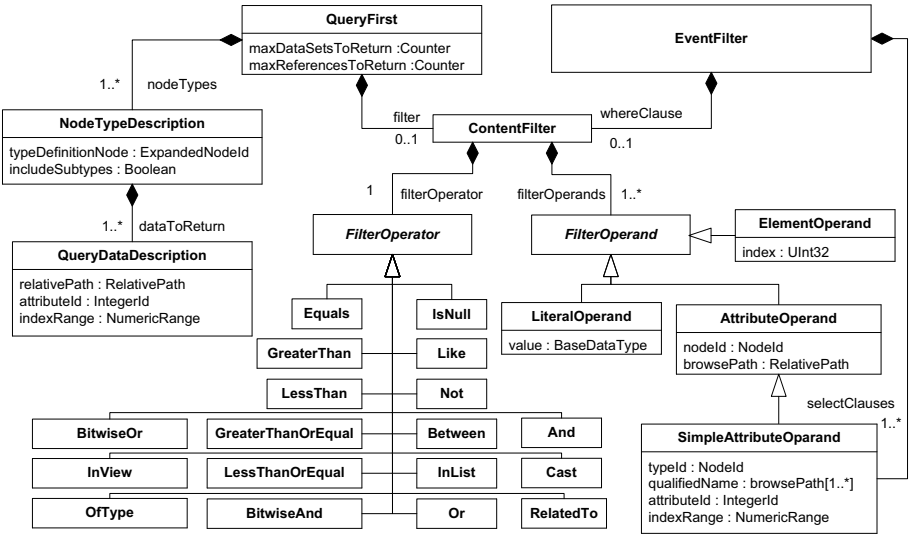


Fig. 3. Metamodel of the OPC UA queries and event filtes services

Figure 3 depicts a metamodel which we derived from the specification of the queries and event filter services given in [8]. SDKs implement this metamodel as classes of an abstract syntax. These classes are then instantiated and plugged together to create queries and event filters. Listing 1 gives an example constructing an event filter based on these classes using the OPC Foundation .Net SDK. Node identifiers and property names have to be provided as string parameters and the object tree is also constructed manually. The OPC UA specification defines that such constructs have to be validated and sanity-checked by servers that receive them at runtime. However, there is currently no way of checking their validity already at the time of development.

Listing 1. Example code that creates an OPC UA event filter that selects the message, the source and the health status of all events of type *DeviceStatusType* that have a *HealthStatus* greater than 100.

```
EventFilter eventFilter = new EventFilter();

SimpleAttributeOperand selectClause0 = new SimpleAttributeOperand();
Opc.Ua.QualifiedNameCollection browsePath;

// Select Event Field Message
selectClause0.AttributeId = Attributes.Value;
selectClause0.TypeId = DeviceStatus.NodeId;
Opc.Ua.QualifiedName propName = new Opc.Ua.QualifiedName("Message", 0);
browsePath = new QualifiedNameCollection();
browsePath.Add(propName);
selectClause0.BrowsePath = browsePath;

// Select Event Field Source
SimpleAttributeOperand selectClause1 = new SimpleAttributeOperand
    { AttributeId = Attributes.Value,
        TypeId = DeviceStatus.NodeId };
propName = new Opc.Ua.QualifiedName("Source", 0);
browsePath = new QualifiedNameCollection();
browsePath.Add(propName);
selectClause1.BrowsePath = browsePath;

// Select Event Field HealthStatus
SimpleAttributeOperand selectClause2 = new SimpleAttributeOperand
    { AttributeId = Attributes.Value,
        TypeDefinitionId = DeviceStatus.NodeId };
propName = new Opc.Ua.QualifiedName("HealthStatus", 1);
browsePath = new QualifiedNameCollection {propName};
selectClause2.BrowsePath = browsePath;

ContentFilter cf = new ContentFilter();

ContentFilterElement cfe = new ContentFilterElement {
    FilterOperator = FilterOperator.GreaterThan};
propName = new Opc.Ua.QualifiedName("HealthStatus", 1);
SimpleAttributeOperand op1 = new SimpleAttributeOperand(
    DeviceStatus.NodeId, propName);
LiteralOperand op2 = new LiteralOperand(100u);
cfe.SetOperands(new List<FilterOperand> { op1, op2 });
cf.Elements.Add(cfe);

// Add to filter
eventFilter.SelectClauses.Add(selectClause0);
eventFilter.SelectClauses.Add(selectClause1);
eventFilter.SelectClauses.Add(selectClause2);
eventFilter.WhereClause = cf;
```

3 The Query/Event Filter DSL

OPC UA client applications are often written using C#/.Net. C# provides a feature called "language integrated queries" (LINQ). LINQ provides a concrete, SQL-like syntax that is directly integrated into C#. It is intended to be mapped to specific query languages such as SQL depending on the target purpose. To achieve this, LINQ includes means for accessing the expression trees of these queries directly from the code. This enables us to create a mapping from LINQ to OPC UA event filters / queries.

For the design of the DSL we tried to keep as close as possible to the semantics of the original LINQ while achieving a complete coverage of features provided by OPC UA. Our approach assumes that there is are node classes available for the part of the address space for which queries are specified. These node classes can easily be generated using the tools available from the OPC Foundation [4], CommServer [2] or Unified Automation [3]. The generator takes a XML based description of the address space as input and generates the appropriate classes for the Object Types, Reference Types, etc. Alternatively the classes can be developed manually. For a manual implementation it is important that there is a clear mapping between the developed classes and the NodeIds defined in the address space. Based on the generated or manually developed classes and the corresponding mapping, we translate the classes used in the LINQ expressions to NodeIds. The NodeIds are then used in the event filter and query elements that are then passed to the OPC UA stack.

The architecture of our mapping approach is depicted in Figure 4. The (generated) address space classes serve as in input data source for the LINQ processor. Based on these classes it is possible to formulate queries in an SQL-like manner. At compile time, the LINQ expression parser creates a LINQ expression tree [9] from this query code. This tree can be accessed at runtime. From this runtime representation the LINQToOPCUA generator instantiates the appropriate classes from OPC UA which resemble the event filter or query.

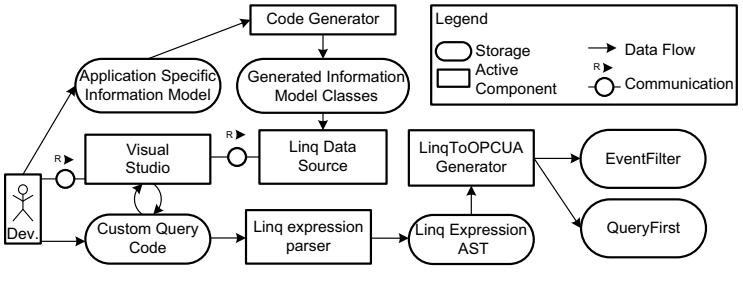


Fig. 4. Architecture of the LINQ for OPCUA prototype

Event filters and queries reuse the same elements such as ContentFilters and SimpleAttributeOperands as illustrated in Figure 3. However, the structure of a query differs from the one of an event filter. Still we were able to use the same syntax based on LINQ for both use cases as shown in the following subsections.

3.1 Event Filter

As shown in Figure 3 an event filter consists of two major parts, *where* clauses and *select* clauses. The former are *ContentFilter* elements that are common with the ones used in queries (see Section 3.3). The latter are specific to event filters

and contain *SimpleAttributeOperands*. Used in this context, the *SimpleAttributeOperands* refer to attributes of the filtered *EventType*. For example, a custom event type *DeviceStatusType* may specify, in addition to generic event attributes such as *Message* and *Source*, an additional custom attribute *HealthStatus*. These attributes may then be selected in the select clauses of an event filter to be included in the message returned by the subscription.

We map these OPC UA select clauses to the select clause available in LINQ. In this select clause we construct an anonymous type (using *new* { ... }). The members of this type can now be formed by properties from the event type class. This class is given by an initial, typed result list that is specified in the *from .. in ..* part of the LINQ query. Listing 2 gives an example for such a query. The basic result list *query* specifies to filter for the *DeviceStatusType*. Based on the usage of that class in *from cEvent in query* the *select* part references the attributes *Message*, *Source* and *HealthStatus* of *DeviceStatusType*. The *where* clause part is handled as described in Section 3.3.

Listing 2. LinqToOpcua example showing the creation of an event filter. This statement corresponds to the about 30 lines of code shown in Listing 1 without using the DSL.

```
var query = new List<DeviceStatus> { };
Expression<Func<IEnumerable<object>>> linqExp = () =>
from cEvent in query where cEvent.HealthStatus > 100
  select new { cEvent.Message, cEvent.Source, cEvent.HealthStatus };
EventFilter eventFilter = getEventFilter(linqExp);
```

The LINQToOPCUA generator uses the LINQ expression tree as input. Using a tree walker approach it traverses the expression tree and instantiates the appropriate OPC UA objects. The objects for the *select* clause are created when traversing a *MethodCallExpression* referring to the *Select* method. Then, for each *MemberExpression* within this clause it instantiates a *SimpleAttributeOperand*. The *NodeId* for the operand is looked up based on the class used for the expression's source. For the example given in Listing 2 this is the *DeviceStatusType* as this is the type of the *cEvent* variable used in the select clause.

3.2 Query

In contrast to event filters the query service (*QueryFirst*) does not use *SimpleAttributeOperands* as select clause. It rather uses so called *QueryDataDescriptions* that are connected to *NodeTypeDescriptions*. The latter is comparable to the *from* clause of SQL. Therefore, a natural match to map this to LINQ is the corresponding *from ... in ...* construct in LINQ. Thus, for each of these constructs one entry in the *nodeTypes* collection is added.

Listing 3 shows an example OPC UA query using LINQ based on the information model described in Figure 1. The *NodeTypeDescriptions* derived from this query are: The type of *controllers*, which is *ControllerType*, the type of the *Controls* reference, which is *FieldBusType* and the type of *ConnectedTo* which is of

type *DeviceType*. We defaulted the *includingSubtypes* attribute of the *NodeTypeDescriptions* to *true* as we considered this the major use case. For specifying the omission of subtypes we use the *NoSubTypes()* method as shown with *controllers* list.

Listing 3. LinqToOpcua example showing the creation of a query

```
/// Example 3: Get ControllerType.SerialNumber, FieldBusType.TypeName,
/// DeviceType.SerialNumber where a controller controls a field bus and the field bus
/// has a connected device and the field bus has a TypeName = 'PB' or 'FF' and
/// the temperature sensor's engineering unit is ''Kelvin'' and the temperature
/// is greater than 256.
var controllers = new List<Controller> { };

Expression<Func<IEnumerable<object>>> linqExp = () =>
from controller in NoSubTypes(controllers)
from fieldbus in (controller.Controls)
from devices in fieldbus.ConnectedTo
where (fieldbus.TypeName == "PB" && fieldbus.TypeName == "FF") &&
    device is TemparatureSensor &&
    ((TemparatureSensor)device).Temparature.EngineeringUnit = "Kelvin" &&
    ((TemparatureSensor)device).Temparature.Value > 256
select new { controller.SerialNumber, FieldBusType.TypeName, DeviceType.SerialNumber };
```

The elements to be returned from a query are specified by the *dataToReturn* reference pointing to a list of *QueryDataDescriptions*. We map this list to the *select* clause of the LINQ query. Its *relativePath* is directly derived from the C# *MemberExpressions*. In the query example (Listing 3) these are *ControllerType:SerialNumber*, *FieldBusType:TypeName* and *DeviceType:SerialNumber*.

Note that we default the *attributeId* to *value* as we consider this is the most frequent use case. If other attributes shall be returned by the query (like *NodeId, EventNotifier, etc.*) we provide additional methods for specifying this (e.g., *NodeId(controller.SerialNumber)*). The *filter* clause of a query is handled the same way as the *where* clause in event filters.

3.3 Content Filter

We map *ContentFilters* to *where* clauses in LINQ. In the LINQ expression tree they occur as *MethodCallExpression ("Where")* expressions. *ContentFilter* elements consist of two types of elements, one *FilterOperator* and one or more *FilterOperands*. Most of the operands are also available in C# in a similar way. Therefore, we define a mapping from these operators to the OPC UA operators as shown in Table 1. For some of the operators there is no suitable counterpart in C#. For these operators we provide methods taking the appropriate parameters (e.g., `InView()`).

The operands of a *ContentFilter* a defined by the operands of the respective C# expression. We create *LiteralOperand* for literal expressions like ("FF" or 256) and *SimpleAttributeOperands* for *MemberExpressions* respectively. The *nodeId* of the involved elements available from the (generated) node classes and can therefore easily be extracted from the involved objects during translation.

Table 1. Operator Mapping from LINQ to OPC UA

LINQ Construct	LINQ syntax	OPC UA Event Filter Construct
ExpressionType.AndAlso	`&&`	FilterOperator.And
ExpressionType.OrElse	`||`	FilterOperator.Or
ExpressionType.Or	`|`	FilterOperator.BitwiseOr
ExpressionType.And	`&`	FilterOperator.BitwiseAnd
ExpressionType.Convert	`(Type)obj`	FilterOperator.Cast
ExpressionType.Equal	`==`	FilterOperator.Equals
ExpressionType.GreaterThan	`>`	FilterOperator.GreaterThan
ExpressionType.GreaterThanOrEqual	`>=`	FilterOperator.GreaterThanOrEqual
ExpressionType.LessThan	`<`	FilterOperator.LessThan
ExpressionType.LessThanOrEqual	`<=`	FilterOperator.LessThanOrEqual
MethodCallExpression ("Like")	`Like(op1,op2)`	FilterOperator.Like
ExpressionType.Not	`~/!`	FilterOperator.Not
ExpressionType.TypeIs	`is`	FilterOperator.OfType
MethodCallExpression ("Between")	`Between(op1,op2,op3)`	FilterOperator.Between
MethodCallExpression ("InList")	`InList(op1,op2[])`	FilterOperator.InList
MethodCallExpression ("InView")	`InView(op1,view)`	FilterOperator.InView
ExpressionType.NotEqual	`!= null`	FilterOperator.IsNull

3.4 FilterOperator.RelatedTo

A special operator not defined in the mapping given in Table 1 is the *RelatedTo* operator. The OPC UA specification [8] defined this operator as follows:

> TRUE if the target Node is of type Operand[0] and is related to a NodeId of the type defined in Operand[1] by the Reference type defined in Operand [2]. Operand[0] or Operand[1] can also point to an element Reference where the referred to element is another RelatedTo operator. This allows chaining of relationships (e.g. A is related to B is related to C). [...]

This concept is similar to *join* operations in SQL. However, LINQ queries can already be based on a *connected* object model where direct references between objects are modeled as properties of the respective classes. Therefore, a join operation can be specified in a much more concise manner. For example, joining two sets of objects *ControllerType* and *FieldBusType* results in the following LINQ expression: *from controller in controllers from fieldbus in controller.Controls*. This will result in a join of controllers with their related fieldbusses.

Instead of explicitly writing all *RelatedTo* filters required for an OPC UA query we derive them from the *from* clauses of the LINQ query automatically. For example, given the *from* clause specified in Listing 3 we can derive the following nested *RelatedTo* filter: *RelatedTo(ControllerType, RelatedTo(FieldBusType, DeviceType, ConnectedTo), Controls)*. As the *form* clause then is used for the specification of *NodeTypeDescriptions* as well as *RelatedTo* clauses we get a concise, easy to read LINQ query including both parts.

OPC UA allows to filter for specific sub types within *RelatedTo* filters. For example, a filter might specify *RelatedTo(FieldBusType, TemperatureSensorType, ConnectedTo)* where *TemperatureSensorType* is a subtype of *DeviceType*. To achieve this kind of filtering an additional expression in the *where* clause is required. For the mentioned example, we would need to add the following: *device is TemperatureSensorType*.

4 Discussion

In order to evaluate our DSL we analyzed it w.r.t. its applicability and usability. We defined the following research questions to help us with this assessment.

Is the DSL development approach powerful enough to enable the development of a complex DSL? Being a query language, LINQ can be mapped to other languages the like. OPC UA event filters and queries are a perfect match for this mapping. DSLs for other purposes would much likely be developed using a different concrete syntax.

How complex is the creation of the DSL using the approach? Through the LINQ expression trees which are available directly from the LINQ statement via reflection, a DSL can be created by implementing a visitor pattern which can be created with little effort. The effort for implementing and testing the visitor was about 3 days.

Can developers effectively use the created DSL? As the syntax of LINQ is pretty close to SQL which is a well-known query language, most developers will be able to immediately use the DSL without too much learning overhead. Especially due to code completion and static type checking in LINQ queries, developers can quickly learn the usage of the DSL compared approaches using a purely string based query language. We evaluated our DSL by going through all query examples described in the specification [8] and developing the corresponding LINQ queries. As a result all of the examples could be expressed with our language. OPC UA allows to have its information models extended by new properties and references not only on type level but also on instance level. Therefore, there may be references for which no code was generated from the information model, yet. To use our DSL also in these cases we provide helper methods that generically access references and properties of within an address space as shown in Listing 4. Of course, some big advantages, i.e., the type safety and code completion features of our DSL are not available in this scenario.

Is the development with the DSL more efficient than creating lower level code? Even though an additional component, i.e., the LINQtoOPCUA generator needs to be maintained, efficiency should be higher than developing queries in lower level code. Especially, as the reduction in amount of lines of code is about an order of magnitude. The queries described as reference in the specification [8] could be expressed using LINQ with 3 to 10 lines of code.

Does it make sense to have this DSL for OPC UA? The OPC UA specification already proposes to use some kind of DSL for this use case. Using the LINQ-based solution provides a lightweight solution for it. Being an internal DSL that is well integrated into the IDE, consisting only of a few additional mapping classes, the effort for creating and maintain this DSL amortizes quickly. Thus, projects using the Event Filter or Query mechanisms benefit from the use of this DSL.

Listing 4. LinqToOpcua generic access

```
from controller in controllers
from fieldbus in NavigateReference(Controller, "Controls")
from device in NavigateReference(fieldbus, "ConnectedTo")
select new {ControllerSN = GetProperty(controller, "SerialNumber"),
      FieldbusName = GetProperty(fieldbus, "TypeName"),
      DeviceSN = GetProperty(device, "SerialNumber") };
```

5 Conclusions and Future Work

In this paper, we presented a DSL for developing OPC UA event filters and queries based on (generated) information model classes and LINQ. We facilitate the existence of information model classes which exist for OPC UA object types, their properties and references to base LINQ queries on them. Having the information model accessible on code level allows for type safe definition of LINQ queries. Using a such queries and the corresponding information model as input we can instantiate the appropriate classes given by OPC UA SDKs. Furthermore, as LINQ is a part of the standard .Net programming model many developers are already familiar to its syntax. Having a defined mapping to OPC UA allows them to reuse their knowledge in the automation domain.

Future work will deal with introducing the LINQToOPCUA DSL in current OPC UA development projects within ABB. Based on the experience gained in these projects we will be able to improve the DSL and assess its usability and impact on developer efficiency.

References

1. OPC Foundation: OPC UA Specification: Part 3 - Address Space Model (2010), http://opcfoundation.org/UA/Part3
2. CommServer: OPC UA Address Space Model Designer (2011), http://www.commsvr.com
3. Unified Automation GmbH: UaModeler (2011), http://www.unified-automation.com
4. OPC Foundation: OPC UA SDK 1.01 (2011), http://www.opcfoundation.org
5. Goldschmidt, T., Mahnke, W.: Evaluating domain-specific languages for the development of OPC UA based applications. In: 7th Vienna International Conference on Mathematical Modelling (MATHMOD)Special Session Modelling and Model Transformation in Automation Technologies (2012)
6. Fowler, M.: Domain-Specific Languages. Addison-Wesley Professional (2010)
7. Marguerie, F., Eichert, S., Wooley, J.: LINQ in action. Manning Publications Co., Greenwich (2008)
8. OPC Foundation: OPC UA Specification: Part 4 - Services (2010), http://opcfoundation.org/UA/Part4
9. Torgersen, M.: Querying in C#: how language integrated query (LINQ) works. In: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion, OOPSLA 2007, pp. 852–853. ACM, New York (2007)

Combining UML Sequence and State Machine Diagrams for Data-Flow Based Integration Testing

Lionel Briand[1], Yvan Labiche[2], and Yanhua Liu[2]

[1] Centre for Security, Reliability, and Trust (SnT), University of Luxembourg, Luxembourg
lionel.briand@uni.lu
[2] Carleton University, SQUALL Lab, 1125 Colonel By Drive, Ottawa, ON, K1S5B6, Canada
{labiche,yliu}@sce.carleton.ca

Abstract. UML interaction diagrams are used during integration testing. However, this will typically not find all integration faults as some incorrect behaviors are only exhibited in certain states of the collaborating classes during interactions. State machine diagrams are typically used to model the behavior of state-dependent objects. This paper presents a technique to enhance interaction testing by accounting for state-based behavior as well as data-flow information. UML sequence and state machine diagrams are combined into a control-flow graph to then generate integration test cases, adapting well-known coupling-based, data-flow testing criteria. In order to assess our technique, we developed a prototype tool and applied it on a small case study. The results suggest that the proposed technique is more cost-effective than the most closely related approach reported in the literature, which only relies on control flow analysis.

Keywords: UML 2, Interaction diagram, State machine, Data flow, Coupling, Integration testing.

1 Introduction

In an object-oriented system, objects collaborate to provide functionalities. Even when classes have been unit tested thoroughly, unexpected failures may arise when they collaborate, leading to the identification of integration faults. Class integration testing focuses on class interactions to ensure functional correctness.

The Unified Modeling Language (UML) has become the de-facto standard for analysis and design of object-oriented software systems [1]. A number of papers (e.g., [2-7]) have proposed test case generation strategies from different UML design artifacts: interaction diagrams (sequence or communication diagrams) have been used to test class integration [2], interaction diagrams along with state machine diagrams have been used for state-based integration testing [3, 5, 7], and state machine diagrams have been used to perform unit testing [4].

The interactions among different instances can be specified using UML sequence diagrams, which are therefore suitable diagrams for integration testing [2]. However, testing interactions among classes based solely on those diagrams is not enough to find all integration faults as some incorrect behaviours are only exhibited in certain

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 74–89, 2012.

states of the collaborating objects during an interaction. This has somewhat been confirmed [8] since experimental results show that test suites derived from sequence diagrams and test suites derived from state machines find complementary sets of faults. Others showed that combining sequence and state information leads to detecting faults in the implementation of guard conditions and that such faults would not necessarily be found by solely using sequence diagrams [7]. Thus, one of our objectives is to generate class integration test cases from a combination of sequence and state machine diagrams, as suggested by others [9], so as to fully exercise the state-based behavior of interacting objects to uncover state-dependent interaction faults.

Note that a system under test may have several sequence diagrams to model its use cases. Our approach will not take all sequence diagrams as inputs. Instead, we only focus on one use case / sequence diagram at a time (e.g., one could start with the most critical ones). Additionally, our approach relies on model elements such as messages, guards, triggers, and actions that have remained unchanged in all versions of UML 2 to date. Unless necessary, we therefore do not indicate any specific version of UML 2 and simply refer to UML in the remainder of the paper.

Adequacy criteria are used to avoid exhaustive testing, which is often impractical (if even feasible), while gaining sufficient confidence in the system under test. One of our objectives is to offer a set of adequacy criteria for our test model built by combining sequence and state machine diagrams. Previous research, both theoretical and empirical, has revealed that data- and control-flow strategies may be complementary. Since previous test case generation strategies, comparable to ours, rely on control-flow criteria, we define data-flow criteria for our test model.

The rest of the paper is organized as follows. Section 2 discusses related work. We then propose a comprehensive methodology to combine UML sequence and state machine diagrams in one test model to conduct coupling-based, data-flow analysis. A set of mapping rules from a UML sequence diagram and state machine diagrams to the test model are formalized by using OCL (Section 3): rules match messages in sequence diagrams to state machine transitions, integrating state machine information into a control flow graph derived from a sequence diagram. Data flow information is analyzed based on the input models, operation signatures and operation contracts, and coupling data-flow criteria are applied to derive test cases (Section 4). A prototype tool has been developed to semi-automate the methodology. A case study is reported in section 5, where we study the cost-effectiveness of our approach and compare it to a previously published one. Conclusions and future work are outlined in section 0.

2 Related Work

There is a plethora of testing techniques based on one or more of the most used UML diagrams: class, activity, sequence, state machine diagrams. For instance, a quick search (conducted in March 2011) on Inspec and Engineering Village databases for papers published between 2007 and 2011 with keywords “testing”, “UML”, and “sequence diagram” (to be searched in titles, abstracts and keywords) resulted in a list of more than 36 unique publications. It is not our intent here to discuss them all, or even list them all, since most of them do not relate closely to our objective: How can we account for the fact that messages in sequence diagrams can be received by objects in

different states? Furthermore, none of these papers attempts to apply data-flow criteria. We rather focus below on the few published approaches that attempt to combine sequence and state machine diagrams [3, 7, 10-16].

One approach is to combine the class, sequence and state machine diagrams to create a test model, a form of control flow graph with data-flow information [7]. Data-flow information pertains to variable assignments from sequence diagrams (i.e., arguments passed to messages, return values used to set variables) and post-conditions of operations triggered by messages. In the combination process, a sequence diagram becomes a control flow graph, that is extended thanks to state machine diagrams (the ones of the classes whose instances are used in the sequence diagram) as follows: if a message in a sequence diagram fires at least one transition in a state machine, the extended control flow graph contains as many nodes/messages as transitions that can potentially be fired, thereby specifying the alternative behaviours that can be triggered. Only control flow criteria (i.e., node, edge, path coverage) are then used to derive test sequences: the data flow information is not used for test case selection; data flow information is used to identify test inputs. A similar combination procedure is described by Ali et al. [3] though from UML 1.x statecharts and collaboration diagrams. No data-flow criterion is used to select test cases. Other similar combinations have been proposed [12, 13], sometimes extending Ali et. al. procedure. Again, only control flow criteria are used to select test cases. Instead of sequence diagrams, some authors combine statecharts and activity diagrams (e.g., [14, 15]). The combination procedures are similar to the ones previously mentioned, and only control flow criteria are used. Sokenou suggests that for each sequence of messages identified in a sequence diagram, an initialization sequence be identified from statecharts [16]. This requires, similarly to the approaches discussed so far, that we identify in which states messages in sequence diagrams can be triggered and received.

The main differences between these works and our work are twofold: we create a different control flow graph test model, and use data-flow test criteria, which they do not. With respect to the latter difference, although previous test models sometimes contain data-flow information [7] (though to a lesser extent as they do not identify uses and definitions of variables for instance), data-flow information is not used to build test sequences (i.e., using data-flow adequacy criteria). Rather it is used to help find test inputs for test sequences derived from control-flow criteria. With respect to the construction of the test models, our approach is different in one or more of the following ways: we support the UML 2 notation to a larger extent (e.g., not all previous approaches support asynchronous messages or “par” combined fragments); we believe our solution is more flexible since we make fewer assumptions with respect to the consistency between the sequence and the state machine diagrams (e.g., some previous works assume that the sequence of messages received by a lifeline is a legal sequence of transitions in the state machine describing the behaviour of the lifeline’s object, whereas we acknowledge their could be some inconsistencies between the two depending on the level of details the designer put in the diagrams); our test model accurately represents nested calls (it is not possible in previous test models to identify which message triggers which other messages—calls are flattened, similarly to [17]). This latter difference about the test model is very important since without information on nested calls, it is impossible to apply criteria specifically targeting interactions between callers and callees.

Another related work [10] relies on AUML sequence and state machine diagrams—AUML is at the same time an extension of UML for specifying agents and testing interactions between them, and a subset of the UML notation. The authors transform those diagrams into a Maude (formal) model, which is used as test model. There is, however, no combination of the sequence and state machine diagrams since the Maude's rewriting rules are only derived from the state machines describing the behaviour of the communicating agents.

Sequence diagrams are also used as test objectives to trigger sequences of transitions in state machines [6, 11, 18-20]. In essence, the authors rely on existing, user-defined (control-flow) test objectives specified as sequence diagrams whereas we intend to (semi-)automatically identify (data-flow) test objectives.

Other related work extends the information provided in sequence diagrams with possible polymorphic messages, i.e., messages that can potentially trigger polymorphic calls [21], instead of state information. This work is complementary to ours and we will study the possibility of combining both approaches in our future work.

Earlier most cited works in the domain include test case generation from UML 1.x collaboration diagrams [2], from communicating finite state machines [5], from communicating UML 1.x statecharts [22]. Our work differs as we combine sequence and state machine diagrams into one test model and use data-flow testing criteria.

3 Message/Event/Action Control Flow Graph (MEACFG)

We transform a sequence diagram and state machine diagrams (of the classes involved in the sequence diagram) into a control flow graph—our test model—which we model as an activity diagram. We describe the construction of our test model in two steps, for illustration purposes only (i.e., our tool generates the test model in one step): we represent the control flow relationship among the messages of the UML sequence diagram (section 3.1) and add state information from state machine diagrams, matching messages in sequence diagrams to state machine transitions (section 3.2). The figures in this section and in section 4 illustrate different, un-related abstract examples, instead of one running example, as this allows us to present the main aspects of our approach in a condensed way: a (real) running example illustrating the same aspects would involve much larger diagrams that would not fit in a conference paper. Section 4 describes the use of the test model to generate test cases.

3.1 From a UML Sequence Diagram to a Control Flow Graph

A message control flow graph (MCFG) represents a UML sequence diagram using the UML activity diagram notation. Executable nodes in the MCFG for a sequence diagram correspond to messages in the sequence diagram, while control nodes show the sequence of execution of messages and object nodes show data-flow. In the following, UML metaclasses are written in courier font.

MCFG nodes are of three types: control nodes, executable nodes, and object nodes. A control node can be one of the following: initial node, final node, fork node, join node, decision node, and merge node. Each MCFG has a single initial node and final

node. An executable node in the MCFG is an `Action` node, which may be a `StructuredActivityNode` or a `CallOperationAction` (specializations of `Action`), corresponding to a message in the sequence diagram that either triggers other messages or not, respectively. A structured activity node contains several executable nodes, control nodes, and object nodes, which indicates a call hierarchy among messages in the sequence diagram. Each structured activity node has its own initial and final nodes. `CallOperationAction`s and `StructuredActivityNode`s can have pre- and post-conditions. An `ObjectNode` is used in an `Activity` to indicate object flow. In the MCFG, object nodes are used to specify input and output parameters for each operation shown in the sequence diagram and show object flow between actions (class `Pin`). An `ActivityPartition` records the target of a message/call whereas we can get the source of a message/call from its nesting node in the MCFG. If a node corresponding to a message is nested in a structured activity node, then the structured activity node is the source of the message.

We assume the `ExecutionSpecification` (a.k.a. activation bar in UML 1.x) is specified on the lifeline of the sequence diagram. This is to ensure that we can unambiguously identify messages that trigger each other. For instance, Fig. 1 (a) does not have execution specifications and is ambiguous as it can correspond to either figure (b) or (c): in figure (a), one does not know whether message `m4` is invoked by `m1` or `m3`. Since some CASE tools, such as IBM RSA, support `ExecutionSpecification`, we consider this a reasonable assumption.

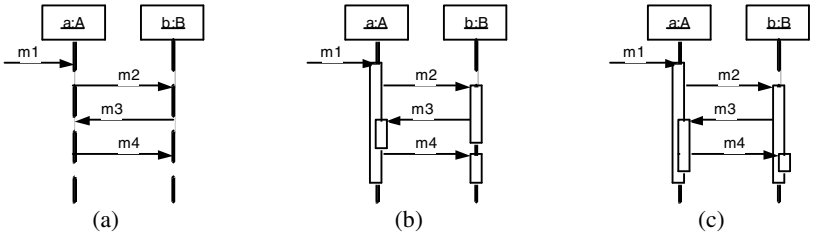


Fig. 1. Usefulness of execution specifications

The construction of a MCFG for a sequence diagram is detailed below where we show how each important UML sequence diagram construct is transformed into a MCFG construct. To facilitate the discussion, we use the example of Fig. 2: sequence diagram (left) and corresponding MCFG (right); where node `m1` (MCFG) denotes message `m1` (sequence diagram), pre- and post-conditions of the operation invoked by message `m1` are constraints associated with node `m1`, and `g1` is a guard condition. Stereotypes conform to UML notations [23]: a `StructuredActivityNode` has a `<<Structured>>` stereotype; the pre- and post-conditions of a node (obtained from the class diagram) have `<<localPrecondition>>` and `<<local-Postcondition>>` stereotypes; nodes inside a `LoopNode` have `<<LoopSetup>>`, `<<LoopTest>>`, and `<<LoopBody>>` stereotypes for each section of a `LoopNode`, respectively. Additional examples illustrating the transformation can be found in [24].

Each **synchronous message** that does not trigger any other message is transformed into a `CallOperationAction`. A synchronous message that triggers other messages is transformed into a `StructuredActivityNode`, containing activity nodes corresponding to the messages it triggers. In Fig. 2, `m1` invokes `m2` and therefore `m1` is transformed into a `StructuredActivityNode` that contains an `ActivityNode` representing `m2`: more specifically `m1` contains a loop since `m2` appears in a loop combined fragment (loops are discussed below). Structured activity node `m1` has two input pins, for the two parameters of message `m1` in the sequence diagram.

UML defines different message sorts, two basic forms of which are operation calls and signals. If a message is an operation call, the pre- and post-condition of the message are those of the operation. If a message is a signal, i.e., the specification of an asynchronous communication between objects, it is realized by an operation, usually called the signal handler. There are two ways to specify the handler of a signal. One is to declare an operation with stereotype `<<signal>>`, which has the same name as the signal, in the class or interface to indicate the receipt of the signal. In this case, the pre- and post-conditions of a signal are those of the signal handler operation. Another way is to use a signal as a trigger of a transition in the receiver's state machine. The actions on the transition are the handlers of the signal. In this case, the signal is just a trigger to invoke the handlers and its pre- and post-conditions are empty.

If a message does not have a sender lifeline or is sent by an actor, we specify a node contained by the activity corresponding to that sequence diagram. If a message is sent several times to the same object, we create several nodes in the MCFG.

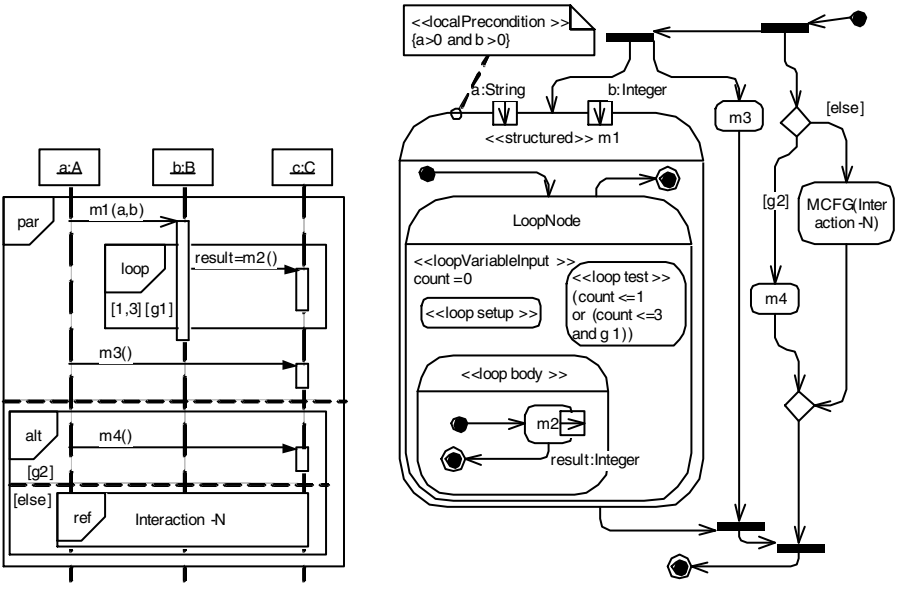


Fig. 2. A sequence diagram and its corresponding MCFG

`par` combined fragments and **asynchronous messages** denote concurrent executions, which we specify with fork and join nodes. In Fig. 2, the sequence diagram is made of a `par` combined fragment (to divide the sequence of messages `m1` and `m3` from the alternative combined fragment containing `m4`). Additionally, in the first part of the `par` combined fragment, message `m1` is asynchronous. This results in two fork nodes and two join nodes in the activity diagram. It is important to note that, referring to Fig. 2, although the sequence diagram does not specify when the asynchronous call `m3` will eventually finish (and return), the corresponding MCFG does indicate that `m3` will not finish after `m1`. The behaviour specified in the MCFG therefore does not necessarily correspond to what was initially intended in the sequence diagram. We made this decision anyway since we needed to specify a behaviour that is possibly coherent with the sequence diagram. Plus, without additional information about `m1` and `m3` (e.g., expected execution times) it is not possible to specify exactly the same behaviour as in the sequence diagram: we do not know where to place the merge node for the asynchronous message. As a result, some data flow information may not be accurate: either false positive or false negative. Only additional case studies will tell us the extent of the impact of our decision. Future work could look into using the MARTE profile when specifying the sequence diagram to obtain a more consistent MCFG.

A **`loop` combined fragment** is specified with a `LoopNode`, and its setup, test, and body sections [1, 23], specified inside the structured activity node with different stereotypes (Fig. 2). The loop body is a structured activity node containing nodes for the messages inside the `loop` combined fragment. Nothing in a loop combined fragment really corresponds to a loop setup, which is therefore empty. The test section is a Boolean expression evaluated before or after the body section, depending on the value of attribute `isTestedFirst` of the loop node object [23]. Since a `loop` combined fragment is a 'while' loop, attribute `isTestFirst` is set to true for each loop node in the MCFG. In a `loop` combined fragment, the guard may include a lower and an upper number of iterations as well as a Boolean expression. After the minimum number of iterations has executed, if either the Boolean expression is false or the maximum number of iterations is reached[1], then the loop terminates [23]. In Fig. 2, `[1, 3]` denotes the minimum and maximum number of iterations, and `[g1]` indicates the Boolean expression to be satisfied. To model such a complex condition in the MCFG, we add a variable, named `count` with stereotype `<<loopVariableInput>>` [1], to the loop node to count the number of iterations (**Fig. 2**). Although not shown in **Fig. 2**, all paths inside the body section of the loop node finish with an activity node incrementing variable `count`. Then, we can write the test section of the loop node as follows[2]:

```
count <= minimum or
(count <= maximum and Boolean_expression_in_loop_combined_fragment).
```

An **`alt` combined fragment** denotes alternative flows of messages, each flow being specified in an interaction operand. This is rendered in our MCFG with a decision

[1] If there is only one number, the loop executes a fixed number of times. If there is no minimum and maximum numbers, the loop lower and upper bounds are considered to be 0 and infinity, and the Boolean expression solely determines the number of iterations.

[2] This general form can be modified if the loop combined fragment has only a min and max number of iterations, only a fixed number of iterations, or only a Boolean expression.

node with as many outgoing edges as interaction operands, and a merge node for merging the different flows. Each outgoing edge of the first decision node has a guard, the one of the corresponding interaction operand. An **`opt` combined fragment** is a specific case that specifies only one interaction operand.

A **`break` combined fragment** specifies a behaviour triggered only under a specific condition and otherwise skipped. When the condition is true, and the corresponding behaviour finishes, the flow jumps to the end of the enclosing interaction fragment (some behaviour of the enclosing interaction fragment is skipped). So a `break` combined fragment is represented as a decision node with two outgoing edges: one skipping the conditional behaviour and one flowing to the conditional behaviour. The flow after the conditional behaviour does not merge the skipping flow (as in an `opt`). Instead, it merges a control node that corresponds to the end of the enclosing interaction fragment. This can be one of the following: (1) a decision (merge) node if the enclosing interaction fragment is an `alt` or an `opt` combined fragment; (2) a join node if the enclosing interaction fragment is a `par` combined fragment; (3) a node right after a loop node, if the enclosing interaction fragment is a `loop` combined fragment; (4) the final node of the activity diagram, if none of the above applies.

An **`InteractionUse` interaction fragment** refers to an interaction, i.e., another sequence diagram. This is modeled in our MCFG as a `CallBehaviorAction` named after the `InteractionUse` name (Fig. 2). This `CallBehaviorAction` can have input and output pins if the `InteractionUse` has actual parameters.

There are other types of combined fragments, namely `critical`, `neg`, `assert`, `strict`, `seq`, `ignore` and `consider`. These are considered to be less used by modelers and we do not account for them. Future work will look into that issue.

Messages in a sequence diagram may have parameters and return values. This is modeled in our MCFG as `InputPin`s and `OutputPin`s of nodes (either `CallOperationAction` or `StructuredActivityNode`) corresponding to messages. For an `in` parameter, an `InputPin` is added to the node, with appropriate type obtained from the class diagram. For an `out` parameter or a return value, an `OutputPin` is added to the node. If a parameter is an `inout` parameter, we add both an `InputPin` and an `OutputPin`. These pins are useful for identifying definitions and uses of variables in the MCFG. In Fig. 2, parameters `a` and `b` of message `m1` are modeled as two input pins with corresponding types, shown at the boundary of the structured activity node `m1`. Similarly, the return value of message `m2` is modeled as an output pin of node `m2`, which indicates that `m2` delivers a value back to `m1`.

The MCFG also contains notes describing pre- and post-conditions of operations, obtained from the class diagram: one example is illustrated in Fig. 2 for node `m1`.

A sequence diagram may show recursive calls, which are not handled in our MCFG generation. However, we consider that the behaviour models we are dealing with are not at the level of detail where recursion would appear. Indeed, we consider recursion to be a low-level design decision (algorithm) whereas we are using analysis and high-level design models as input.

3.2 Adding State Information

A message in a sequence diagram can trigger transitions in some state machines. Adding state information to the MCFG is to identify, as accurately as possible, the transitions in state machines that may be fired by messages.

In UML one can add a `StateInvariant` on a `Lifeline` to specify conditions that hold before or after a message or sequence of messages. If such information is available, we can precisely identify which transition(s) in the state machine of the target object of the message(s) is actually triggered by the message. If the actions resulting from firing a transition are not shown in the sequence diagram (i.e., the message firing the transition should trigger messages corresponding to actions), this information can be added to the MCFG (i.e., nested node). The obtained MEACFG combines the behaviours specified in a sequence diagram and state machines.

If state invariants are not specified on lifelines, we proceed as follows. For each MCFG node `n`, that corresponds to message `m`, if the behaviour of the receiver of `m` is not specified by a state machine, then we do not modify the MCFG. Should the opposite occur, we look at the state machine, and identify transitions that `m` triggers. (Note that we only support explicit triggers and do not handle completion events in the state machine.) Then, we add to the MCFG the behaviour(s) specified in the state machine when `m` triggers those transitions. Three different situations can occur:

Case 1: Node `n` is a `CallOperationAction` instance, i.e., `m` does not invoke any interclass message. (We assume that intra-class messages may have been omitted to simplify the sequence diagram, but that interclass messages should always be specified.) The transitions fired by `m` should not have any (interclass) operations or signals as actions (consistency between the diagrams). However, it is possible that the transitions have intra-class operations or signals as actions. What the message is doing to the object may be specified in the state machine but not in the sequence diagram. Therefore, the transitions that `m` can trigger are those without any action or those with intra-class operations or signals as actions.

Case 2: Node `n` is a `StructuredActivityNode` instance (except loop nodes), i.e., `m` invokes other messages, and the messages triggered by `m` should match some sequences of transition actions in the state machine of the recipient of `m` (consistency between the diagrams). The transitions that `m` can fire are those with actions matching the messages triggered by `m`.

Case 3: No matching is possible. In this case, we consider there is an inconsistency between the diagrams, which has to be resolved by the user.

Fig. 3 illustrates cases 1 and 2 for the same state machine (left). In Fig. (a), MCFG node `m1` is an action node representing message `m1` in the sequence diagram. Since `m1` doesn't invoke any inter-class message, both transitions `t1` (no resultant action) and `t2` (the resultant action is an intra-class operation) in the state machine correspond to the sequence diagram (assuming action `a1()` is an intra-class action, whereas `m2()` and `m3()` are not). In Fig. (b), `m1` is a structured activity node corresponding to message `m1` in the sequence diagram: `m1` invokes messages `m2` and `m3`. Therefore, transition `t3` in the state machine is the behaviour triggered by `m1` in this sequence diagram as messages invoked by `m1` match the sequence of actions (`m2` and `m3`) on `t3`.

Once we have identified transition(s) matching a message `m`, state, guard condition, and additional action information on the transitions are added to the MCFG, producing the MEACFG. If there is only one identified transition, the source state and target state of the transition are transformed into Boolean expressions that are associated with the incoming and outgoing edges of the executable node `n` in the MEACFG respectively. This is the case of Fig. 3 (b), and the procedure above leads to Fig. 4 (b). The format of the Boolean expressions is `[objectName.state = stateName]` where `objectName` is derived from the lifeline of the sequence diagram, and `stateName` is obtained from the state machine diagram of the object. If a class has a `state` attribute, we use it in the Boolean expression directly. Otherwise, a state attribute is added to the class implicitly for generating test cases and deriving test oracle.

If there are actions on the transition, `m` invokes operations. Thus, `n` is a structured activity node, and contains nodes specifying invoked actions. The first action on the transition is connected to the outgoing edge of the initial node in `n`, and the outgoing edge of the last action is connected to the flow final node. Actions connect to each other according to their written order on the transition.

If there is more than one identified transition that message `m` can fire, a decision node is added to specify alternative paths resulting from several fired transitions. Each path corresponds to one of those transitions. For each transition, and therefore for each outgoing edge from the decision node, the approach described above is applied. The paths then merge into another decision node. This is illustrated in Fig. 4 (a) that is derived from Fig. 3 (a): the alternatives indicate that what happens in the MCFG node `m1` in Fig. 3 (a) depends on whether the state of the object is `S1` (`m1` is triggered as part of transition `t1` and the new state is `S2`) or `S2` (`m1` is triggered as part of transition `t2`, which triggers `a1`, and the new state is `S3`).

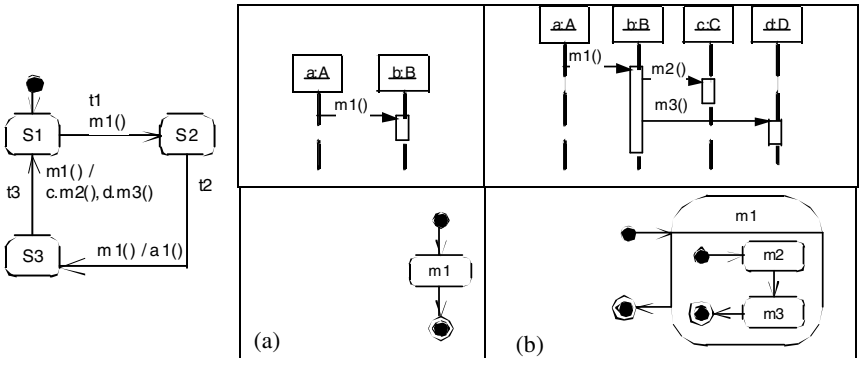
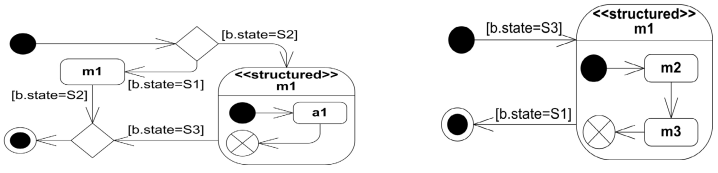


Fig. 3. Illustrating the recovery of state information

4 Coupling-Based Testing of Class Interactions

The MEACFG test model is more complete than previous attempts (i.e., a larger portion of the UML standard is accounted for), as discussed in section 2. With this test model, one can use control-flow adequacy criteria as in previous works, but also data-flow criteria as discussed below.



(a) (b)

Fig. 4. MEACFG: MCFG plus state information

Coupling-based data-flow criteria for integration testing of procedural software [25] have been adapted to class integration testing [26]. We adapt them to model-based integration testing, and use them to generate integration test cases. In a nutshell, the criteria [25, 26] are to exercise paths between the last definitions of variables before calls to and returns from the called operation, and the first uses of variables after calls to and returns from the called operation. Due to space constraints we refer the reader to [24-26] for more details on these criteria. What matters and requires a detailed discussion is how we identify definitions and uses of variables in a MEACFG, which we discuss below and illustrate with Fig. 5: (a) sequence diagram and (b) corresponding MEACFG (definitions and uses are indicated on the side of the diagram, for illustration purposes only). (The corresponding class diagram is not shown.)

A MEACFG is a control flow graph showing sequences of executions of operations (nodes) and calls between these operations (structured activity nodes). A structured activity node (e.g., node `m1`) is a caller message/operation and its nested nodes (e.g., node `m3`) are called messages/operations: each pair (`n1`, `n2`) in the MEACFG where `n1` is a structured activity node and `n2` is a node (either structured activity or not) representing a message denotes a call, `n2` being the call site, i.e., the place in the control flow indicating the call; for instance pairs `(m1, m2)`, `(m1, m3)` and `(m1, m4)` represent the three calls `m1` makes, `m2`, `m3`, and `m4` denoting call sites.

As discussed earlier, actual parameters, mapping formal ones, and return values become pins. For instance, actual parameter `pf1` of operation `m1`, maps to formal `Integer` parameter `pa1`, and becomes an input pin to structured activity node `m1`. With respect to data-flow, input pins denote uses whereas output pins denote definitions.

All the formal parameters of a caller message are specified as being defined in the initial node of the corresponding structured activity node, e.g., formal parameter `pa1` is defined in the initial node of structured activity node `m1`. Additionally, any local variable in a structured activity node is considered to be defined in the initial node of that structured activity node, e.g., `lP`, and `at` for `m1`. We consider a variable to be local when it is not an attribute or reference that can be accessed (directly or through a navigation) from the context object executing the corresponding (enclosing) operation/message. The return value of a structured activity node, if there is one, is also specified as being defined at the initial node of the structured activity node, e.g. `ret` for node `m1`. These ensure that if our analysis does not detect any definition of such variables/parameters between the initial node of the structured activity node and a call site (where one of the variables would be passed as an actual parameter) we have at least one coupling-definition involving the use of that variable at the call site. This is

a conservative heuristic to ensure that uses of variables/parameters are exercised by the testing criteria. This may however result in coupling-definitions that do not exist, e.g., if there is a definition between the initial node of the structured activity node and the call site that we cannot detect, the coupling-definition we identify with our heuristic is not the right one. Only case studies will tell us the extent of false-positives. Such a heuristic is necessary since we decided to work only from models, which are, in essence, abstractions.

This data flow information is solely retrieved from operation signatures (from the class diagram). Additional data flow information is identified from guard conditions (e.g., `alt` and `loop` combined fragments), state invariants, and operation contracts. Guard conditions such as `g1` and `g2` in Fig. 5, if expressed in OCL, can be analyzed automatically to identify the variables, formal parameters, attributes, links they manipulate, and which translate into uses on edges in the MEACFG. For the analysis of OCL expressions, we rely on a previous work [4] where the authors defined a set of rules to systematically analyze OCL expressions and identify definitions and uses of model elements (attributes, links, collections, …). Possible types of uses and definitions have been formally specified using OCL expressions, based on the MEACFG metamodel, which can be found in [24] along with illustrating examples.

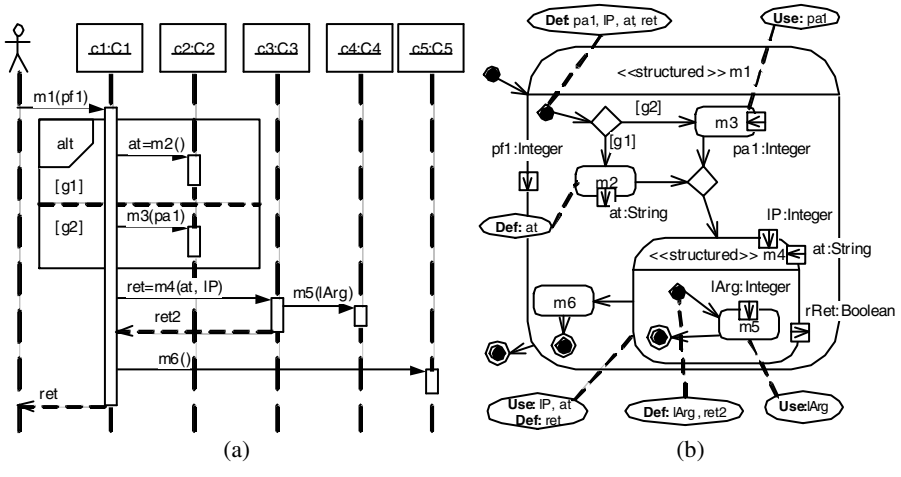


Fig. 5. Illustrating data flow in MEACFG

For the MEACFG, annotated with data-flow information as discussed above, coupling-based testing criteria [25, 26], and a graph algorithm to identify test objectives for these criteria (typically definition-clear paths) and then test paths (i.e., complete paths, from start node to end node, in the MEACFG), can be readily applied. Given such test objectives and test paths, our approach relies on manual identification of test inputs to make test paths executable. In the future we will investigate a solution to automatically identify such test inputs [7, 27]. What we presented above applies directly to the notion of parameter coupling, i.e., when two operations interact through parameters. Other types of coupling [25], e.g., shared variable coupling

(which in our context can translate into attribute coupling), can be accounted for in a similar way.

5 Case Study

To conduct our case study we implemented our approach as an Eclipse plugin using IBM Rational Software Architect (RSA). It is based on EMF (to model the MEACFG) and RSA's model-to-model transform engine (to create a MEACFG from a sequence diagram and related state machine diagrams). More details about the tool architecture are available in [24].

To validate our approach, we compared it with SCOTEM [3], as our approach is closely related to it. In doing so we want to investigate whether using data-flow criteria (our approach) improves fault detection over control-flow criteria (SCOTEM). Indeed, the SCOTEM approach supports four control-flow testing criteria to generate test paths: Single-Path, Transition, n-Path, and Path testing criteria. Another comparison that could be investigated is with [7] though, as described above, this approach covers a smaller subset of the UML notation and relies on additional assumptions about the modeling methodology. We defer this comparison to future work.

We therefore use the same case study system as the one used to evaluate SCOTEM: the Arithmetic Tutor (AT) system. To determine the effectiveness of our approach, we seeded faults into the code, using mutation operators, as has been done in numerous similar experiments and showed to be an adequate way of comparing testing techniques [28]. We used the same mutant programs as in the SCOTEM case study since (1) these mutants were carefully selected to be varied (in terms of operators and locations in the source code) and lead to interaction faults, and (2) this will allow us to precisely compare the effectiveness of the two approaches. We refer the reader to [3, 24] for more details on this aspect. Overall, 49 mutants were generated by using 12 mutation operators.

The AT case study teaches a variety of arithmetic operations to students and provides self-evaluation of learning activities. It can be run in two modes: training and assessment. In the training mode, the AT system provides complete, step-wise gradual explanations for arithmetic operations. Students can choose any type and complexity of arithmetic operations to learn. In the assessment mode, the AT system randomly generates a set of operations to evaluate the arithmetic skills of a student, and the complexity of generated problems is dynamically adjusted according to the student performance. The detailed performance for each session is logged and the history record of the student is updated at the end of each session.

The AT system that we consider, and that was used in [3], is an incomplete design and implementation, which is a simple version of the assessment mode that generates basic arithmetic operations only, such as addition, subtraction, multiplication, and division. The problems are generated randomly by the AT system (complexity of generated problems are not taken into account in our case study). The AT case study is implemented in Java, and contains nine classes (including 42 methods, 339 LOC): five of them are specified with state machines (with more than one state). The UML

sequence and state machine diagrams are available in [24], where the constructed MEACFG (with data flow information) can also be found.

Since different test suites for the different criteria could result in different mutation scores, to provide a meaningful evaluation and also facilitate the comparison with the SCOTEM approach, we adopted the SCOTEM strategy and chose 10 randomly selected adequate test suites for each testing criterion and calculated the minimum, average, and maximum mutation scores for each of them. For each criterion, the 10 adequate test suites have the same test paths (in the MEACFG). What differ are the test inputs to execute the paths.

Test suites generated for the Coupling-Defs criterion have six paths, which detected 45 to 49 mutants. The mutation score of a Coupling-Defs adequate test suite was 98% on average. Call-Sites, Coupling-Uses, and Coupling-Paths criteria generated 18, 18, and 27 test paths, respectively, and adequate test suites were all able to detect all mutants, which is probably due to the fact that the case study is simple.

A qualitative investigation of alive mutants when using Coupling-Defs showed that the mutation operators modified conditional statements. These correspond to guards in MEACFG edges, and since Coupling-Defs does not necessarily cover those edges, the corresponding adequate test suites may miss the mutants.

A comparison between SCOTEM and our approach is summarized in Table 1. The table reports on the different criteria supported by the two approaches, the number of paths (i.e., test cases) the criteria require, and the mutation score achieved (on average over 10 adequate test suites).

Table 1. SCOTEM vs. our approach

SCOTEM			MEACFG		
Criteria	Number of Test paths	Mutant scores on average	Criteria	Number of Test paths	Mutant scores on average
Single-Path	1	75%	Coupling-Defs	6	98%
Transition	3	91%	Call Sites	18	100%
n-Path (n=82)	82	95%	Coupling-Uses	18	100%
Path	162	100%	Coupling-Paths	27	100%

Single-Path and Transition criteria (SCOTEM), with one and three tests respectively, can only detect less than 91% mutants, and n-Path only kills 95% of the mutants with many more tests (82). However, Coupling-Defs (MEACFG) kills 98% mutants with six test paths only. When applying Call Sites, Coupling-Uses and Coupling-Paths criteria (MEACFG), we only need 18 to 27 test paths to kill all the mutants. However, to achieve 100% mutation score following the SCOTEM approach, one needs to execute all the 162 paths, which is much more expensive and may turn out to be impossible for a larger system. We analyzed mutants that remain alive when following the SCOTEM approach, and we noticed that they are state-related faults: an object is in a wrong state before receiving a message, or the states of the sending and receiving objects of a message contradict the specification of the message. Given that the data-flow criteria specifically focus on definition and uses of variables and some of the variables specified in the MEACFG are specifically modeling changes of states (e.g., in conditions), it is not surprising that data-flow adequate test suites have more

chances (with fewer tests) to find those faults. The results of this case study show that coupling data-flow criteria are effective for selecting test paths from the MEACFG to detect state faults. Though the case study used is admittedly simple, it nevertheless shows a significant improvement over SCOTEM, which was our objective.

6 Conclusion

Testing interactions among classes based on UML interaction diagrams may not find all integration faults as some incorrect behaviors may only be exhibited in certain states of the collaborating classes during an interaction. Additionally, data-flow testing criteria are known, in general, to complement control-flow ones. Hence, this paper has presented a comprehensive approach to conduct state-based integration testing based on coupling data-flow testing criteria. The approach consists in combining information from a UML 2 sequence diagram and state machine diagrams (of the classes whose instances are involved in the sequence diagram) and then generating a control flow graph (a UML activity diagram). It does so while supporting a large portion of the UML 2 sequence and state machine notations. This graph is annotated with data-flow information, also derived from the UML model (operation signatures, message and transition guards, operation contracts), and is used as a test model to derive test cases according to well known coupling-based, data-flow testing criteria.

The approach has been implemented in a tool and used in a case study to compare it with the closely related SCOTEM approach. Results show that accounting for state and coupling-based, data-flow information when deriving tests from sequence diagrams is more cost-effective at finding faults than simply relying on control-flow information (SCOTEM).

Several venues for future work can be identified. Among them we can mention experimenting with our approach using case study systems of varying complexities, comparing the cost and effectiveness of the different criteria, comparing it with SCOTEM, and comparing our approach with other approaches (e.g., [7]).

References

[1] Pender, T.: UML Bible. Wiley (2003)

[2] Abdurazik, A., Offutt, J.: Using UML Collaboration Diagrams for Static Checking and Test Generation. In: Evans, A., Caskurlu, B., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 383–395. Springer, Heidelberg (2000)

[3] Ali, S., Briand, L.C., Rehman, M.J., Asghar, H., Zafar, Z., Nadeem, A.: A State-based Approach to Integration Testing based on UML Models. IST 49(11-12), 1087–1106 (2007)

[4] Briand, L.C., Labiche, Y., Lin, Q.: Improving the Coverage Criteria of UML State Machines Using Data Flow Analysis. STVR 20(3), 177–207 (2010)

[5] Gallagher, L., Offutt, A.J., Cincotta, A.: Integration testing of object-oriented components using finite state machines. STVR 16(4), 215–266 (2006)

[6] Pelliccione, P., Muccini, H., Bucchiarone, A., Facchini, F.: TeStor: Deriving Test Sequences from Model-based Specifications. In: ACM CBSE, pp. 267–282 (2005)

[7] Bandyopadhyay, A., Ghosh, S.: Test input generation using UML sequence and state machines models. In: IEEE ICST, pp. 121–130 (2009)
[8] Kansomkeat, S., Offutt, J., Abdurazik, A., Baldini, A.: A comparative evaluation of tests generated from different UML diagrams. In: ACIS SNPD, pp. 867–872 (2008)
[9] Wu, Y., Chen, M.-H., Offutt, A.J.: UML-Based Integration Testing for Component-Based Software. In: Erdogmus, H., Weng, T. (eds.) ICCBSS 2003. LNCS, vol. 2580, pp. 251–260. Springer, Heidelberg (2003)
[10] Mokhati, F., Badri, M., Badri, L., Hamidane, F., Bouazdia, S.: "Automated testing sequences generation from AUML diagrams: a formal verification of agents' interaction protocols". IJAOSE 2(4), 422–448 (2008)
[11] Pickin, S., Jard, C., Jeron, T., Jezequel, J.-M., Le Traon, Y.: Test synthesis from UML models of distributed software. IEEE TSE 33(4), 252–268 (2007)
[12] Sarma, M., Mall, R.: Automatic generation of test specifications for coverage of system state transitions. IST 51(2), 418–432 (2009)
[13] Wu, C.-S., Chang, W.-C., Kim, S., Huang, C.-H.: Generating State-based Polymorphic Interaction Graph from UML Diagrams for Object Oriented Testing. In: IAENG IMECS, pp. 726–731 (2011)
[14] Barisas, D., Bareiša, E.: A Software Testing Approach Based on Behavioral UML Models. ITC 38(2), 119–124 (2009)
[15] Swain, S.K., Mohapatra, D.P., Mall, R.: Test Case Generation Based on State and Activity Models. JOT 9(5), 1–27 (2010)
[16] Sokenou, D.: Generating Test Sequences from UML Sequence Diagrams and State Diagrams. In: GI Jahrestagung, pp. 236–240 (2006)
[17] Garousi, V., Briand, L.C., Labiche, Y.: Control Flow Analysis of UML 2.0 Sequence Diagrams. In: ECMFA, pp. 160–174 (2005)
[18] Ledru, Y., du Bousquet, L., Bontron, P., Maury, O., Oriat, C., Potet, M.-L.: Test Purposes: Adapting the Notion of Specification to Testing. In: IEEE ASE, pp. 127–134 (2001)
[19] Li, L., Zhongsheng, Q., He, T.: Test purpose-based test generation for Web applications. In: IEEE NDT, pp. 238–243 (2009)
[20] En-Nouaary, A., Liu, G.: Timed test cases generation based on test purposes expressed as message sequence charts. In: IEEE ICTTA, pp. 585–586 (2004)
[21] Zeng, Y., Chen, L.-P., Chai, Y.-X., Zhou, X.: UML-based approach to generate polymorphic testing sequence and its implementation. In: WRI WCSE, pp. 251–255 (2009)
[22] Hartmann, J., Imoberdorf, C., Meisinger, M.: UML-Based Integration Testing. In: ACM ISSTA, pp. 60–70 (2000)
[23] OMG, UML 2.0 Superstructure Specification, Object Management Group, Final Adopted Specification ptc/03-08-02 (2003)
[24] Liu, Y.: Combining UML 2.0 sequence and state machine diagrams for control- and data-flow based integration testing, M.A.Sc. thesis, Carleton University (2009)
[25] Jin, Z., Offutt, A.J.: Coupling-based Criteria for Integration Testing. STVR 8(3), 133–154 (1998)
[26] Briand, L.C., Labiche, Y., Wang, Y.: A comprehensive and systematic methodology for client-server class integration testing. In: IEEE ISSRE, pp. 14–25 (2003)
[27] Ali, S., Iqbal, M.Z., Arcuri, A., Briand, L.C.: A Search-based OCL Constraint Solver for Model-based Test Data Generation. In: IEEE QSIC (2011)
[28] Andrews, J.H., Briand, L.C., Labiche, Y., Namin, A.S.: Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. IEEE TSE 32(8), 608–624 (2006)

Model Transformations for Migrating Legacy Models: An Industrial Case Study

Gehan M.K. Selim[1], Shige Wang[2], James R. Cordy[1], and Juergen Dingel[1]

[1] School of Computing, Queen's University, Kingston, Ontario, Canada, K7L3N6
[2] Electrical and Controls Integration Lab, General Motors Research & Development, Warren, Michigan, USA, 48090
{gehan,cordy,dingel}@cs.queensu.ca, shige.wang@gm.com

Abstract. Many companies in the automotive industry have adopted MDD in their vehicle control software development. As a major automotive company, General Motors has been using a custom-built, domain-specific modeling language, implemented as an internal proprietary metamodel, to meet the modeling needs in its control software development. As AUTOSAR (AUTomotive Open System ARchitecture) is being developed as a standard to ease the process of integrating components provided by different suppliers and manufacturers, there is a growing demand to migrate these GM-specific, legacy models to AUTOSAR models. Given that AUTOSAR defines its own metamodel for various system artifacts in automotive software development, we explore using model transformations to address the challenges in migrating GM legacy models to their AUTOSAR equivalents. As a case study, we have built a model transformation using the MDWorkbench tool and the Atlas Transformation Language (ATL). This paper reports on the case study, makes observations based on our experience to assist in the development of similar types of transformations, and provides recommendations for further research.

Keywords: Model Driven Development (MDD), model transformations, AUTOSAR, transformation languages and tools, automotive control software.

1 Introduction

MDD is a relatively new software development methodology that uses models for software specification and communication. In MDD, software development is a sequence of model transformations where abstract models are successively converted into detailed models, and eventually into code. Model transformations are implemented using a model transformation language, which can be declarative, imperative, or hybrid. While a declarative language yields a compact specification, an imperative language is more capable of specifying complex transformations.

As one of the early MDD adopters in industry, General Motors (GM) has created a domain-specific modeling language, implemented as an internal proprietary metamodel, for Vehicle Control Software (VCS) development. The metamodel defines modeling constructs for vehicle control software development, including schedules and interfaces. VCS models conforming to this metamodel have been used in several vehicle production domains at GM, such as body control and monitoring.

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 90–101, 2012.

Recently, AUTOSAR (the AUTomotive Open System ARchitecture) [2] has been developed as an industry standard to facilitate integration of software components from different manufacturers and suppliers and enable exchangeability and interoperability among them. AUTOSAR defines its own metamodel with a well-defined layered architecture and interfaces. Since converging to AUTOSAR is a strategic direction for future modeling activities, transforming GM legacy models to their equivalent AUTOSAR models becomes essential. Model transformation is a key enabling technology to achieve this convergence objective.

Despite the existence of studies in MDD industry adoption [19][23], no transformation is reported to have migrated legacy models in the automotive industry. To test the practicality of using transformations for migrating industrial legacy models, we have implemented a transformation of GM legacy models to AUTOSAR models.

The rest of this paper is organized as follows. Section 2 discusses the process context in which our transformation is implemented. Section 3 describes the source and target metamodels of the transformation. Section 4 details the transformation development. Section 5 discusses our experiences and issues that require further research. Section 6 provides a summary, a comparison to related work and future work.

2 VCS Development, Models and Model Transformations

Applying transformation requires understanding of the development process, which provides a context for the transformation. The VCS development process is described as a V-diagram (Fig. 1). The stages on the left-hand side of the V-diagram are design and implementation activities, and the stages of the right-hand are integration and validation activities. The design starts from system requirements models, which are decomposed into hardware and software subsystem requirements models. The subsystem requirements models then are assigned to engineering groups for refinement into design models and then implemented by hardware and software components. These implemented components are integrated into Electronic Control Units (ECUs), configured for a designated vehicle product. The components are then tested at various levels against their models on the same level on the left-hand side of the V-diagram.

Different types of models in different formalisms are manipulated in the VCS development process. For example, control models use differential equations and timing-variation functions; software models use dataflow diagrams or class diagrams; and architecture models use annotated block diagrams. Selected modeling tools (e.g., Simulink, Rhapsody) and languages (e.g., UML, AADL) are used for modeling.

The transformations used in the VCS development process can be *horizontal or vertical transformations*. Horizontal transformations manipulate models at the same abstraction level but possibly in different formalisms, e.g. transforming a Matlab Stateflow state machine into a UML state machine. Such transformations are normally used to verify integration of subsystems to realize a system function. The source and target modeling languages may have different syntax, but must share similar semantics. Vertical transformations manipulate models at different abstraction levels, e.g. generating a deployment model from software and hardware architecture models. Vertical transformations are usually more complex than horizontal transformations due to the different semantics of the source and target models.

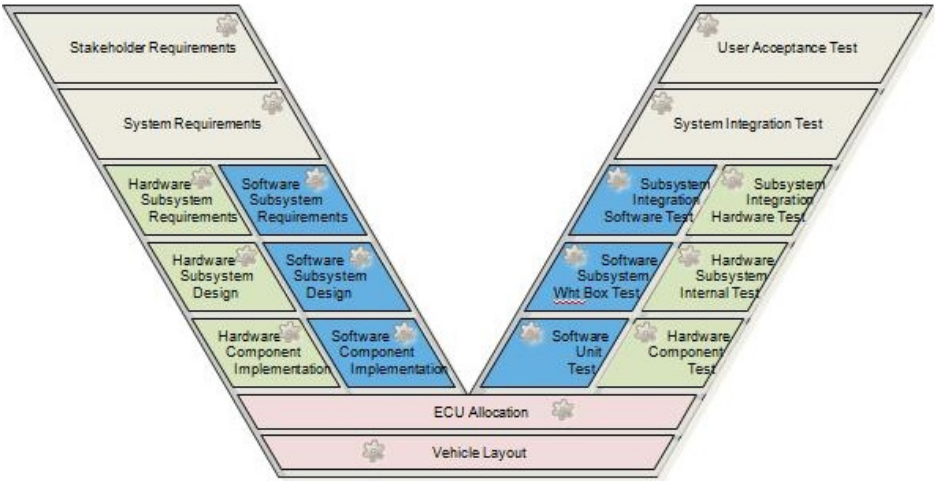


Fig. 1. V-Diagram for the VCS development process

3 Source and Target Metamodels

In this study, our models are those generated and used at the software subsystem design stage in the VCS development process. The source metamodel is an internal, proprietary GM metamodel which we will refer to as the GM metamodel. The target metamodel is the AUTOSAR System Template [2]. To simplify the exercise without losing generality, a subset of the two metamodels is manipulated in the transformation. Specifically, we focus on the modeling elements related to the software components' deployment and interactions, as discussed below.

3.1 The GM Metamodel

Fig. 2 illustrates the meta-types in the GM metamodel[1] that represent the physical nodes, deployed software components and their interactions. The *PhysicalNode* type specifies a physical node on which software is deployed. A *PhysicalNode* may contain multiple *Partition* instances, each of which defines a processing unit or a memory

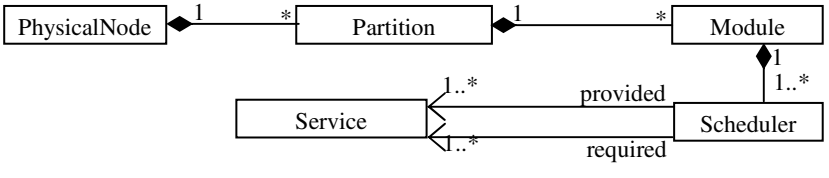


Fig. 2. The subset of the GM metamodel used in our transformation

[1] The metamodel has been altered for reasons of confidentiality. However, the relevant aspects required for the purpose of this paper have all been preserved.

partition on which software is deployed. Multiple *Module* instances can be deployed on a single *Partition*. The *Module* type is the atomic, reusable element in a product line and can contain multiple *Scheduler* instances. The *Scheduler* type is the basic unit for software scheduling and manages services provided or required by behavior-encapsulating entities. Thus, each *Scheduler* may provide or require many *Services*.

3.2 The AUTOSAR Metamodel

The AUTOSAR metamodel is defined as a set of templates, each of which is a collection of classes used to specify an AUTOSAR artifact. The *System* template [3] is used to capture the configuration of a system or an Electronic Component Unit (ECU). An ECU is a physical unit on which software is deployed. When used for the configuration of an ECU, the template is referred to as the *ECU Extract*. Fig. 3. shows the metatypes in the ECU Extract that capture software deployment on an ECU.

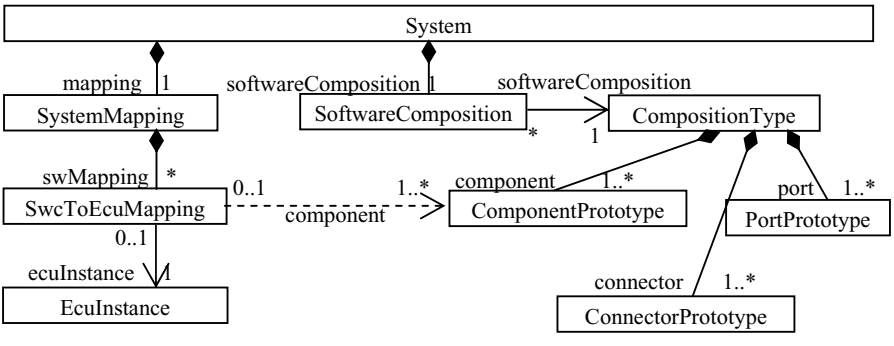


Fig. 3. The AUTOSAR System Template containing relevant types used by our transformation

The ECU extract contains the *System* type which aggregates *SoftwareComposition* and *SystemMapping* elements. The *SoftwareComposition* type points to the *CompositionType* type which eliminates any nested software components in a *SoftwareComposition* instance. The *SoftwareComposition* type models the architecture of the software components deployed on an ECU, their ports, and the ports' connectors. Software components are modeled using the *ComponentPrototype* type; ports are modeled using the *PPortPrototype* type or *RPortPrototype* type for providing or requiring services; connectors are modeled using the *ConnectorPrototype* type.

The *SystemMapping* type binds the software components to ECUs and the data elements to signals and frames. The *SystemMapping* type aggregates the *SwcToEcuMapping* type, which maps *ComponentPrototype* elements to an *EcuInstance*. According to AUTOSAR, only one *SwcToEcuMapping* instance should be created for every processing unit or memory partition in an ECU.

4 GM-to-AUTOSAR Model Transformation

We implement a GM-to-AUTOSAR model transformation to demonstrate the practicality of adopting transformations in the automotive industry. We rationalize our choice of the tool and language and we summarize the pragmatics of the chosen language. We then discuss the transformation rules and implementation details. Our transformation takes as inputs the source GM metamodel, the target AUTOSAR system template, and an input GM model. The output is an AUTOSAR model.

4.1 Selecting Model Transformation Tool and Language

Several tools and their accompanying languages have been considered for implementing the transformation including IBM Rational Asset Manager (RAM) [13], the RulesComposer add-on for IBM Rhapsody [14], and MDWorkbench [18].

After investigating the candidate tools, we concluded that IBM RAM and Rules Composer are not suitable for this transformation. RAM is a repository-based tool that offers APIs to create relationships between repository assets (e.g. models). The APIs can manipulate a model as a whole, not the individual model elements. As fine-grained manipulations are essential for our transformation, the support provided by RAM is not sufficient. RulesComposer is a rule-based model-to-text generator. Rules are specified as templates composed of static text and placeholders. When executed, the static text is copied into the output, and the placeholders are extracted from the input models. When defining rules, one must ensure that the template generates well-formed XMI files. Thus, defining the template is time-consuming and error-prone. Moreover, the rule templates can be very verbose, and thus, difficult to maintain.

MDWorkbench is an Eclipse-based tool for developing model-to-model transformations using the Atlas Transformation Language (ATL) [1] or the Model Query Language (MQL) [18]. ATL has declarative and imperative constructs, while MQL has imperative constructs only. MDWorkbench can manipulate models conforming to the metamodels registered in the tool (e.g. AUTOSAR) using rules defined in ATL and MQL. Thus, we choose MDWorkbench to implement the transformation. ATL was chosen rather than MQL because ATL provides flexibility to mix-and-match declarative and imperative constructs in the same rule definition.

4.2 ATL Pragmatics

In ATL, a model transformation is defined as a set of rules and helpers. Rules specify the creation of output model elements. Helpers are used to modularize a transformation. ATL defines four types of rules and two types of declarative helpers.

Rule Types. The four types of rules are matched rules, lazy rules, unique lazy rules, and called rules. A matched rule specifies how a source pattern is transformed to a target pattern. Matched rules are executed in the order of their specification and are automatically executed once for each matching pattern. A lazy rule is a rule that is executed only when called for a matching pattern and can be called multiple times for

any match in the input model. A unique lazy rule is a rule that is executed only when called and can be called at most once for any match in the input model. A called rule is a parameterized rule that is executed only when called and creates an element in the output model without matching any source patterns. The four kinds of rules have an optional imperative code block to specify complicated functionality.

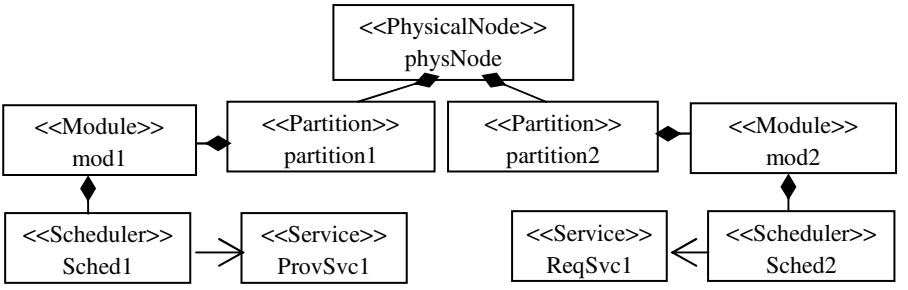
Matched rules are suitable for automatic detection of all pattern matches in the input model and creation of their corresponding target patterns; lazy rules and unique lazy rules are suitable for selective pattern matching, with consideration of the number of times these rules should be run; and called rules are suitable for creating output model elements that do not match any input model elements.

Helper Types. The two types of helpers are functional helpers and attribute helpers. A functional helper is a parametric function that is evaluated each time it is called. An attribute helper is a non-parametric function that is evaluated only in the first call. An attribute helper is more efficient to implement a non-parametric functionality. Otherwise, a functional helper can implement a parametric functionality.

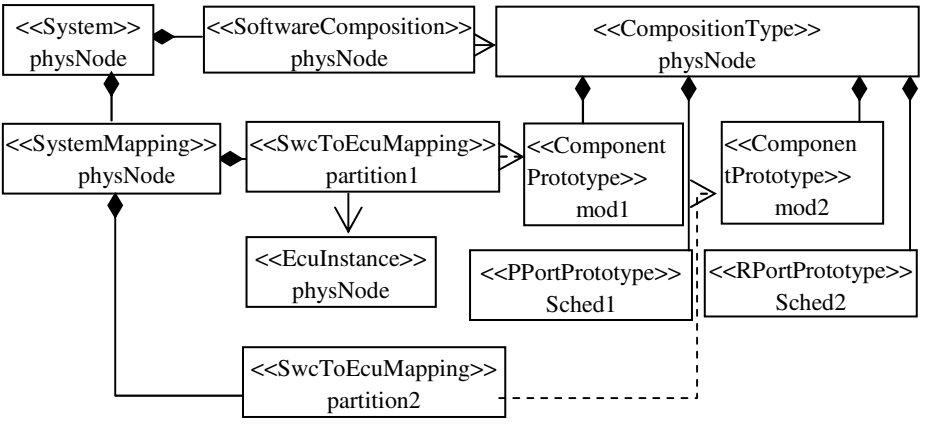
4.3 Model Transformation Design and Development

Our transformation rules were crafted in consultation with domain experts at GM to realize the required mappings between the metamodels. For reasons of confidentiality, we present a simplified version of the actual rules. Let *M* be the input GM model and *M'* the to-be-generated output AUTOSAR model. The rules are defined as follows:

1. For every element *physNode* of the *PhysicalNode* type in M, generate an element *sys* of the *System* type, an element *swcompos* of the *SoftwareComposition* type, a containment relation (*sys*, *swcompos*), an element *composType* of the *CompositionType* type, a relation (*swcompos, composType*), an element *sysmap* of the *SystemMapping* type, a containment relation (*sys*, *sysmap*) and an element *ecuInst* of the *EcuInstance* type in M';
2. For every element *partition* of the *Partition* type in M, generate an element *swc2ecumap* of the *SwcToEcuMapping* type and a containment relation (*sysmap, swc2ecumap*) in M';
3. For every containment relation (*physNode, partition*) in M, generate a relation (*swc2ecumap, ecuInst*) in M';
4. For every element *mod* of the *Module* type in M, generate an element *comp* of the *ComponentPrototype* type in M';
5. For every containment relation (*partition, mod*) in M, generate a containment relation (*composType, comp*) and a relation (*sw2ecumap, comp*) in M';
6. For every relation (*sched, svc*) of the *provided* type between a *sched* element of the *Scheduler* type and a *svc* element of the *Service* type with a containment relation *(mod, sched)*, generate a *pPort* element of the *PPortPrototype* type and a containment relation (*composType , pPort*) in M';
7. For every relation (*sched, svc*) of the *required* type between a *sched* element of the *Scheduler* type and a *svc* element of the *Service* type with a containment relation (*mod, sched*), generate a *rPort* element of the *RPortPrototype* type and a containment relation (*composType, rPort*) in M'.



(a) Sample input GM model



(b) Output AUTOSAR model for (a)

Fig. 4. (a) Sample GM input model and (b) its corresponding AUTOSAR output model

Fig. 4 demonstrates the required transformation from a sample GM model (Fig. 4 (a)) to its expected output AUTOSAR model (Fig. 4(b)) based on the above mentioned rules. The *PhysicalNode* element is mapped to a *System* element, an *EcuInstance* element, a *SystemMapping* element, a *SoftwareComposition* element, and a *CompositionType* element (Rule 1). The *Partition* elements are mapped to the *SwcToEcuMapping* elements (Rule 2), each of which is associated with the generated *EcuInstance* element (Rule 3). The *Module* elements are mapped to the *ComponentPrototype* elements aggregated by a *CompositionType* element and referred to by their corresponding *SwcToEcuMapping* elements (Rules 4-5). The *Scheduler* element aggregating a provided *Service* is mapped to a *PPortPrototype* element (Rule 6). The other *Scheduler* element is mapped in a similar manner (Rule 7).

The transformation development follows an iterative, incremental process. First, a simple GM model is created in the MDWorkbench model editor. Then, a transformation is implemented to transform the input GM model into an AUTOSAR model. The AUTOSAR model is then validated and if the transformation is correct, the process is repeated with additional types in the input model and additional transformation rules. If the output model contains errors, the transformation is analyzed and fixed.

Validation is performed manually. For an input GM model, an expected output AUTOSAR model is created in the MDWorkbench Model Editor. The transformation's output model is compared with the manually-created model. Equivalence of the models implies a correct transformation.

4.4 The Transformation Implementation Using ATL

The GM-to-AUTOSAR transformation contains two ATL matched rules and 9 functional helpers implementing the 7 rules in Section 4.3. We also define 6 attribute helpers to access the model attribute values. Table 1 lists the matched rules and functional helpers and their implemented rules in Section 4.3.

Table 1. Matched rules and functional helpers and the implemented rules

Matched Rule (MR)/ Functional Helper (FH)	Corresponding Rules: Section 4.3
MR1: `createComponent`	4
MR2: `initSysTemplate`	1
FH1: `initEcuInst`	1
FH2: `createSwc2EcuMappings` FH3: `initSingleSwc2EcuMapping`	2-3
FH4: `addComponents`	5
FH5: `getAllPPortsInEcu` FH6: `createPPort`	6
FH7: `getAllRPortsInEcu` FH8: `createRPort`	7
FH9: `getAllSWCinEcu`	5

The matched rule `createComponent` maps *Module* elements to *ComponentPrototype* elements. The matched rule `initSysTemp` maps a *PhysicalNode* element to a *System* element, a *SystemMapping* element, a *SoftwareComposition* element and a *CompositionType* element by calling the 9 functional helpers to implement rules 1-3 and 5-7. The helper `initECUInst` initializes an *EcuInstance* element. The helper `initSingleSwc2EcuMapping` initializes a *SwcToEcuMapping* instance. The helper `createSwc2EcuMappings` creates a list of *Swc2EcuMapping* elements corresponding to all the *Partition* elements in the input model. The helper `getAllSwcInEcu` creates the containment relation between the *CompositionType* elements and the *ComponentPrototype* elements. The helper `addComponents` creates the relation between the *SwcToEcuMapping* elements and their corresponding *ComponentPrototype* elements. The helper `getAllPPortsInEcu` creates a *PPortPrototype* element using the helper `createPPort` for *Schedulers* with at least one provided *Service*. Similar helpers generate *RPortPrototype* elements.

The ATL predefined function `resolveTemp` connects the *ComponentPrototype* elements created by the `createComponent` matched rule to the *CompositionType* elements created by the `initSysTemp` matched rule.

Implementing the transformation revealed some insights on using MDWorkbench and ATL in industrial applications. Both the GM and the AUTOSAR metamodels are complex in structure. To process models conforming to complex metamodels, ATL provides flexibility of using declarative and imperative constructs to implement complex transformations. Moreover, since the output models have many relationships among model elements, decisions on where an element should be created in the transformation such that it will be accessible for the downstream transformation are required. One such example is the relation between the *SoftwareComposition* element and the *ComponentPrototype* element. The transformation can be either specified as one rule or modularized as many rules. Although modularization requires that the order of the rules be consistent with their dependencies, ATL mitigates this drawback through the `resolveTemp` function which allows a rule to reference the elements that are yet to be generated by other rules regardless of their specification order. However, the `resolveTemp` function makes the transformation less readable and difficult to debug, so the function should be used only when necessary.

For validation, sample GM models were created in the MDWorkbench Model Editor, including the model in Fig. 4(a), and were used for evaluation. The output models were verified as described in Section 4.3. The transformation was found to produce the expected output models. Sample GM models were used for validation instead of actual GM models since many of the actual GM models did not conform to the GM metamodel, which represents a major challenge for adopting MDD in industrial environments.

5 Discussion

Based on our case study, we present open issues requiring further investigation for successful adoption of model transformations in the automotive industry. Recommendations for MDD tool and language development are also discussed.

5.1 Interoperability of MDD Tools

One of the major challenges encountered in our study was the lack of interoperability between commercial tools for developing transformations. Specifying the model transformation using ATL was not straightforward due to the formats of the manipulated metamodels. ATL can only manipulate MOF [21] or Ecore [23] metamodels, which the GM metamodel in Rhapsody native format is not compatible with. This required the conversion of the GM metamodel to a compatible format.

MDWorkbench has a Rhapsody connector that allows importing the GM metamodel into MDWorkbench and converting it to Ecore format. To avoid the issue of dual license from different vendors with different licensing policies with such an approach, we addressed the problem using XMI. An Ecore metamodel is essentially an XMI file and Rhapsody has an XMI toolkit to export Rhapsody metamodels to XMI files. Exporting the GM metamodel using the XMI toolkit generated an XMI file that does not conform to the Ecore meta-metamodel. To create an Ecore version, we import the

XMI into RulesComposer as a metamodel, which creates an Ecore metamodel and an Eclipse plugin project. Exporting the project from RulesComposer to MDWorkbench as a plugin generates a registered GM Ecore metamodel.

Blanc et al. [5] decomposed the interoperability problem into two concerns: the compatibility of the exchanged models, and the definition of an exchange mechanism. Their study proposed an architecture to address these two concerns. Implementing transformations between tools manipulating models that conform to different metamodels was proposed in [6], [4]. Kolovos et al. [15] proposed a framework that supports composing model management tasks with software development tasks in coherent workflows. Although these solutions have been integrated into IDEs, they are not fully automated in applications. MDD tools and transformation languages deserve further research to support easy integration and interoperability with each other.

5.2 Optimization in Model Transformations

Our transformation mapped GM models representing a deployment of the software components on physical nodes to their equivalent AUTOSAR models. The transformation exercised one mapping between the two metamodels and generated an AUTOSAR model reflecting the deployment configuration. From the deployment perspective, there are other design options that may yield a more desirable deployment in the output AUTOSAR model with respect to some utility function.

Solutions exist to support optimization during the transformation. Schätz et al. [22] proposed a formalized approach to explore the design space using rule-based transformations. Intermediate models were represented using a relational formalization and rules were represented using predicates. Drago et al. [9] proposed the QVT-Rational framework to explore design options which optimize quality metrics. First, a domain expert specifies the metamodels to be manipulated, the quality metrics of interest, the quality-prediction tool chain and the method for design feedback generation. Then, a designer specifies desirable values for quality metrics and asks QVT-Rational for design solutions. Tools that target industry use need to support scalable design-space exploration to aid developers in exploring design options of the generated model.

5.3 Dealing with Semantic Differences between Metamodels

Identifying which target metamodel elements best represent a given source metamodel element can be a difficult task. Reasons include: (1) the precise semantics of a metamodel may not have been documented sufficiently and only be fully known to metamodel developers themselves; consultation of these developers may be time consuming or even impossible. (2) The lack of support in metamodel evolution often means that the metamodels contain redundancies or inconsistencies. (3) The mapping of source to target elements is dependent on the transformation's purpose, because it determines to what extent aspects of model semantics can be removed (e.g., for abstraction), preserved (e.g., for refactorings) or refined (e.g., for code generation).

To facilitate transformation development, techniques to (1)enforce documenting metamodel semantics, (2) suggest mappings between metamodels using similarity matching or "learning" [17], [20], and (3) validate transformations are of high interest.

6 Conclusions and Future Work

In this study, we present a solution to migrating legacy VCS design models using model transformations in the automotive industry. The study has two major goals: (1) exploring the practicality of using model transformations in an industrial context to map between industrial metamodels and (2) benefitting GM by supporting automated convergence to AUTOSAR. The implemented transformation converts domain-specific GM models to their equivalent AUTOSAR models. We discussed the transformation context in the development process. Based on our experiences, we discuss which tool and language are appropriate for implementing the transformation, the challenges encountered and open issues that need further investigation.

Research studies on adopting MDD in industry have been published [19], [23], but a few investigated adopting transformations in industry. Daghsen et al. [8] transformed AUTOSAR timing models to classical scheduling models to perform timing analysis. Giese et al. [12] used triple graph grammars to synchronize between SysML system engineering models and AUTOSAR software engineering models. Our study differs from other studies in that the two manipulated metamodels are complex, industrial metamodels, which allows us to draw realistic conclusions regarding the practicality of adopting transformations in industry. Our study considers the entire transformation development process, from tool and language selection to transformation creation and validation. Future work includes extending the transformation to the full GM metamodel and using white-box or black-box testing [11], [16] for validation.

Acknowledgements. This work is supported in part by NSERC, as part of the NECSIS Automotive Partnership with General Motors, IBM Canada and Malina Software Corp.

References

[1] Atlas Transformation Language – ATL, http://eclipse.org/atl/

[2] AUTOSAR Consortium. AUTOSAR, http://AUTOSAR.org/

[3] AUTOSAR Consortium. AUTOSAR System Template, http://AUTOSAR.org/index.php?p=3&up=1&uup=3&uuup=3&uuuup=0&uuuuup=0/AUTOSAR_TPS_SystemTemplate.pdf

[4] Bezivin, J., Brunelière, H., Jouault, F., Kurtev, I.: Model engineering support for tool interoperability. In: Workshop in Software Model Engineering (WiSME), Montego Bay, Jamaica (2005)

[5] Blanc, X., Gervais, M.-P., Sriplakich, P.: Model Bus: Towards the Interoperability of Modelling Tools. In: Aßmann, U., Aksit, M., Rensink, A. (eds.) MDAFA 2003. LNCS, vol. 3599, pp. 17–32. Springer, Heidelberg (2005)

[6] Brunelière, H., Cabot, J., Clasen, C., Jouault, F., Bézivin, J.: Towards Model Driven Tool Interoperability: Bridging Eclipse and Microsoft Modeling Tools. In: Kühne, T., Selic, B., Gervais, M.-P., Terrier, F. (eds.) ECMFA 2010. LNCS, vol. 6138, pp. 32–47. Springer, Heidelberg (2010)

[7] Cottenier, T., Berg, A., Elrad, T.: The Motorola WEAVR:Model weaving in a large industrial context. In: Aspect-Oriented Software Development (AOSD), Vancouver, Canada (2007)
[8] Daghsen, A., Chaaban, K., Saudrais, S., Leserf, P.: Applying holistic distributed scheduling to AUTOSAR Mmethodology. In: Embedded Real-Time Software & Systems (ERTSS), Toulouse, France (2010)
[9] Drago, M.L., Ghezzi, C., Mirandola, R.: Towards Quality Driven Exploration of Model Transformation Spaces. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 2–16. Springer, Heidelberg (2011)
[10] Eclipse Modelling Framework (EMF), http://wiki.eclipse.org/EMF
[11] Fleurey, F., Baudry, B., Muller, P.-A., Le Traon, Y.: Qualifying input test data for model transformations. Software System Modelling (SoSyM) 8(2), 185–203 (2007)
[12] Giese, H., Hildebrandt, S., Neumann, S.: Model Synchronization at Work: Keeping SysML and AUTOSAR Models Consistent. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Nagl Festschrift. LNCS, vol. 5765, pp. 555–579. Springer, Heidelberg (2010)
[13] IBM Corporation. IBM Rational Asset Manager (RAM), http://www01.ibm.com/software/rational/products/ram/
[14] IBM Corporation. IBM Rational Rhapsody, http://www.ibm.com/developerworks/downloads/r/rhapsodydeveloper/index.html
[15] Kolovos, D., Paige, R., Polack, F.: A framework for composing modular and interoperable model management tasks. In: Model Driven Tool & Process Integration (MDTPI), Berlin, Germany (2008)
[16] Küster, J., Abd-El-Razik, M.: Validation of model transformations - First experiences using a white box approach. In: Model Development, Validation & Verification (MoDeVa), Genova, Italy, pp. 62–77 (2006)
[17] Mandelin, D., Kimelman, D., Yellin, D.: A Bayesian approach to diagram matching with application to architectural models. In: Intl. Conf. on Software Engineering (ICSE), Shanghai, China, pp. 222–231 (2006)
[18] Sodius. MDWorkbench, http://www.mdworkbench.com/
[19] 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)
[20] Nejati, S., Sabetzadeh, M., Chechik, M., Easterbrook, S., Zave, P.: Matching and merging of Statechart specifications. In: Intl. Conf. on Software Engineering (ICSE), Minneapolis, USA, pp. 54–64 (2007)
[21] Object Management Group (OMG): Meta Object Facility (MOF) Specification — Version 1.4 (April 2002)
[22] Schätz, B., Hölzl, F., Lundkvist, T.: Design-space exploration through constraint-based Mmodel transformation. In: Engineering of Computer Based Systems (ECBS), Oxford, UK, pp. 173–182 (2010)
[23] Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Chapter 5 Ecore Modeling Concepts. In: Eclipse Modeling Framework, 2nd edn. Addison-Wesley Professional (2009)
[24] Teppola, S., Parviainen, P., Takalo, J.: Challenges in the deployment of model driven development. In: Intl. Conf. on Software Engineering Advances (ICSEA), Porto, Portugal, pp. 15–20 (2009)

Derived Features for EMF by Integrating Advanced Model Queries*

István Ráth, Ábel Hegedüs, and Dániel Varró

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

Abstract. When designing complex domain-specific languages, metamodels are frequently enriched with *derived features* that correspond to attribute values or references (edges) representing computed information in the model. In the popular Eclipse Modeling Framework, these are typically implemented as imperative Java code.

In the paper, we propose to integrate the EMF-IncQuery model query framework to the Ecore metamodeling infrastructure in order to facilitate the efficient and automated (re-)computation of derived attributes and references over EMF models. Such an integration allows to define derived features using an expressive graph-based model query language [1], and offers high performance and scalability thanks to the incremental evaluation technique of EMF-IncQuery [2]. In addition, our approach offers to automate two typical associated challenges of EMF tools: (1) values of derived features are immediately recalculated upon model changes and (2) notifications are sent automatically to other EMF model elements to report changes in derived features.

1 Introduction

The design of complex domain-specific languages (e.g. in the automotive or avionics domains) frequently necessitate the use of advanced metamodeling techniques. Metamodels are complemented with *well-formedness constraints*, which enable the validation of the consistency of instance models with respect to such constraints, thus allowing to spot design flaws early in the development process. *Derived features*, which correspond to attribute values or references (edges) that represent computed information in the model, also proved to be useful in complex metamodeling scenarios. For instance, they frequently serve as auxiliary (helper) functions when implementing model simulators, and they also allow to compact the storage of the model.

In the popular Eclipse Modeling Framework (EMF), these derived features are most often implemented as user-defined algoritms computed by imperative

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

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 102–117, 2012.

Java code. Unfortunately, (1) most existing techniques re-calculate values of derived features in EMF models on-demand (i.e. when corresponding getters are called), which hinders integration into user interfaces where changes in the values of derived features should immediately be reflected. Furthermore, (2) it is challenging to properly implement notification propagation between (a chain of) derived features upon value changes, which is necessary when components or model elements are required to depend upon a derived feature. Finally, (3) as the calculation of derived features is always started from scratch (not taking previous computations and changes into account), it is also challenging to implement complex queries in Java in a way that does not severely impact the overall performance.

The advanced model query framework EMF-IncQuery has proved to be efficient in the incremental re-validation of well-formedness constraints over large models [2] scaling up to millions of elements[1]. Its expressive, declarative graph-based query language offers high level of reuse in queries [1]. In the paper, we propose to seamlessly integrate the EMF-IncQuery framework to the Ecore metamodeling infrastructure, in order to facilitate the efficient and automated computation of derived attributes and references over EMF models.

Our proposed approach, which is fully implemented and documented[2], offers to automate the entire workflow of developing derived features in EMF. In the approach, (1) derived features are defined using an expressive graph-based model query language and are calculated by an algorithm that (2) listens to all *incoming notifications* that impact on the computation, (2) issues *outgoing notifications* when the value of the derived feature changes, (3) keeps an *up-to-date cache* that is refreshed based on incoming notifications and used for computing outgoing notification. Finally, (4) since outgoing notifications may cause incoming notifications, the algorithm also *stabilizes such notification loops*.

In the rest of the paper, Section 2 provides a brief overview on derived features in EMF models. Then, we propose to use a graph based model query language to define derived features for EMF in Section 3. Section 4 provides a detailed architecture and core algorithms to synthesize notifications for derived features based upon incremental query evaluation. Additional issues for seamless integration to the EMF infrastructure are discussed in Section 5. Finally, Section 6 overviews related work and Section 7 concludes our paper.

2 Derived Features in EMF

Derived features in EMF models represent information that can be calculated from other model elements and typically represent an aggregate view of the model. Essentially, we distinguish between *derived attributes* and *derived references* (representing "virtual" connections between model elements). In our example, both are represented graphically by the derived stereotype in Figure 1.

[1] The current paper does not include performance specific contributions to the EMF-IncQuery framework, we kindly refer the reader to `http://viatra.inf.mit.bme.hu/performance` for additional details.

[2] `http://viatra.inf.mit.bme.hu/incquery/examples/derivedfeatures`

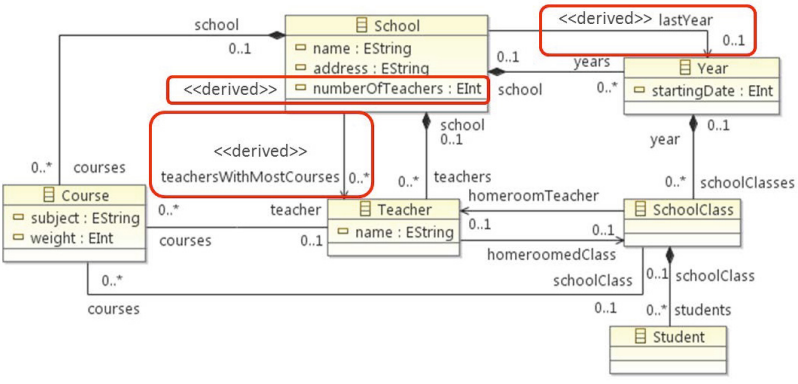


Fig. 1. The metamodel of the Schools domain

In the current paper, we illustrate our approach on a simple demonstration domain of Schools (encoded in EMF's Ecore language as illustrated in Figure 1) that manage Courses involving Teachers, and enroll their students assigned to Years and SchoolClasses. The metamodel contains simple EAttributes (like e.g. name of a Teacher or the startingDate of the school Year) and regular EReferences such as the school of a Teacher. More importantly for the sake of this paper, it also contains three *derived features*:

- numberOfTeachers is a derived attribute of School representing a counter for the total number of Teachers belonging to the School as represented by the corresponding school EReference;
- lastYear is a derived reference from School to Year, and points to the last academic Year stored in the model, which can be calculated from the startingDate of all Years;
- teachersWithMostCourses represent the busiest teachers of the School, i.e. those who teach the most courses.

Derived features in EMF are not maintained explicitly in instance models, but calculated on-demand by hand-written code. These calculations are frequently supported by ad-hoc Java implementations integrated directly into the EMF model representation, which significantly reduces the portability and compatibility of the metamodel.

Unfortunately, developers may encounter additional key challenges when aiming to use derived features in EMF models:

- **Performance.** Depending on the complexity of the semantics of derived features, their evaluation may impose a *severe performance impact* (since complex calculations and extensive model traversal may be necessary for execution). Note that this scalability issue is especially important when derived feature values need to be re-evaluated many times and will affect all other software layers using the model code, including the user interface, model transformations, well-formedness validators etc.

– **Notifications.** Due to the *difficulty of propagating notifications* for derived features, derived features are typically re-evaluated on demand. This may also manifest as model changes not (properly) triggering user interface updates. Note that EMF defines the notifications for derived features as well, however, it is the programmer's responsibility to create notifications. Since the values of derived features are usually not cached, proper notifications including the old values (e.g. setting a single value or removing from a list) are hard to implement. Furthermore, notifications of one derived feature may cause new notifications, leading to notification loops, the programmer must ensure that these are stabilized in order to avoid infinite loops.

Our proposal, namely, the integration of an advanced model query framework EMF-IncQuery provides a solution for all of these challenges using a high-level graph-based query language for defining derived value calculations. As the performance characteristics of the EMF-IncQuery engine have been shown to be agnostic of query complexity and model size [2], derived features of complex semantics and inter-dependencies can be used without severe evaluation performance degradation. Additionally the update propagation mechanism of EMF-IncQuery (using *delta monitors* [3]) will be connected to the EMF Notification layer so that the application software components are automatically kept up-to-date about the value changes of derived features.

3 Definition of Derived Features as Model Queries

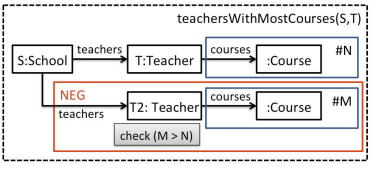
We now propose to use the graph pattern based model query language of EMF-IncQuery as the specification language for derived features of EMF models. Therefore a brief introduction to this query language is provided first, followed by a detailed description on how this general purpose query language is adapted to specify derived features.

3.1 Model Queries by Graph Patterns: An Overview

Graph patterns [4] 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 [1], while several examples will be given below.

3.2 Derived Features as Model Queries

Sample derived features First, we demonstrate on an example how the graph pattern teachersWithMostCourses(S,T) (Figure 2) can be used to express the calculation of the derived EReference teachersWithMostCourses (connecting *School* and *Teacher* in Figure 1), that is, to identify those teachers who have the maximum number of *Course* instances assigned (through the *Teachers.courses* reference).




```
1  pattern teacherWithMostCourses(S, T)=
2  {
3   School.teachers(S,T);
4   neg pattern moreCourses(S,T) = {
5    Teacher.courses(T,C) # N;
6    School.teachers(S,T2);
7    Teacher.courses(T2,C2) # M;
8    check(M > N);
9   }
10 }
```

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

This model query formulated as a graph pattern has two parameters: S and T, denoting the source and the target end of the derived EReference. The query defines the designated set of teachers by combining a NAC and cardinality constraints. It expresses that a teacher T belongs to this set if and only if there is no other teacher $T2$ whose number of courses M (calculated by counting the number of elements connected along the *courses* reference) would be larger than the number of courses N (counted as before) of teacher T. The right side of Figure 2 shows the corresponding textual syntax.

Model queries for derived features numberOfTeachers and lastYear are defined similarly in Figure 3. The definition of the latter contains some additional interesting language elements.

- The modifier shareable prescribes that different (but type consistent) pattern variables are allowed to be bound to the same model elements (e.g. $D1$ and $D2$ can be bound to the same date element).
- Y =/= Y2 checks that the two model elements bound to variables Y and $Y2$ are different.
- Using the find keyword, graph patterns are allowed to reuse other graph patterns. Therefore, if a derived feature is defined as a model query by a corresponding graph pattern, this derived feature can be reused in other queries, and thus, in other derived features. In fact, we will discuss in Section 5 that even legacy derived features (defined by Java code) can participate in such usage with appropriate notification mechanisms.

Derived features can be defined as model queries using the graph pattern based language of EMF-IncQuery if the following three well-formedness rules are met by corresponding query definitions:

```
pattern numberOfTeachers(S,N)=
{
 School.teachers(S,T) # N;
}
```

```
pattern lastYear(S,Y)= {
 find years(S,Y);
 neg shareable pattern laterYear(S,Y)= {
   find years(S,Y);
   find startingDateOfYear(Y,D1);
   find years(S,Y2);
   find startingDateOfYear(Y2,D2);
   check(D1 < D2);
   Y =/= Y2;
} }
```

Fig. 3. Model queries for numberOfTeachers and lastYear

1. *Each graph pattern should have exactly two parameters.* In case of derived attributes, the first parameter denotes the corresponding EClass of the attribute, while the second parameter denotes the value of the parameter itself (see numberOfTeachers). In case of derived references, the first parameter denotes the source (i.e. the container EClass) while the second parameter denotes the target of the EReference.
2. *First parameter: always input.* General model queries allow the same pattern to be used with either input or output parameters (i.e. parameter bindings can be carried out at execution time), in case of derived features, the first parameter (referring to the container) should always be an input parameter, which is a bound to a type-compliant contextual EMF object (e.g. S is bound in all three graph patterns above). This restriction is conceptually equivalent to the context element of an OCL constraint.
3. *Restrictions on result set.* In case of a derived features with explicit lower and upper bounds (e.g. 1..* or 0..1), the result set of the model query should comply with these restrictions. While upper bounds can be enforced by omitting results, the violation of lower bound is logged only as warnings.

In the actual query language, rules 1 and 2 are can be satisfied either by using exactly two query parameters, or 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 all three rules are checked at editing time by a built-in query language validator in the EMF-IncQuery tooling. In summary, the modular nature of the EMF-IncQuery language aims to allow the language engineer to construct a library of cross-referencing queries without copy-paste reuse.

4 From Incremental Query Evaluation to Notifications for Derived Features

In this section, we outline how the incremental query features of the EMF-IncQuery framework are integrated to notification-based applications in transparent way, by mapping changes of the results sets to notification objects for derived features. We present an architectural overview and an algorithm to carry out this mapping.

4.1 Incremental Evaluation of Queries

The key to efficient evaluation and change notification for derived features is the incremental graph pattern matching infrastructure of the EMF-IncQuery framework (introduced in [3]). The internal architecture is shown in Figure 4.

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

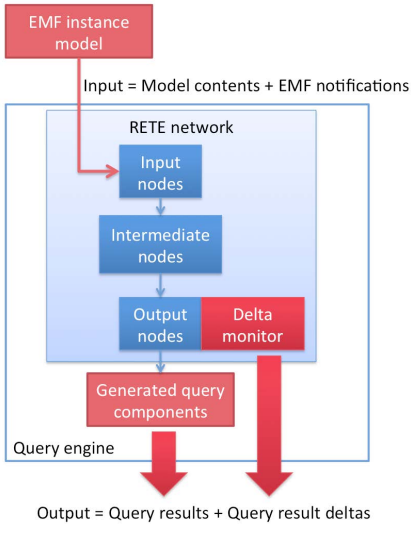


Fig. 4. The EMF-IncQuery architecture

Based on a query specification, EMF-IncQuery constructs a Rete rule evaluation network [3] 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).

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 retrieved directly. 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 [2] address the scalability challenge (Section 2) as long as enough memory is available, as they allow EMF-IncQuery-based derived features to be evaluated incrementally, even for complex queries over large instance models.

4.2 Integration Architecture

To support derived features, the outputs of the EMF-IncQuery engine are 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 5.

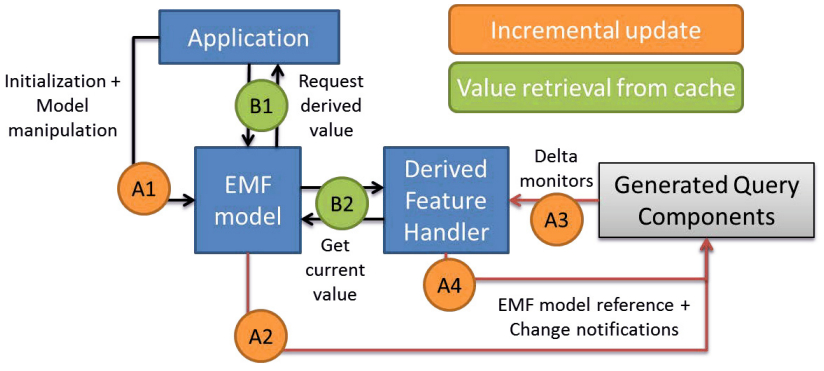


Fig. 5. Overview of the integration architecture

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 component type, *derived feature handlers* are attached to the EMF model plugin that integrate the generated query components (pattern matchers). This approach follows the official EMF guidelines of implementing derived features and is identical to how ad-hoc Java code, or OCL expression evaluators are integrated.

When an EMF application intends to read a derived feature (B1), the current value is provided by the corresponding derived feature 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 derived features (A3). Changes of derived features may in turn trigger further changes in the results sets of other derived features (A4).

Illustrative example. Figure 6 illustrates a detailed elaboration EMF-IncQuery feature handlers, which process elementary model manipulation notifications to update, and generate notifications for derived features. The figure corresponds to a case where the user created a new Teacher for a School through the Editor which is essentially a School.getTeachers().add(teacher) method call on the Model. During the add method, the School 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 it's finished with updating the Rete network, it invokes the callback method of each IncqueryFeatureHandler. Each handler has a Deltamonitor from which it retrieves the found

and lost match events since the last callback to processes them. During the processing, the handler may send notifications of its own that are propagated to listeners. Anytime the derived feature value is retrieved from the model (e.g. getNumberOfTeachers), the handler is accessed for the current value of the feature, which is returned directly.

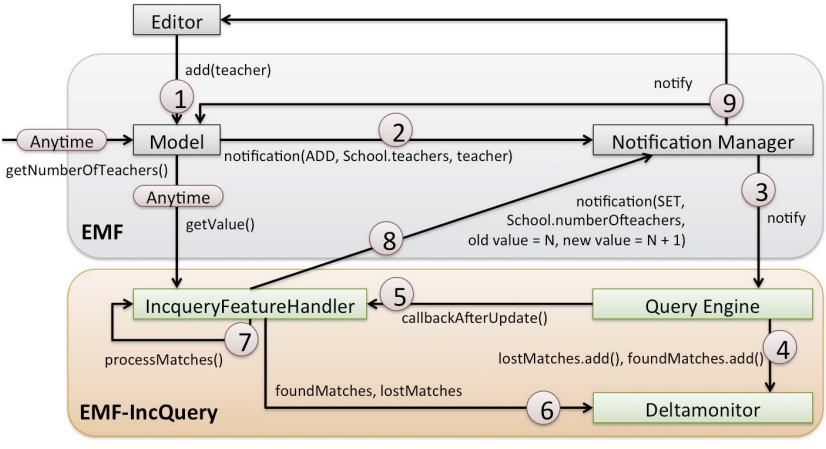


Fig. 6. Elaboration of the execution

4.3 From Changes of Match Sets to Notifications

We now explain the notification processing and propagation procedure in algorithmic detail. For the sake of simplicity, we introduce an auxiliary discriminator variable *Kind* whose value represents three distinct cases:

- SINGLE and MANY correspond to derived references of target multiplicity 1 and *, respectively (lastYear and teachersWithMostCourses in Figure 3);
- COUNTER corresponds to the simplified case where a value of the derived attribute is defined as the match set size of a query (see numberOfTeachers in Figure 3).
- More complex derived feature kinds with an arbitrary, deterministic iteration algorithm can also be handled by the approach.

The main part of our derived feature handler algorithm is an event loop that is called by the EMF-IncQuery query engine each time the underlying Rete network is updated as a result of some model manipulation (see Algorithm 1).

The algorithm is initialized with the following input variables (line 1): (1) the EObject *Source* whose derived feature is handled; (2) the derived *Feature*; (3) the *DeltaMonitor* for the query matcher; and (4) the previously mentioned discriminator value *Kind*. Each handler stores an internal value for the feature, initialized in line 2 depending on *Kind*. Finally, the handler uses two global variables: *pU* for storing partial events and the set *N* of unsent notifications.

Algorithm 1. Main event loop

```
1:  let S ← Source, F ← Feature, DM ← DeltaMonitor, k ← Kind                ▷ Input variables
2:  let (k = SINGLE)?iV ← null : (k = COUNTER)?iV ← 0 : iV ← ∅              ▷ Internal value init
3:  let pU ← null, N ← ∅                                                   ▷ Global variables
4:  function EVENTLOOP
5:      let pU ← null
6:      let found ← PROCESSFOUNDMATCHES(DM.matchFoundEvents)               ▷ Processing found events
7:      let DM.matchFoundEvents ← DM.matchFoundEvents \ found              ▷ Removing events
8:      let lost ← PROCESSLOSTMATCHES(DM.matchLostEvents)                  ▷ Processing lost events
9:      let DM.matchLostevents ← DM.matchLostevents \ lost                 ▷ Removing events
10:     if partialUpdate ≠ null then   ▷ Stored value not yet used, handle partial match event
11:         let N ← N ∩ notification(SET, null, pU)
12:         let iV ← pU                                                    ▷ Updating value
13:     end if
14:     while N ≠ ∅ do                                                    ▷ Notification sending loop
15:         let n ← N[0]
16:         let N ← N \ n
17:         S.eNotify(n)                                                  ▷ Sending notification through source
18:     end while
19: end function
```

Algorithm 2. Processing match-found events

```
1:  function PROCESFOUNDMATCHES(events)
2:      let P ← ∅
3:      for all e ∈ events do
4:          if e.source = S then
5:              let target ← e.target                                     ▷ Extracting feature target from event
6:              if k = COUNTER then
7:                  let N ← N ∩ notification(SET, iV, iV + 1)
8:                  let iV ← iV + 1                                       ▷ Updating value of repeating algorithm
9:              else if k = SINGLE then
10:                 let pU ← target                                       ▷ Storing value for later processing
11:             else if k = MANY then
12:                 let N ← N ∩ notification(ADD, null, target)
13:                 let iV ← iV ∩ target                                  ▷ Updating value
14:             end if
15:         end if
16:         let P ← P ∩ e
17:     end for
18:     return P
19: end function
```

The event loop starts from line 4, it first resets the partial event store, then processes matches found since the last execution of the loop (line 6). These events are supplied by the delta monitor of the query and removed after processing is finished. Similarly, the matches lost since the last execution are also processed (line 8) and removed after. When a derived feature with SINGLE kind is used and only a match-found event occurs without a match-lost event, an additional processing step is required to handle the partial event (line 11). This occurs when the query did not lose any matches since the last event loop, but a new match is found. This translates to a notification representing the setting of the feature value from *null* to *pU* (line 12). Finally, if there are any unsent notifications (line 14), the first notification n in the list N is sent through the *Source* EObject. By separating the notification sending from the calculation of the derived feature value, the notification loop is stabilized, since new notifications caused by n are simply added to the list N, which will be depleted after all, if causal circularity between the definitions of derived features is avoided.

Algorithm 3. Processing match-lost events

```
1:  function processLostMatches(events)
2:      let P ← ∅
3:      for all e ∈ events do
4:          if e.source = S then
5:              let target ← e.target                    ▷ Extracting feature target from event
6:              if k = counter then
7:                  let N ← N ∩ notification(SET, iV, iV − 1)
8:                  let iV ← iV − 1                      ▷ Updating value of repeating algorithm
9:              else if k = single then
10:                 let N ← N ∩ notification(SET, target, pU)        ▷ Using stored value
11:                 let iV ← target                                  ▷ Updating value
12:                 let pU ← null                                    ▷ Resetting stored value
13:             else if k = many then
14:                 let N ← N ∩ notification(REMOVE, target, null)
15:                 let iV ← iV \ target                             ▷ Updating value
16:             end if
17:         end if
18:         let P ← P ∩ e
19:     end for
20:     return P
21: end function
```

New matches. The handling of match-found events is detailed in Algorithm 2. The processFoundMatches function iterates through the match-found events (line 3), and extracts the target object from the event (line 5), if the source EObject of the event equals *Source*. Depending on the *Kind* of the feature, a notification is created and the internal value is updated (line 7 for counter and line 12 for many). For single kind features, the target object is stored for later usage (line 10). Finally, the list of processed events is returned.

Lost matches. The handling of match-lost events is similar to the processing of match-found events, see Algorithm 3. The processLostMatches function iterates through the match-lost events (line 3), and extracts the target object from the event (line 5), if the source EObject of the event equals *Source*. Depending on the *Kind* of the feature, a notification is created and the internal value is updated (line 7 for counter and line 14 for many). For single kind features, the stored value of *pU* is used for creating the notification (line 10). Finally, the list of processed events is returned at the end of the function.

Summary. In summary, the combined pattern matching and notification processing process ensures that EMF-IncQuery-based derived features behave exactly as normal features of EMF instance models. This addresses the final, integration-related challenge of Section 2), by ensuring that user interfaces, model validators etc. can safely depend on such derived features, without on-demand querying.

5 Integration Issues with EMF Tooling

5.1 Integration with Ecore

In the prototype implementation of our proposal, we integrated our approach to the EMF Tooling by a code generator that supports the automatic generation

of integration code for our components (EMF-IncQuery *derived feature handlers*). The input of the code generator is a simple generator model (referencing the EMF genmodel for the domain) that crosslinks derived features with EMF-IncQuery query specifications (which are stored as EMF models thanks to the Xtext2-based tooling).

```
    teachersWithMostCoursesHandler = IncqueryFeatureHelper.createHandler(
            this,
            SchoolIncqDerivedPackage.Literals.SCHOOL__TEACHERS_WITH_MOST_COURSES,
            TeacherWithMostCoursesMatcher.FACTORY,
            "School",
            "Teacher",
            FeatureKind.MANY_REFERENCE);
 * @generated NOT
 */
public EList<Teacher> getTeachersWithMostCourses() {

    if(teachersWithMostCoursesHandler != null) {
        Collection<Object> temp = teachersWithMostCoursesHandler.getManyReferenceValue();
        return new UnmodifiableEList<Teacher>(this,
                SchoolIncqDerivedPackage.Literals.SCHOOL__TEACHERS_WITH_MOST_COURSES,
                temp.size(), temp.toArray());
    } else {
        return new UnmodifiableEList<Teacher>(this,
                SchoolIncqDerivedPackage.Literals.SCHOOL__TEACHERS_WITH_MOST_COURSES,
                0, null);
    }
}
```

Fig. 7. Sample generated code for derived feature handler instantiation and getter

The generated integration code (Figure 7) consists of (a) the instantiation of derived feature handlers (in the constructor of EObjects), which ensures that their lifecycle is tied to the hosts, to enable their garbage collection together with the instance model itself; (b) getter implementations that delegate calls to the appropriate function of the feature handler object, and wrap the result in unmodifiable ELists to ensure that any attempt to write to derived features will result in a runtime exception.

5.2 Integration with Legacy Java Code for Derived Features

In practice, a complete refactoring of an EMF-based tool to exclusively use EMF-IncQuery-based derived features might not be realistic. Hence, we implemented an additional *derived feature adapter* (Figure 8) as a lightweight add-on component for EMF model plugins, which can be used to augment existing derived feature implementations (regardless of whether Java or OCL is used).

The basic concept motivated by a suggestion in the Eclipse FAQ[3] is analogous to the previous discussion. The language engineer can add a few lines of Java code

[3] http://wiki.eclipse.org/EMF/Recipes#Recipe:_Derived_Attribute_Notifier

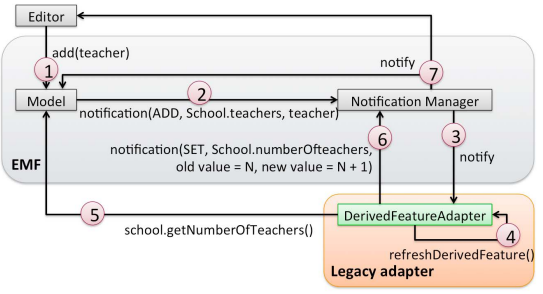


Fig. 8. Derived feature handlers

to the generated EMF model plugin: these derived feature adapters attach listeners (through the EMF Notification API) to the (explicitly specified) features a derived feature depends on, and receive notifications when model changes are registered (steps 1-2-3 in Figure 8). These notification objects are then processed and converted into new notification objects for the derived feature, propagating through the manager to application code (steps 4-5-6-7 in Figure 8).

This approach has additional key advantages: (1) notification support can be added – with a small implementation effort – to "legacy" derived features, without having to re-write them in EMF-IncQuery; (2) queries specified in EMF-IncQuery (whether for derived features, or on-the-fly validation purposes, or within model transformations) can reference derived features seamlessly.

6 Related Work

Model queries over EMF. There are several technologies for providing declarative model queries over EMF, e.g. EMF Model Query 2 [5] and EMF Search [6]. Other graph pattern based techniques like [7,8] have been successfully applied in an EMF context. But none of these support incremental evaluation, therefore they cannot be used for integrating derived features in the way we proposed.

OCL evaluation approaches. OCL [9] 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 provides a powerful query interface that evaluates OCL expressions over EMF models. However, backwards navigation along references in EMF can still have low performance [2], which may influence the performance of OCL evaluation without additional support.

Aiming at incremental evaluation, the impact analysis (IA) approach for OCL constraints [10] is functionally similar to our approach (but conceptually different in terms of underlying incremental algorithm) in using change notifications to identify constraints that should be re-evaluated, although it does not cache partial matches. An added feature of our approach is to automatically provide

notifications for derived features (which could be – but currently is not – implemented for OCL tools). As future work, we aim to compare IA and our approach and even combine the benefits of our current implementation with the benefits of existing OCL-based solutions.

Cabot et al. [11] present an approach for incremental runtime validation of OCL constraints and uses promising optimizations, however, it works only on boolean constraints, and as such it is less expressive than our technique.

An interesting model validator over UML models [12] incrementally re-evaluates constraint instances whenever they are affected by changes, however the approach is only applicable in environments where read-only access to the model can be easily recorded, unlike EMF. Additionally, the approach is tailored for model validation, general-purpose model querying is not viable.

Balsters [13] 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. Note, that while the OCL approach does not offer incrementality, an EMF-IncQuery based approach would.

Derived features. There are several approaches that make extensive use of derived features or provide additional support for their usage.

The PROGRES language [14] allows the rule-based programming of graph rewriting systems. It uses derived attributes for encoding node properties concerning aspects of dynamic semantics. The language includes support for defining how these derived attributes are calculated, and also uses functional attribute dependencies that would allow similar implementation as described in Section 5. However, PROGRES has not been adapted to EMF up to our best knowledge. The FUJABA [15] tool suite also supports derived edges by path expressions in a non-incremental way.

In [16] Diskin describes a theoretical model synchronization framework that uses derived references for propagating changes between corresponding models. The derived attributes defined in the framework are queries, similarly to our approach, although algebraic and not incrementally updated.

Scheidgen [17] presents a MOF tool that 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, thus incrementality is not supported.

JastEMF [18] 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 problems.

ConceptBase.cc [19] is a database system for metamodeling and method engineering. It allows the definition of active rules that react to events and can update the database or call external routines. Using this functionality, it would be possible to create derived features in models that are updated incrementally based on the data stored in the ConceptBase.cc database. On the other hand, this framework has not been applied in an EMF context.

In a previous tool paper of ours [20], we give an architectural overview of the entire EMF-IncQuery tool where derived features are listed as one of the new features of the tool. The current paper provides all the technical details on using incremental queries for derived features in EMF.

7 Conclusion

We proposed to seamlessly integrate the EMF-IncQuery framework to the EMF infrastructure in order to facilitate the efficient and automated computation of derived attributes and references over EMF models by advanced model queries. Our approach (1) allows to define derived features using an expressive graph-based model query language, (2) offers high performance and scalability thanks to the incremental evaluation technique of EMF-IncQuery [2], and (3) automatically provides notifications to and from derived features which has to be implemented manually in an EMF application.

Future work. Our current research directions include the application of query-based derived features for handling soft interconnections in EMF models and for managing virtual EMF objects derived from query result sets. Furthermore, the EMF-IncQuery framework is under active development, with derived feature support being only one of its many capabilities.

Acknowledgements. We would like to thank E.D. Willink for his suggestions on improving the paper and the anonymous reviewers for their helpful comments.

References

1. 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)
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. LNCS, vol. 6394, pp. 76–90. Springer, Heidelberg (2010)
3. 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)
4. Varró, D., Balogh, A.: The Model Transformation Language of the VIATRA2 Framework. Science of Computer Programming 68(3), 214–234 (2007)
5. The Eclipse Project: EMF Model Query 2, http://wiki.eclipse.org/EMF/Query2
6. The Eclipse Project: EMFT Search, http://www.eclipse.org/modeling/emft/?project=search
7. 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)

8. Giese, H., Hildebrandt, S., Seibel, A.: Improved flexibility and scalability by interpreting story diagrams. In: Proceedings of GT-VMT 2009, vol. 18. ECEASST (2009)
9. The Object Management Group: Object Constraint Language, v2.3.1. (January 2012), http://www.omg.org/spec/OCL/2.3.1/
10. Uhl, A., Goldschmidt, T., Holzleitner, M.: Using an OCL impact analysis algorithm for view-based textual modelling. ECEASST 44 (2011)
11. Cabot, J., Teniente, E.: Incremental integrity checking of UML/OCL conceptual schemas. J. Syst. Softw. 82(9), 1459–1478 (2009)
12. 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)
13. 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)
14. Schürr, A.: Introduction to PROGRESS, an Attribute Graph Grammar Based Specification Language. In: Nagl, M. (ed.) Graph-Theoretic Concepts in Computer Science. LNCS, vol. 411, pp. 151–165. Springer, Heidelberg (1990)
15. Nickel, U., Niere, J., Zündorf, A.: The FUJABA environment. In: Proc. ICSE 2000, pp. 742–745 (2000)
16. Diskin, Z.: Model Synchronization: Mappings, Tiles, and Categories. In: Fernandes, J.M., Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2009. LNCS, vol. 6491, pp. 92–165. Springer, Heidelberg (2011)
17. Scheidgen, M.: On implementing MOF 2.0 new features for modelling language abstractions (2005)
18. 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)
19. Jeusfeld, M.A., Jarke, M., Mylopoulos, J.: Metamodeling for Method Engineering. The MIT Press (2009)
20. 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)

A Lightweight Approach for Managing XML Documents with MDE Languages

Dimitrios S. Kolovos, Louis M. Rose,
James Williams, Nicholas Matragkas, and Richard F. Paige

Department of Computer Science, University of York,
Derramore Lane, Heslington, York, YO10 5GH, UK
{dkolovos,louis,jw,nikos,paige}@cs.york.ac.uk

Abstract. The majority of contemporary model management languages that support MDE tasks (such as model transformation, validation and code generation) require models to be captured using metamodelling architectures such as Ecore and MOF. In practice, a limited subset of modelling tools – with the exception of some UML tools – build atop such architectures. For many modelling languages and tools outside of the UML/Ecore/MOF family, plain XML is a widely-used model storage and exchange format. In this paper, we argue for the importance of integrating XML-based models in the MDE process. We identify the challenges involved in integrating XML-based models into MDE processes, and we present a technical solution that addresses these challenges, which enables developers to perform a wide range of model management tasks on models captured in XML.

1 Introduction

Model Driven Engineering (MDE) focuses on elevating machine-processable *models* to first-class artefacts of the software development process. MDE is technology-agnostic in the sense that it does not prescribe a specific architecture or framework atop which models should be captured, or a particular format in which they should be stored. Therefore, in principle, any structured machine-processable document can play the role of a model in an MDE process.

The majority of recent research on MDE has focused on 3-level metamodelling architectures where models conform to metamodels which are defined in terms of architecture / framework-specific metamodelling languages such as MOF [1] or Ecore [2]. As a result, most contemporary model management languages that support tasks such as model transformation, code generation, model validation etc., require models to be captured atop such architectures. In practice however, very few modelling tools actually use MOF/Ecore to manage and store their models; XML appears to be the most commonly used model persistence format [3].

Although XML is clearly inferior for MDE purposes to elaborate object-oriented metamodelling architectures from a technical perspective, due to its popularity and simplicity, it has the potential of lowering the entry barrier and

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 118–132, 2012.

playing the role of a stepping stone for the wider adoption of automated model management and MDE. In an effort to make model management languages and MDE techniques more accessible to XML-literate developers, in this paper we propose a lightweight approach for providing first-class support for managing XML documents within Epsilon [4], a mature and well-established family of model management languages. By first-class in this context, we mean support for XML documents in their native standard W3C DOM[1] representation, and not through an implicit or a behind-the-scene injection to a proprietary representation (e.g. as instances of an Ecore-based XML metamodel) that Epsilon already provides support for.

The remainder of the paper is organised as follows. In Section 2 we discuss the importance of XML for MDE and highlight the need for providing first-class support for XML documents in model management languages. In Section 3 we discuss how we implemented such support in the context of Epsilon and in Section 4 we present a case study that illustrates using languages of the Epsilon platform to perform model management tasks on XML documents. In Section 5 we discuss related work and in Section 6 we conclude and provide directions for further work on this subject.

2 Background and Motivation

XML is ubiquitous in the world of software: a vast number of off-the-shelf tools either use XML as a native format for storing structured data they manage, or provide import/export capabilities from/to XML. Also, literally hundreds of modelling languages have been defined atop XML [3] such as the Systems Biology Markup Language (SBML)[2], the Financial products Markup Language[3] and exchange formats such as the Graph Exchange Language[4]. This is consistent with the experience obtained through our interaction with industrial collaborators, which also indicates that XML is particularly popular as a native representation format for bespoke modelling tools developed in-house.

Compared to contemporary metamodelling architectures such as EMF and MOF, plain XML is technically inferior as it only supports capturing tree-structured metadata and does not provide support for types. XML Schema remedies these limitations by adding support – among other – for formalising cross-references between XML elements, and for defining complex and primitive types but is still geared more towards the concrete representation rather than towards the abstract syntax of the metadata it models.

Despite its technical limitations, we argue that plain XML has the potential to lower the entrance barrier for developers that have not been previously exposed to MDE; it can be used to enable developers to capture primitive *models* that contain domain-specific information of interest and start managing them in an

[1] http://www.w3.org/DOM/
[2] http://sbml.org/
[3] http://www.fpml.org/
[4] http://www.gupro.de/GXL/

automated manner with MDE languages, without requiring them to first become familiar with metamodelling architectures such as EMF and MOF. In the sequel, and if automated model management (e.g. code generation, model transformation, validation) appears to be delivering results, a transition to a contemporary metamodelling architecture that addresses the limitations of XML is the next logical step.

In the following sections we demonstrate an approach for contributing first-class support for managing plain XML documents to the Epsilon family of MDE languages. Our aim with this work is to render MDE languages useful and attractive to developers that are experienced with XML but not with metamodelling architectures, thus providing means that lower the entrance barrier to MDE.

2.1 Epsilon

Epsilon [4] is a mature and well-established family of interoperable languages for model management. Languages in Epsilon can be used to manage models of diverse metamodels and technologies. At the core of Epsilon is the Epsilon Object Language (EOL) [5], an OCL-based imperative language that provides features such as model modification, multiple model access, conventional programming constructs (variables, loops, branches etc.), user interaction, profiling, and support for transactions. Although EOL can be used as a general-purpose model management language, its primary aim is to be reused in task-specific languages. Thus, a number of task-specific languages have been implemented atop EOL, including those for model transformation (ETL), model comparison (ECL), model merging (EML), model validation (EVL), model refactoring (EWL) and model-to-text transformation (EGL).

With regard to the types of models supported, Epsilon provides the Epsilon Model Connectivity (EMC) layer that offers a uniform interface for interacting with models of different modelling technologies. Currently, EMC drivers have been implemented to support EMF [2] (XMI 2.x), MDR [6] (XMI 1.x) and Z [7] specifications in LaTeX using CZT [8] Also, to enable users to compose complex workflows that involve a number of individual model management tasks, Epsilon provides ANT [9] tasks and an inter-task communication framework discussed in detail in [10].

3 Managing XML Documents in Epsilon

In this section we illustrate how we have implemented first-class support for managing XML documents in all the languages provided by Epsilon to perform tasks such as model transformation, validation, comparison, refactoring, merging and code generation.

3.1 The Epsilon Model Connectivity Layer

The Epsilon Model Connectivity (EMC), shown in Figure 1, is an abstraction layer for managing models in Epsilon. Via EMC, the model management

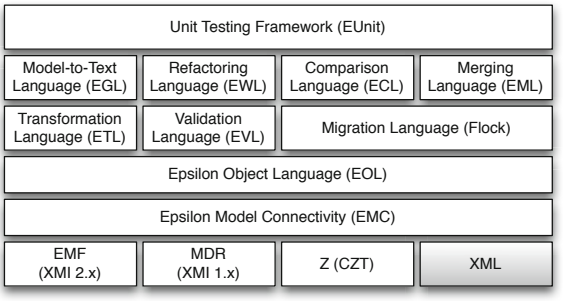


Fig. 1. Overview of the architecture of Epsilon

languages of Epsilon can query and modify models of varying modelling technologies without needing to be aware of the low-level details of each technology.

EMC enables developers to implement *drivers* – essentially classes that implement the `IModel` interface of Figure 2 – to support diverse modelling technologies. This work illustrates the design and implementation of an additional driver (on top of the existing drivers for managing EMF, MDR and Z models) for interacting with schema-less XML documents.

In addition to abstracting over the technical details of specific modelling technologies, EMC facilitates the concurrent management of models expressed with different technologies. For instance, Epsilon can be used to transform an EMF-based model into an MDR-based model, to perform inter-model validation between a Z model and an EMF model, or to develop a code generator that consumes information from an EMF-based and an XML model at the same time.

3.2 The Plain XML EMC Driver

To support management of XML documents with languages of the Epsilon family, a new driver has been implemented atop EMC. The XML driver uses the standard W3C DOM Java implementation as the underlying representation for XML documents and this, combined with the ability of Epsilon languages to invoke Java operations enables developers to access the complete standard DOM API[5] in their model management programs.

By contrast to drivers for 3-tier architectures such as EMF/MOF, in this driver, in the absence of a metamodel or a schema, the developer needs to assist Epsilon in navigating the XML model and performing type coercion / casting. Therefore, the plain XML driver (shaded box in Figure 1) uses predefined naming conventions to allow developers to programmatically access and modify XML documents in an elegant and concise way. It is worth noting that providing support for XML documents in Epsilon did not require any other changes beyond

[5] `http://www.w3.org/DOM/`

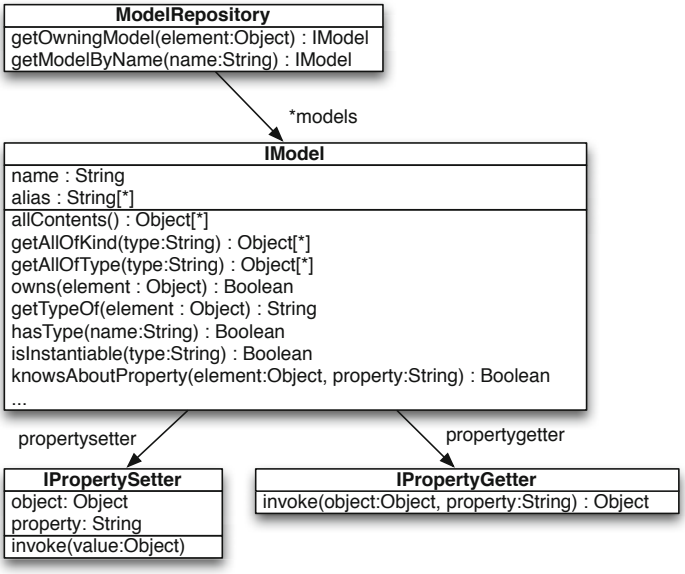


Fig. 2. The Model Connectivity Layer of Epsilon

the addition of the XML driver. This section outlines the supported conventions using the document of Listing 1.1 as a running example.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<library>
  <book title="Eclipse Modeling Framework" pages="744">
    <author>Dave Steinberg</author>
    <author>Frank Budinsky</author>
    <author>Marcelo Paternostro</author>
    <author>Ed Merks</author>
    <published>2009</published>
  </book>
  <book title="Eclipse Modeling Project: A Domain-Specific
      Language (DSL) Toolkit" pages="736">
    <author>Richard Gronback</author>
    <published>2009</published>
  </book>
  <book title="Official Eclipse 3.0 FAQs" pages="432">
    <author>John Arthorne</author>
    <author>Chris Laffra</author>
    <published>2004</published>
  </book>
</library>
```

Listing 1.1. Example XML document

Accessing Elements by Tag Name. The `t_` prefix before the name of the tag is used to represent a type, instances of which are all the elements with that tag. For instance, `t_book.all` can be used to retrieve all elements tagged as `<book>` in the document, `t_author.all` to retrieve all `<author>` elements etc. Also, if `b` is an element with a `<book>` tag, then `b.isTypeOf(t_book)` shall return true.

```
1  // Get all <book> elements
2  var books = t_book.all;
3
4  // Get a random book
5  var b = books.random();
6
7  // Check if b is a book
8  // Prints 'true'
9  b.isTypeOf(t_book).println();
10
11 // Check if b is a library
12 // Prints 'false'
13 b.isTypeOf(t_library).println();
```

Listing 1.2. Accessing elements by tag name

Getting and Setting Attribute Values of Elements. An attribute name, prefixed by `a_`, can be used as a property of the element object. For example, if `b` is the first book of the XML document of Listing 1.1, `b.a_title` will return EMF Eclipse Modeling Framework. Attribute properties are readable and writable.

In this example, `b.a_pages` will return `744` as a string. For `744` to be returned as an integer, the `i_` prefix should be used instead (i.e. `b.i_pages`. The driver also supports the following prefixes: `b_` for boolean, `s_` for string (alias of `a_`) and `r_` for real values.

```
1  // Print all the titles of the books in the library
2  for (b in t_book.all) {
3    b.a_title.println();
4  }
5
6  // Print the total number of pages of all books
7  var total = 0;
8  for (b in t_book.all) {
9    total = total + b.i_pages;
10 }
11 total.print();
12
13 // ... the same using collect() and sum()
14 // instead of a for loop
15 t_book.all.collect(b|b.i_pages).sum();
```

Listing 1.3. Getting and setting attribute values

Getting/Setting the Text of an Element. The `.text` property can be used to read/write the value of the textual content of an element.

```
for (author in t_author.all) {
  author.text.println();
}
```

Listing 1.4. Getting and setting the text of an element

Accessing the Parent of an Element. The `.parentNode` read-only property can be used to retrieve the parent node of an element.

```
// Get a random book
var b = t_book.all.random();

// Print the tag of its parent node
// Prints 'library'
b.parentNode.tagName.println();
```

Listing 1.5. Getting the parent of an element

Retrieving the Children of an Element. The `.children` read-only property can be used to retrieve all the child-nodes of an element.

```
// Get the <library> element
var lib = t_library.all.first();

// Iterate through its children
for (b in lib.children) {
  // Print the title of each child
  b.a_title.println();
}
```

Listing 1.6. Getting the children of an element

Getting Child Elements with a Specific Tag Name. Using what has been discussed so far, this can be achieved using a combination of the `.children` property and the `select`/`selectOne()` EOL operations. However, the driver also supports `e_` and `c_`-prefixed shorthand properties for accessing one or a collection of elements with the specified name respectively. `e_` and `c_` properties are read-only.

```
// Get a random book
var b = t_book.all.random();

// Get its <author> children using the
// .children property
var authors = b.children.select(a|a.tagName = "author");

// Do the same using the shorthand
```

```
authors = b.c_author;

// Get its <published> child and print
// its text using the
// .children property
b.children.selectOne(p|p.tagName = "published").text.
    println();

// Do the same using the shorthand
// (e_ instead of c_ this time as
// we only want one element,
// not a collection of them)
b.e_published.text.println();
```

Listing 1.7. Getting children with a specific tag name

Creating New Elements. The standard new operator can be used to create new elements in the XML document.

```
// Check how many <books> are in the library
// Prints '3'
t_book.all.size().println();

// Creates a new book element
var b = new t_book;

// Check again
// Prints '4'
t_book.all.size().println();
```

Listing 1.8. Creating new elements

Add a Child to an Existing Element. The `.appendChild(child)` operation can be used to add a child-node to an element. If the node to be added is already a child of another node, it is first detached from its previous parent.

```
// Create a new book
var b = new t_book;

// Get the library element
var lib = t_library.all.first();

// Add the book to the library
lib.appendChild(b);
```

Listing 1.9. Adding a child to an existing element

Setting the Root Element of an XML Document. The `.root` property of the model can be used to set the root element of an XML document.

```
1 XMLDoc.root = new t_library;
```

Listing 1.10. Setting the root element of an XML document

The XML driver also supports (optional) caching so that expensive operations such as collecting all elements with a particular tag do not need to be performed repetitively.

3.3 Alternative Design Choices

As discussed above, the plain XML driver presented in this section makes use of particular naming conventions – such as the `t_`, and `c_` prefixes – to specify XML model element types, to distinguish between child elements and attribute values, to specify the expected result type when retrieving children of an element by name (single element vs. collection of elements), and to perform type-casting of the values of attributes. Given that Epsilon is dynamically typed, the prefixes could have been eliminated in an alternative design, but this would have introduced several inconveniences, which are now discussed.

Ordinarily, Epsilon throws a runtime error when trying to use undefined variables or types of model element that do not exist. These runtime errors provide valuable information to users, alerting them to problems with their programs. Schema-less models, such as plain XML models, do not provide type information. Without the `t_` prefix for XML model element types, the EOL interpreter would become unable to distinguish between undefined variables and XML model element types. Consequently, undefined variables would have to be treated as XML model element types, and the user would not be alerted to the potential error in their program. In addition, had we not used `c_` and `e_` to distinguish between single and multiple children, all child element navigations would need to return a collection of elements. Also, explicitly specifying attribute value type-casting using the `i_`, `r_`, `b_` prefixes avoids unintended type casts.

In our view, employing these prefixes makes up to an extent for the lack of a formal metamodel and makes code easier – albeit slightly more verbose – to write and maintain.

4 Case Study

In this section we present a case study that demonstrates how the XML driver that was presented in the previous section can be used to validate and transform the XML-based OO model of Listing 1.11 to a respective EMF-based UML model. This case study has been intentionally kept simple for brevity reasons.

```
1 <?xml version="1.0"?>
2 <model>
3    <class name="Customer">
4       <property name="name" type="String"/>
5       <property name="address" type="Address"/>
6    </class>
```

```
7     <class name="Invoice">
8        <property name="serialNumber" type="String"/>
9        <property name="customer" type="Customer"/>
10       <property name="items" type="InvoiceItem" many="true"
             />
11    </class>
12    <class name="InvoiceItem">
13       <property name="quantity" type="Integer"/>
14       <property name="product" type="Product"/>
15    </class>
16    <class name="Product">
17       <property name="name" type="String"/>
18       <property name="unitPrice" type="Float"/>
19    </class>
20    <class name="Address">
21       <property name="number" type="String"/>
22       <property name="postCode" type="String"/>
23    </class>
24    <datatype name="String"/>
25    <datatype name="Integer"/>
26    <datatype name="Float"/>
27 </model>
```

Listing 1.11. OO model captured using XML

Listing 1.12 illustrates a constraint expressed using the Epsilon Validation Language (EVL) which checks that the type of each property in the XML model of Listing 1.11 corresponds to a defined type (class or datatype). Line 2 defines that the constraint applies to all elements tagged as `property` and line 5 checks that there is an element tagged as `datatype` or `class` whose name matches the value of the `type` attribute of the `property`. If such an element is not found, in lines 7-9 a diagnostic message is produced.

```
1  import "util.eol";
2  context t_property {
3     constraint TypeMustBeDefined {
4
5        check : typeForName(self.a_type).isDefined()
6
7        message : "Property " + self.a_name + " of class " +
8           self.parentNode.a_name + " is of unknown type: " +
9           self.a_type
10    }
11 }
```

Listing 1.12. XML validation constraint expressed in EVL

Listing 1.13 illustrates a model-to-model transformation expressed using the Epsilon Transformation Language (ETL) that transforms the XML model of Listing 1.11 to an EMF-based UML model. The transformation consists of 4 rules

which transform elements tagged as `model`, `class`, `property` and `datatype` to respective Models, Classes, Properties and DataTypes in the target UML model. This transformation illustrates how EMC enables programs in all Epsilon languages to manage models that conform to different technologies concurrently.

```
1  import "util.eol";
2
3  rule t_model2Model
4     transform s : XML!t_model
5     to t : UML!Model {
6
7     t.packagedElement.addAll(s.children.equivalent());
8  }
9
10 rule t_class2Class
11    transform s : XML!t_class
12    to t : UML!Class {
13
14    t.name = s.a_name;
15    t.ownedAttribute.addAll(s.children.equivalent().
16       select(e|e.isTypeOf(UML!Property)));
17 }
18
19 rule t_property2Property
20    transform s : XML!t_property
21    to t : UML!Property {
22
23    t.name = s.a_name;
24    var type = typeForName(s.a_type);
25    t.type = type.equivalent();
26
27    if (s.b_many) { t.upper = -1; }
28
29    if (not type.isTypeOf(XML!t_datatype)) {
30       var association = new UML!Association;
31       association.ownedEnds.add(t);
32       var opposite = new UML!Property;
33       opposite.type = s.parentNode.equivalent();
34       association.ownedEnds.add(opposite);
35       UML!Model.all.first().packagedElement.
36          add(association);
37    }
38
39 }
40
41 rule t_datatype2DataType
42    transform s : XML!t_datatype
43    to t : UML!DataType {
44
```

```
45      t.name = s.a_name;
46
47  }
```

Listing 1.13. XML to UML transformation expressed in ETL

```
1  operation typeForName(type : String) {
2     return allTypes().selectOne(t|t.a_name = type);
3  }
4
5  operation allTypes() : Sequence {
6     return XML!t_class.all.includingAll(XML!t_datatype.all);
7  }
```

Listing 1.14. Utility methods (util.eol) used in Listings 1.13 and 1.12

5 Related Work

The importance of XML has been recognised by the developers of the Eclipse Modelling Framework (EMF) and as a result EMF provides support for managing schema-based XML documents. To support schema-based XML documents, EMF provides a built-in transformation that can produce an Ecore metamodel from an XML schema, a parser that can parse XML files that conform to an XSD into in-memory models that conform to the respective Ecore metamodel, and a serialiser that can then persist in-memory models back to XML. While Ecore and XSD share many common features such as being able to define complex structures (e.g. through EClasses in Ecore and Complex Types in XSD), inheritance, references with cardinality etc. they also differ in some respects. For instance, XSD can define anonymous complex types while Ecore cannot define anonymous EClasses, EMF models can contain multiple root objects while XML documents can only have one root node, Ecore does not have equivalent constructs for the XSD `<choice>` element or the `mixed` feature, etc. In an effort to compensate for these differences, the XSD to Ecore transformation employs conventions that, while necessary, can lead to non-straightforward Ecore metamodels.

For example, the XML Schema of listing 1.15 is transformed into the Ecore metamodel illustrated in Figure 3. In the Ecore metamodel the reader can observe the ItemType and ItemType1 EClasses which have been generated by the anonymous complex types in lines 8 and 21 of the XSD. Also, in order for a developer to access the text content of an item element, they need to query the `mixed` feature of ItemType (or ItemType1) – which is not straightforward for a developer with no EMF expertise.

```
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
3
4  <xs:element name="invoice">
```

```
5    <xs:complexType>
6      <xs:sequence>
7        <xs:element name="item">
8          <xs:complexType mixed="true">
9            <xs:sequence>
10             <xs:element name="unitPrice" type="xs:float"
                   />
11           </xs:sequence>
12         </xs:complexType>
13       </xs:element>
14     </xs:sequence>
15   </xs:complexType>
16 </xs:element>
17 <xs:element name="order">
18   <xs:complexType>
19     <xs:sequence>
20       <xs:element name="item">
21         <xs:complexType mixed="true">
22           <xs:sequence>
23             <xs:element name="quantity" type="xs:int"/>
24             </xs:sequence>
25           </xs:complexType>
26         </xs:element>
27       </xs:sequence>
28     </xs:complexType>
29   </xs:element>
30 </xs:schema>
```

Listing 1.15. Example XML Schema

The Atlas Transformation Language (ATL) provides support for schema-less XML documents through an injection transformation that converts an XML document to a respective EMF model that conforms to a simple Ecore-based XML metamodel, and an extraction transformation that does the reverse. As such, the syntax for managing XML documents in ATL is particularly verbose as illustrated by Listings 1.16 and 1.17.

```
1 XML!t_book.all.first().a_title.println();
```

Listing 1.16. EOL statement that prints the title of the first book

```
1 XML!Element.allInstances()->select(e|e.name = 'book')->
      first()
2   .children->select(c|c.oclIsTypeOf(XML!Attribute)
3     and c.name = 'title')->first().value.println();
```

Listing 1.17. Equivalent ATL statement that prints the title of the first book

Xlinkit [11] is a tool for checking consistency issues in distributed documents. Using Xlinkit, developers can specify cross-document constraints that can be automatically evaluated to reveal inconsistencies. For the specification of

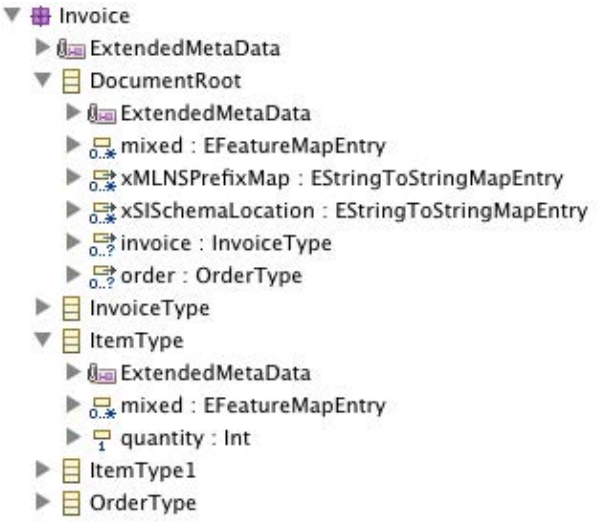


Fig. 3. Ecore metamodel generated from the XML Schema of Listing 1.15

constraints, Xlinkit defines an XML-based language that uses XPath [12] for document navigation. Listing 1.18 demonstrates an exemplar Xlinkit constraint that applies on a UML and a Java model and states that for each class in the UML model, a class with the same name must exist in the Java model. In our view, the main shortcoming of this approach is that the concrete syntax of the expression language is based on XML and that, as illustrated in Listing 1.18, results in lengthy and challenging to read and maintain specifications.

```
<globalset id="classes"
  xpath="//Foundation.Core.Class[@xmi.id]"/>
<globalset id="javaclasses" xpath="/java/class"/>
<consistencyrule id="r1">
 <forall var="c" in="classes">
  <exists var="j" in="javaclasses">
   <equal
     op1="c/Foundation.Core.ModelElement.name/text()"
     op2="j/@name"/>
  </exists>
 </forall>
</consistencyrule>
```

Listing 1.18. Example Xlinkit constraint

6 Conclusions and Further Work

In this paper we have highlighted the importance of XML in the context of MDE; in particular we have discussed the role of XML both as a legacy format in which

a significant amount of data is already encoded, and as a means of lowering the entrance barrier for newcomers in MDE. Following that we illustrated a technical solution for adding first-class support for XML to the Epsilon MDE platform so that plain XML documents can be used in a wide range of MDE tasks such as model validation, transformation, comparison, merging and code generation as they are and without needing to first transform them to models that conform to metamodelling architectures such as MOF or EMF.

Although in this paper we have illustrated a solution for adding support for managing XML documents to a particular family of model management languages, it is worth noting that this approach is also directly applicable to other model management languages (such as ATL[13] or MOFScript[14]) that provide a layer of indirection between the language run-time and the concrete modelling technologies they support.

Acknowledgements. The work in this paper was supported by the European Commission via the MADES and INESS projects, co-funded under the 7^{th} Framework programme (grants #218575 (INESS), #248864 (MADES)).

References

1. Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification, http://www.omg.org/cgi-bin/doc?ptc/03-10-04
2. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modelling Framework. 2nd edn. Eclipse Series. Addison-Wesley Professional (December 2008)
3. CoverPages. XML Applications and Initiatives (June 2005), http://xml.coverpages.org/xmlApplications.html
4. Eclipse Foundation. Epsilon Modeling GMT component, http://www.eclipse.org/gmt/epsilon
5. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon Object Language (EOL). In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 128–142. Springer, Heidelberg (2006)
6. Sun Microsystems. Meta Data Repository, http://mdr.netbeans.org
7. Woodcock, J., Davies, J.: Using Z: Specification, Refinement, and Proof. Prentice-Hall (March 1996)
8. Community Z Tools, http://czt.sourceforge.net
9. The Apache Ant Project, http://ant.apache.org
10. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: A Framework for Composing Modular and Interoperable Model Management Tasks. In: Proc. Workshop on Model Driven Tool and Process Integration (MDTPI), ECMDA, Berlin, Germany (June 2008)
11. Nentwich, C., Capra, L., Emmerich, W., Finkelstein, A.: xlinkit: A Consistency Checking and Smart Link Generation Service. ACM Transactions on Internet Technology 2(2), 151–185 (2002)
12. W3C. XML Path Language (XPath), Official Web-Site, http://www.w3.org/TR/xpath
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. Oldevik, J.: MOFScript User Guide, http://www.eclipse.org/gmt/mofscript/doc/MOFScript-User-Guide.pdf

Bridging the Gap between Requirements and Aspect State Machines to Support Non-functional Testing: Industrial Case Studies

Tao Yue and Shaukat Ali

Certus Software V&V Center, Simula Research Laboratory
P.O. Box 134, 1325, Lysaker, Norway
{tao,shaukat}@simula.no

Abstract. Requirements are often structured and documented as use cases while UML state machine diagrams often describe the behavior of a system. State machines capture rich and detailed behavior of a system, which can serve as a basis for many automated activities such as automated test case and code generation. The former is of interest in this paper. Non-functional behavior can be modeled using standard UML state machines, but usually results in complex state machines. To cope with such complexity, Aspect-Oriented Modeling (AOM) is often recommended. AspectSM is a UML profile defined to model crosscutting behavior on UML state machines called as aspect state machines with the focus of supporting model-based test case generation for non-functional behavior. Hence, an automatic transition from use cases to aspect state machines would provide significant, practical help for testing system requirements. In this paper, we propose an approach to automatically generate aspect state machines from use cases for the purpose of non-functional testing. Our approach is implemented in a tool, which we used for two industrial case studies. Results show that high quality aspect state machines can be generated, which can be manually refined at a reasonable cost to support testing.

Keywords: Use Case Modeling, UML, Aspect State Machine, Model-Based Testing (MBT), State-based Testing.

1 Introduction

Model-based testing (MBT) has attracted much attention in both industry and academia, as indicated by a large number of MBT tools produced in recent years [1]. MBT however relies on complete and precise models for executable test case generation. Developing such models has always been a challenge, especially for large-scale industrial systems, and entails a thorough domain understanding and solid modeling expertise. Oftentimes, developing such models is difficult for Software Quality Assurance teams as they are often not sufficiently acquainted with modeling. On the other hand, these teams are comparatively much more familiar with writing textual use cases and the application domain.

This paper is part of an automated methodology (aToucan [2, 3]) to assist the development of high-level models from use case models (UCMods). The aToucan

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 133–145, 2012.

tool relies on a number of existing technologies and is built as an Eclipse plug-in. aToucan involves three steps. 1) Requirements engineers manually define use cases complying with a use case modeling approach, Restricted Use Case Modeling (RUCM) [4, 5], which relies on a use case template and a set of restriction rules for textual Use Case Specifications (UCSs) to reduce the imprecision and incompleteness inherent to UCSs. We have conducted two controlled experiments with human subjects [5] to evaluate RUCM and results indicate that RUCM, though it enforces a template and restriction rules, has enough expressive power, is easy to use, and helps improve the understandability of use cases. 2) aToucan reads these textual UCSs to identify Part-Of-Speech (POS) and grammatical relation dependencies of sentences, and then records that information into an instance of UCMeta (our intermediate metamodel). UCMeta complies with the restrictions and use case template of RUCM, is currently composed of 108 metaclasses, and is implemented as an Ecore model, using Eclipse EMF [6]. During this transformation, the Stanford Parser [7] is used as a Natural Language (NL) parser in aToucan. 3) Transform the instance of UCMeta into UML models. The generation of UML models relies on Kermeta [8]. Due to space limitation, the detailed description of aToucan is given in [2] and we omit it from this paper.

We have proposed an approach [9], as part of the aToucan framework, to automatically generate standard, system-level UML state machines from use case models. The focus of this work is however on generating aspect state machines in AspectSM, a UML profile which was defined to model crosscutting behavior on UML state machines with the focus of supporting model-based non-functional testing [10]. AspectSM has been evaluated both empirically (through controlled experiments, e.g., [11-13]) and practically (via real industrial case studies, e.g., [10]) to be applicable. These models are subsequently refined such that executable test cases can be generated using our MBT tool, TRansformation-based tool for Uml-baSed Testing (TRUST) [14]. The TRUST tool has been successfully applied to two industrial case studies for model-based functional and non-functional testing [14]. In this paper, we performed two industrial case studies: a video conference system from Cisco Systems Inc, Norway [15] and a subsea oil production system from FMC Technologies [16] were performed, to evaluate UML state machines and aspect state machines modeling non-functional behaviors generated by aToucan. The generated state machines were evaluated by domain experts, who assessed them to mostly conform to the existing, manually developed state machines.

The rest of the paper is organized as follows. In Section 2, we briefly discuss RUCM, AspectSM and the running example being used to exemplify the transformation. The transformation approach is discussed in Section 3. The industrial case studies are discussed in Section 4. The related work is presented in Section 5 and Section 6 concludes the paper.

2 Background

In this section, we briefly, due to space limitation, review the use case modeling approach RUCM (Section 2.2) and AspectSM (Section 2.3). A running example will be presented in Section 2.1 to exemplify RUCM and the transformations.

2.1 Running Example

The running example is a simplified subsystem (called Saturn) of a communication system (Video Conferencing System (VCS)) developed by Cisco Systems Inc, Norway [15], which is a leading global provider of telepresence, high-definition video conferencing and mobile video products and services. This subsystem is the industrial case study used to evaluate this work (Section 4).

The core functionality of a VCS is sending and receiving multimedia streams. The use case diagram capturing main use cases of the simplified subsystem Saturn is given in Figure 1. Saturn deals with establishing video conferencing calls, disconnecting calls, and starting/stopping presentations. It can also receive requests for establishing calls, disconnecting calls, and starting/stopping presentations from other video conferencing systems (Endpoints) participating in a videoconference. The endpoints communicating with Saturn are modeled as secondary actors in the use case diagram.

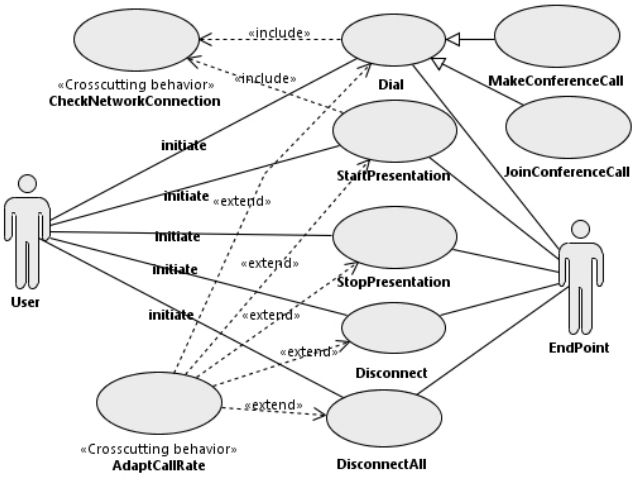


Fig. 1. Use case diagram of Saturn

2.2 RUCM

RUCM encompasses a use case template and 26 well-defined restriction rules [4]. Rules are classified into two groups: restrictions on the use of NL, and rules enforcing the use of specific keywords for specifying control structures. The goal of RUCM is to reduce ambiguity and facilitate automated analysis. Two controlled experiments evaluated RUCM in terms of its ease of application and the quality of the analysis models derived by trained individuals [4, 5]. Results showed that RUCM is overall easy to apply and that it results in significant improvements over the use of a standard use case template (without restrictions to the use of NL), in terms of the quality of derived class and sequence diagrams. UCSs documented with RUCM of use cases `AdaptCallRate` and `CheckNetworkConnection` (Figure 1) are presented in Table 1 and Table 2. UCSs of the other use cases in Figure 1 are provided in [9] for a reference.

2.3 AspectSM

Using the AspectSM profile [10], we model crosscutting behaviors as a UML state machine with stereotypes, which is called as aspect state machine and hence reduce modeling effort when compared to modeling crosscutting directly on UML state machines. The readability of models is then improved as crosscutting behavior that tends to be redundant when modeled directly is clearly separated out and expressed once. This profile was developed by augmenting many of the concepts in existing UML state machine profiles for AOM in order to achieve the specific goal of supporting automated, model-based non-functional testing. Due to the space limitation, we didn't provide stereotypes and their attributes in this paper, however, their details can be found in [10].

Table 1. Use case AdaptCallRate

Use Case Name	AdaptCallRate
Brief Description	The system adjusts the call rate based on the quality of services of the network.
Precondition	Network connection is established.
Primary Actor	Timer
Basic Flow	1) The system VALIDATES THAT the network is experiencing packet loss. 2) The system gradually decreases conference call rate to the minimum call rate.
	Postcondition: The system is in a degraded mode.
Specific Alt. Flow (RFS Basic flow 1)	1) The system gradually increases conference call rate up to the maximum call rate.
	Postcondition: The system is in the normal operation mode.

Table 2. Use case CheckNetworkConnection

Use Case Name	CheckNetworkConnection
Brief Description	The system checks the network connection.
Precondition	The system is idle.
Basic flow	**1)** The system VALIDATES THAT Network connection is OK. **Postcondition**: The network connection is checked.
Specific Alt. Flow (RFS Basic flow 1)	**1)** The system sends a failure message to User. **2)** ABORT. **Postcondition**: The system is idle.

3 Approach

The transformation from the textual UCMod to the instance of UCMeta is not discussed in this paper, but provided in [2] for a reference. In this section, we however only focus on the transformation from instances of UCMeta to the base and aspect state machines. We present detailed transformation rules in Section 3.1. The steps required for transforming generated state machines into the state machines that can be used for automated test case generation are presented in Section 3.2.

3.1 Transformation

Before generating aspect state machines, we should first identify crosscutting behaviors. Two heuristics should be followed to identify crosscutting behavior use cases (CUSs) in the use case model. First, if a use case is included more than once or

extends more than one use cases, it is a candidate CUS. A user later on can always manually identify crosscutting behavior. The rest of the section describes how to generate the base state machine and after that, how to generate aspect state machine(s) for identified crosscutting behaviors.

3.1.1 Generating Base State Machine

The transformation from an instance of UCMeta to a base state machine involves three rules, summarized in Table 3. These rules are adapted from [9], where we described the transition to standard UML state machines. The difference between generating a base state machine in the context of aspect-oriented modeling and a standard UML state machine is that when generating the base state machine, identified crosscutting behaviors should not be specified in the base state machine, as they are modeled as aspect state machines. Subscripts on rule numbers (Column 1, Table 3) indicate the type of the rule: "c" and "a" denote composite and atomic rules, respectively; a composite rule is decomposed, whereas an atomic rule is not. The automatically generated base state machine diagram for the use case model presented in Section 2.1 is provided in Figure 2.

Table 3. Summary of transformation rules for generating the base state machine for the system

Rule #	Description
1_a	a) Generate an instance of UML `StateMachine`, as the base state machine, for the use case model. The name of the state machine should be the name of the system (e.g., 'Saturn') plus 'base state machine'. b) Generate the initial state (instance of `Pseudostate` with `PseudostateKind = initial`) for the state machine. c) Generate an instance of `State`, named as 'start', representing the start state of the state machine. d) Generate an instance of `Transition`. Its trigger is named as 'construct'. This transition connects the initial state to the start state.
2_c	Invoke rules 2.1-2.4 to process each use case of the use case model that is considered as not specifying crosscutting behaviors.
2.1_a	Generate an instance of `State` for the precondition of the use case, as long as such a state has not been generated, which is possible because two use cases might have the same preconditions.
2.2_a	Generate an instance of `State` for the postcondition of the basic flow of the use case, as long as such a state has not been generated.
2.3_a	If the use case does not include any other use case, then connect the state corresponding to the precondition to the state representing the postcondition of the basic flow of the use case with the transition whose trigger is the name of use case. Otherwise, invoke rule 3.
2.4_c	Process the postcondition of each alternative flow of the use case.
$2.4.1_a$	Generate an instance of `State` for the postcondition of each alternative flow.
$2.4.2_a$	Connect the state corresponding to the precondition of the basic flow to the states corresponding to the postconditions of the alternative flows with transitions whose triggers are the name of use case.
3_c	Process Include relationships between use cases. Notice that use cases capturing identified crosscutting behaviors are not processed with the following rules, as separate rules will be applied to generate aspect state machines for them (Section 3.1.2).
3.1_a	Connect the precondition of the included use case to the postcondition of the flow of events where the included use case is included in the including use case through a transition. Connect the precondition of the included use case to the postconditions of the alternative flows of the included use case.
3.2_a	If there is a sequence of included use cases in the including use case, then link all the preconditions of the included use cases sequentially, and then link the precondition of the last use case to the postcondition of the including use case through transitions.

Rule 1 generates an instance of UML `StateMachine` for a UCMod, which can then be visualized as a state machine diagram, the initial state (an instance of `Pseudostate`), the start state (an instance of `State`), and the transition (an instance of `Transition`) from the initial state to the start state.

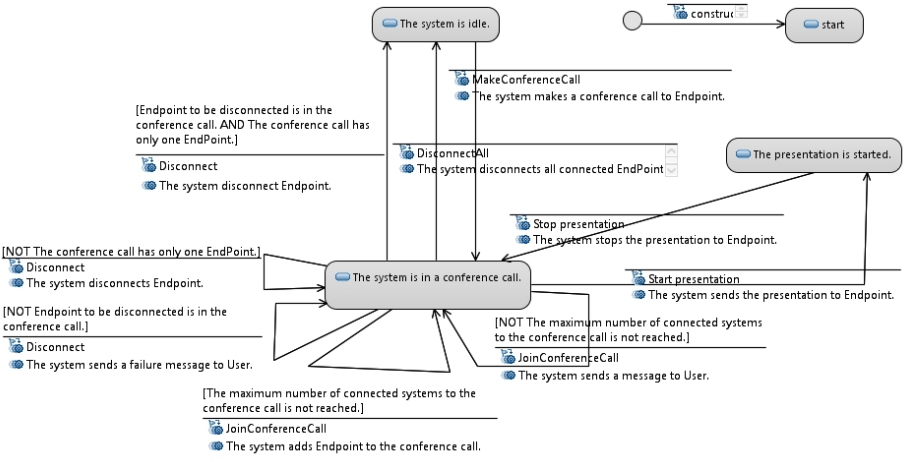


Fig. 2. Generated base state machine

Composite rule 2 invokes rules 2.1-2.4 to process each use case of the UCMod. Notice that we generate a single, system-level base state machine for the whole UCMod. We generate an instance of `State` for the precondition (Rule 2.1) and each postcondition (Rule 2.2), but making sure that no duplicate state is generated. Since the precondition of a use case indicates what must happen before the use case can start and the postconditions of the use case specify what must be satisfied after the use case completes, we define Rule 2.3 to generate a transition between the state derived from the precondition and the state derived from the postcondition of the basic flow. Rule 2.4 processes the postcondition of each alternative flow of the use case. Notice that RUCM enforces that each flow of events (both basic flow and alternative flows) of a UCS contains its own postcondition. This characteristic of RUCM makes the transformation (Rule 2.4) systematic. When generating transitions, how to determine the guard condition, trigger, and effect is described in [9], due to space limitation. Rule 3 processes each include relationship of the UCMod.

3.1.2 Generating Aspect State Machines

In this section, we describe the transformation for generating aspect state machines. Generated aspect state machines for two CUSs in Figure 1 are shown in Figure 3 and Figure 4, respectively. Notice that the base state machine has to be generated before the rules are applied to generate any aspect state machine. We generate an aspect state machine with stereotypes from AspectSM for each identified crosscutting behavior.

Rule 1 generates an instance of UML `StateMachine`, stereotyped with <<Aspect>>, which can then be visualized as a state machine diagram. Stereotype <<Aspect>> has two attributes: `name` and `baseStateMachine`, which represent the name of the aspect and the base state machine, on which the aspect is applied ([10]). The `name` of <<Aspect>> is the name of the included use case (extending use case) and the `baseStateMachine` should refer to the base state machine. Notice that we generate a single base state machine for the system. Therefore, the multiplicity of attribute `baseStateMachine` of <<Aspect>> is always 1. For example, both the name

of the aspect state machine and the name of <<Aspect>> are 'AdaptCallRate' when generating the aspect state machine for use case AdaptCallRate, and the aspect state machine refers to its base state machine (Figure 2), though this information is not shown in Figure 3. **Rule 2** generates the initial state (an instance of `Pseudostate`).

Rule 3 generates a state stereotyped with AspectSM stereotype <<Pointcut>>. A *pointcut* in AspectSM selects one or more joinpoints with similar properties, where advices can be applied [10]. Six attributes are defined for stereotype <<Pointcut>> in AspectSM: `name`, `type`, `selectionConstraint`, `beforeAdvice`, `afterAdvice`, and `aroundAdvice`. The `name` of the pointcut state and <<Pointcut>> is generated as string "SelectStates X', where X can be any number uniquely identifying a state in the aspect state machine. Notice that it is possible to have more than one state stereotyped with <<Pointcut>>. For example, as shown in Figure 3, the pointcut state `SelectedStates1` is generated for use case AdaptCallRate. Two pointcut states `SelectedStates1` and `SelectedStates2` are generated for use case CheckNetworkConnection.

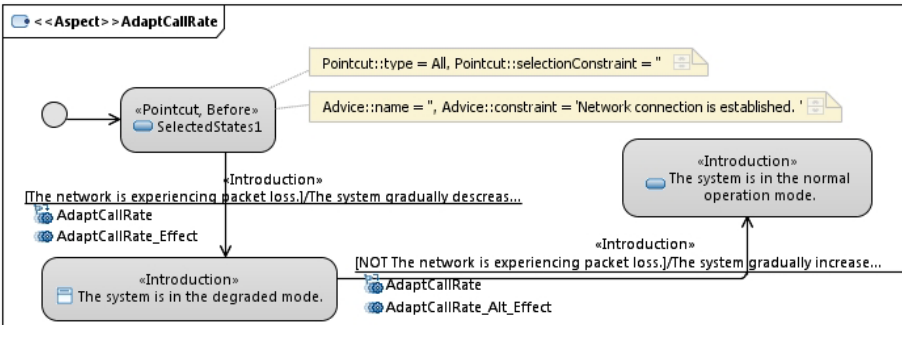


Fig. 3. Generated aspect state machine for use case AdaptCallRate

The `type` and `selectionConstraint` of the pointcut should be determined by the following three rules:

- If the CUS extends all the use cases or is included by all the use cases in the base state machine, then the `type` of the pointcut is 'All', and the `selectionConstraint` should be equal to empty. As shown in Figure 3, state `SelectedStates1` selects all the states in the base state machine through the <<Pointcut>> attribute `type`, since use case AdaptCallRate extends all the use cases except the other crosscutting use case CheckNetworkCondition. There is no point to assign a value to `selectionConstraints` as all the use cases are selected.
- If the CUS extends a subset of the use cases or is included by a subset of the use cases in the base state machine, then the `type` is 'Subset', and the `selectionConstraint` should be a list of the names of the states generated for the preconditions and postconditions of the use cases either including or being extended by the CUS in the base state machine. For example, as shown in Figure 4, state `SelectedStates1` selects a subset of the states in the base state machine: `The system is in a conference call.` and `The presentation`

`is started.` and `The system is idle.` (Figure 2). The reason of selecting these three states from the base state machine is that they were generated for either the preconditions or the postconditions of use cases MakeConferenceCall, JoinConferenceCall, and StartPresentation, which all include use case CheckNetworkConnection (Figure 1).

- If the CUS extends one use case or is included by one use case in the base state machine, then the type of the pointcut is 'One', and the selectionConstraint of the pointcut should be the states generated for the precondition and postconditions of the use case either including or being extended by the CUS in the base state machine.

Though there are three types of advices in AspectSM, in our transformation only `beforeAdvice` is used to introduce the precondition of the CUS as additional state invariants, through stereotype <<Before>>, to the selected states of the base state machine. As shown in Figure 3 and Figure 4, `beforeAdvice` is introduced to the state invariants of the selected states through state `SelectedStates1`. Notice that AspectSM has three types of advice (i.e., Before, After and Around). In our context, we only use Before, but we could have used After, which can bring the same semantics to the generated aspect state machines, as we discussed in our previous work [10] where AspectSM is discussed in details.

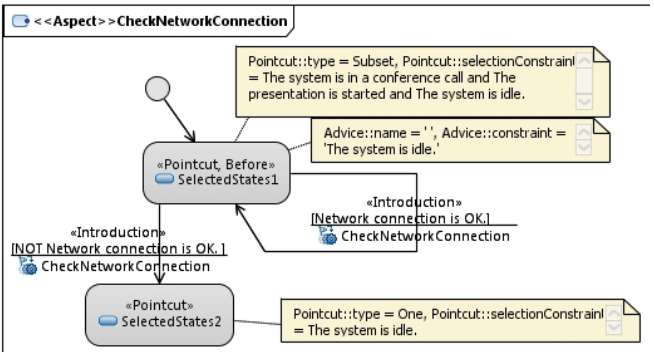


Fig. 4. Generated aspect state machine for use case CheckNetworkConnection

Rule 4 generates the transition (an instance of `Transition`) from the initial state to the pointcut state. **Rule 5** handles the postcondition of the basic flow of the CUS. There are two situations for applying this rule:

- If the CUS is an extending use case, generate a state for the postcondition, as long as such a state has not been generated in the base state machine. This newly generated state should be stereotyped with AspectSM stereotype <<Introduction>> showing that this state will be introduced in the base state machine. Otherwise, a pointcut state should be generated to point to the existing state in the base state machine. Rule 3 should be followed to generate values for the attributes of stereotype <<Pointcut>>. For example, as shown in Figure 3, state `The system is in a degraded mode.` is generated,

corresponding to the postcondition of the basic flow of use case AdaptCallRate (Table 1). Also generate a transition stereotyped with <<Introduction>> to connect the pointcut state either to the newly generated state or the pointcut state pointing to the existing state in the base state machine (Rule 5). The trigger of the transition should be the name of the CUS. The guard is the conjunction of all the conditions of the condition sentences in the basic flow of the use case. The effect of the transition should be the last step of the basic flow. For example, in Figure 3, a transition between `SelectedStates1` and `The system is in the degraded mode.` is generated with trigger `AdaptCallRate`, guard `The system is experiencing packet loss.`, and effect `The system gradually decreases conference call rate to the minimum call rate.`

- If the CUS is an included use case, a self-transition stereotyped with <<Introduction>> is generated for the pointcut state (Rule 3). The trigger of the transition should be the name of the included use case. The guard is the conjunction of all the conditions of the condition sentences in the basic flow of the CUS. The effect of the transition should be the last step of the basic flow. For example, as shown in Figure 4, a self-transition for `SelectedStates1` is generated, with trigger `CheckNetworkConnection` and guard `Network connection is OK`. Notice that when the last step of the basic flow of the use case is a condition check sentence (containing keyword VALIDATES THAT), we don't generate an effect for the transition. This is because condition check sentences are considered as representing system internal interactions [5].

Rule 6 handles alternative flows of the CUSs in a similar fashion as for base state machine (Rule 2.4, Table 3). The difference is that, when a new state is generated for the postcondition of an alternative flow, stereotype <<Introduction>> should be always applied; when an existing state is identified in the base state machine for the postcondition, then a pointcut state should be generated in the aspect state machine and it should point to the existing state in the state machine through attribute `selectionConstraint` of <<Pointcut>>. For example, as shown in Figure 4, a transition is generated between `SelectedStates1` and `SelectedStates2` as state `The system is idle.` has been generate as a state in the base state machine (Figure 2). In Figure 3, a state `The system is in the normal operation mode.` is generated and a transition between states `SelectedStates1` and `The system is in the normal operation mode`.

Rule 7 handles the situation when a CUS includes other use cases and these use cases are only connected to the CUS. In such case, Rule 2.3 and Rule 3 in Table 3, used for generating the base state machine, should be applied. As for Rule 7, the difference is that whenever a new state or transition is generated, stereotype <<Introduction>> should be applied and whenever an existing state is identified in base state machine, a poincut state should be generated.

3.2 Transition to State Machines for Automated Test Generation

The following steps should be followed to refine the generated base and aspect state machines so that they can be used as an input to automatically generate test cases.

1. The generated base and aspect state machines have to be manually refined by a user. More specifically, in the generated state machines, missing transitions and states should be added, extra states and transitions should be removed, and incorrect ones should be modified. For the generated aspect state machines, the user additionally has to refine elements related to AspectSM.
2. Add state invariants using the Object Constraint Language (OCL) [17] for each state of the generated state machines based on the actual state variables of a system.
3. Map all the triggers of all the state machine diagrams to the actual API calls of the SUT so that the API of the system can be invoked while executing test cases generated from the state machines.
4. Last, it is also required to replace textual guard conditions of the generated state machines with corresponding OCL constraints, based on the state variables and/or input parameters of the triggers associated with the guards.

4 Industrial Case Studies

Our goal here is to assess 1) whether the tool does generate system-level state machine diagrams based on UCMods, 2) whether our transformation rules are semantically complete, 3) whether our transformation rules lead to state machine diagrams that are syntactically correct, and 4) whether the automatically generated state machine diagrams can be refined by test engineers to support MBT with a reasonable effort. Regarding point 3, syntactic correctness means that a generated state machine diagram conforms to the UML 2.2 state machines notations. Regarding point 2, semantic correctness means that a generated state machine diagram correctly represents its UCMod; all the constructs that are related to the transformation in the UCMod are correctly transformed by following the transformation rules and no redundant model elements are generated.

Regarding the first three evaluation points, two large-scale, network-based and distributed systems, respectively from the communication domain and the maritime and energy sectors are used to evaluate our approach. One is a Video Conference System (VCS) and the other is a Subsea Oil Production System (SOPS).

VCS contains four endpoints, which are of the same functionalities. These functionalities are modeled as the same set of use cases. Each endpoint has 10 use cases; in total the whole system contains 40 use cases. The core functionality of the system 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. Each of the VCS endpoint is operated by a human actor. A timer is needed to periodically initiate the adaption of call rate. Eight crosscutting concerns (e.g., *Standby*, *Do Not Disturb*, *Noise Cancellation*) were specified and transformed into aspect state machines using our approach.

SOPSs are large-scale, integrated, distributed, and highly configurable systems of systems for managing exploitation of oil and gas production fields, with various field layouts ranging from single satellite wells to large multiple sites (more than 50 wells).

SOPS has four different types of systems, three of which are located above the sea level and the other is located in subsea. These systems have distinct functionalities and are connected through different types of communication media. Due to the reason that we had no access to all the requirements of these systems, we were not able to specify the UCSs of all the systems. Only 12 out of 65 representative use cases were specified. We modeled the following six crosscutting concerns (e.g., Operation Mode Exchange, Runtime Configuration, Data Update Mechanism Switch) using RUCM and they are transformed into aspect state machines using our approach.

In total, 14 aspect state machines were generated and we carefully examined them, and we could verify that the generated state machines were syntactically correct and mostly but not entirely semantically complete. Due to space limitation and confidential issues, we are not able to provide more detailed information about these two industrial case studies.

5 Related Work

We conducted a systematic literature review [18] on transformations of textual requirements into analysis models, represented as class, sequence, state machine, and activity diagrams. A carefully designed paper selection procedure in scientific journals and conferences from 1996 to 2008 and Software Engineering textbooks identified 20 primary studies (16 approaches). The method proposed here is based on the results of this review, with a focus on automatically deriving state machine diagrams from UCMods.

A series of methods is proposed in [19] (one of the primary studies of our systematic review [18]) to precisely capture requirements and then manually transform requirements into a conceptual model composed of object models (e.g., class diagrams), dynamic models (i.e., state machines and sequence diagrams), and functional diagrams. The approach does not purport to provide a solution for transforming requirements into analysis models. Instead, it proposes a set of techniques for users to precisely specify requirements and conceptual models, and also proposes a process to guide the users in deriving the conceptual models from the requirements. No transformation method is reported in the paper.

Somé [20], another primary study of our systematic review, proposes an approach to generate finite state machines from use cases in restricted Natural Language (NL). The approach requires the existence of a domain model. The domain model serves two purposes: a lexicon for the NL analysis of use cases, and the structural basis of the state transition graphs being generated. The domain model acts as the lexicon for NL analysis of the use cases, because the model elements of the domain model are used to document the use cases. For example, actors of the use cases refer to the classes of the domain model. Interactions between the system and the actors are defined as one type of use case operations (also including branching statements, use case inclusion statements) which correspond to class operations in the domain model. An algorithm is described in the paper to explain how to automatically transform the use cases plus the domain model into state machines. A working example is used to explain the approach. No case study is presented to evaluate the approach.

In summary, none of the existing approaches is able to fully and automatically generate either standard UML state machine diagrams or aspect state machines from requirements, which is what we are proposing in the paper.

6 Conclusion

The success of Model-based testing (MBT) relies on developing complete and precise input models. Especially to support modeling system non-functional behavior such as robustness and security, which is typically crosscutting functional behavior and thus modeling such behavior directly with functional behavior is not scalable since it leads to redundant and cluttered models. To cope with this issue, usually Aspect-oriented Modeling (AOM) is recommended to model crosscutting behavior. In this paper, we focused on a UML 2.0 profile (AspectSM), which supports comprehensive aspect modeling for UML 2.0 state machines (aspect state machines) and enables automated non-functional testing. As with other Aspect-Oriented Modeling (AOM) approaches, AspectSM can potentially offer several benefits such as: enhanced modularization, easier evolution of models, increased reusability, reduced modeling effort, and improved readability [11-13]. Developing such aspect state machines from scratch is a challenging task, especially when testers are not acquainted with modeling. To assist the initial modeling required for MBT, we propose an approach to transform use case specifications into UML state machines and aspect state machines.

A precise and rigorous use case modeling approach (RUCM) was proposed in [4] and was used in this paper, as part of aToucan [2, 3, 9], to automatically generate UML and aspect state machines from use cases. We evaluated our approach on two industrial case studies and we assessed the quality of generated base and aspect state machines and found them largely consistent. Our industry partners benefited not only from the executable test cases, but also from the system specification expressed as UML and aspect state machines and precise requirements expressed with RUCM. All these activities took no more than few hours, including documenting the use case model and refining the generated base and aspect state machines.

References

1. Shafique, M., Labiche, Y.: A Systematic Review of Model Based Testing Tool Support. Carleton University. Technical Report SCE-10-04
2. Yue, T., Briand, L.C., Labiche, Y.: Automatically Deriving a UML Analysis Model from a Use Case Model. Simula Research Laboratory. Technical Report 2010-15 (2010)
3. Yue, T., Briand, L.C., Labiche, Y.: An Automated Approach to Transform Use Cases into Activity Diagrams. In: Kühne, T., Selic, B., Gervais, M.-P., Terrier, F. (eds.) ECMFA 2010. LNCS, vol. 6138, pp. 337–353. Springer, Heidelberg (2010)
4. Yue, T., Briand, L.C., Labiche, Y.: A Use Case Modeling Approach to Facilitate the Transition towards Analysis Models: Concepts and Empirical Evaluation. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 484–498. Springer, Heidelberg (2009)

5. Yue, T., Briand, L., Labiche, Y.: Facilitating the Transition from Use Case Models to Analysis Models: Approach and Experiments. Accepted for publication in Transactions on Software Engineering and Methodology, TOSEM (2011)
6. Eclipse Modeling Framework (EMF), http://www.eclipse.org/modeling/emf/
7. The Stanford Natural Language Processing Group. The Stanford Parser version 1.6
8. Triskell team, http://www.kermeta.org/
9. Yue, T., Ali, S., Briand, L.: Automated Transition from Use Cases to UML State Machines to Support State-Based Testing. In: France, R.B., Kuester, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 115–131. Springer, Heidelberg (2011)
10. Ali, S., Briand, L., Hemmati, H.: Modeling Robustness Behavior Using Aspect-Oriented Modeling to Support Robustness Testing of Industrial Systems. Accepted for publication in the Journal of Software and Systems Modeling (2011)
11. Ali, S., Yue, T., Briand, L.: Empirically Evaluating the Impact of Applying Aspect State Machines on Modeling Quality and Effort. Simula Research Laboratory. Technical Report 2011-06 (2011)
12. Ali, S., Yue, T., Briand, L.: Does Aspect-Oriented Modeling Help Improve the Readability of UML State Machines? Simula Research Laboratory. Technical Report 2010-11 (2011)
13. Ali, S., Yue, T.: Comprehensively Evaluating Conformance Error Rates of Applying Aspect State Machines for Robustness Testing. In: ACM International Conference on Aspect-Oriented Software Development (AOSD)
14. 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)
15. Cisco Norway (Tandberg), http://www.tandberg.no/
16. FMC Technologies, http://www.fmctechnologies.com/
17. OMG: OCL 2.0 Specification. Final Adopted Specification
18. Yue, T., Briand, L.C., Labiche, Y.: A systematic review of transformation approaches between user requirements and analysis models. Requirements Engineering 16 (2011)
19. Insfrán, E., Pastor, O., Wieringa, R.: Requirements Engineering-Based Conceptual Modelling. In: Requirements Engineering, pp. 61–72
20. Some, S.S.: An approach for the synthesis of state transition graphs from use cases, pp. 456–462. CSREA Press

Badger: A Regression Planner to Resolve Design Model Inconsistencies

Jorge Pinna Puissant[1], Ragnhild Van Der Straeten[2,3], and Tom Mens[1]

[1] University of Mons, 20 Place du Parc, Mons, Belgium
{jorge.pinnapuissant,tom.mens}@umons.ac.be
[2] Vrije Universiteit Brussel, Brussel, Belgium
rvdstrae@vub.ac.be
[3] Université Libre de Bruxelles, Brussel, Belgium

Abstract. One of the main challenges in model-driven software engineering is to deal with design model inconsistencies. Automated techniques to detect and resolve these inconsistencies are essential. We propose to use the artificial intelligence technique of automated planning for the purpose of resolving software model inconsistencies. We implemented a regression planner in Prolog and validated it on the resolution of different types of structural inconsistencies for generated models of varying sizes. We discuss the scalability results of the approach obtained through several stress-tests and discuss the limitations of our approach.

Keywords: automated planning, inconsistency resolution, model, scalability.

1 Introduction

One of the main challenges in *model-driven software engineering (MDE)* is to deal with evolving models, and to provide automated mechanisms to support this evolution [23]. A particular point of attention is to manage inconsistencies in software models [20]. Such model inconsistencies are inevitable, because a software system's description is composed of a wide variety of diverse models, some of which are developed and maintained in parallel. Our research does not focus on the activity of model inconsistency *detection*, that has become well-established. Instead, we address the *resolution* of model inconsistencies. In particular, we focus on more automated ways to resolve a selection of previously identified model inconsistencies through the generation of so-called *resolution plans*.

To do this, we use the technique of *automated planning* [19] originating from the field of artificial intelligence. This technique allows the generation of possible resolution plans through an automated planner without the need of manually writing resolution rules. In [18] we used the progression planner called *FF* [7,8]. Using this planner in the context of inconsistency resolution suffers from various scalability problems and lack of expressiveness, making the approach unusable in practice. To address the aforementioned limitations we present here a new planner called *Badger*[1], a *regression planner* implemented in Prolog.

[1] The name *Badger* comes from the *honey badger*, an animal that is able to run backwards.

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 146–161, 2012.

This paper is structured as follows. Section 2 introduces the problem of model inconsistency resolution and presents a motivating example. Section 3 introduces automated planning. Section 4 explains the automated planner *Badger* that we implemented for resolving model inconsistencies. Its scalability to large models is assessed in Section 5. Section 6 discusses the threats to validity, Section 7 presents the related work and Section 8 concludes this paper.

2 Model Inconsistency Resolution

A wide variety of modeling languages, domain-independent as well as domain-specific, exists. As a consequence, there are many different kinds of, often interrelated, models that can suffer from many types of inconsistencies. In this paper the Unified Modeling Language (UML) is used to express design models because it is the de-facto general-purpose modeling language [17]. Its visual notation consists of a set of different diagram types, such as class diagrams, sequence diagrams and statecharts, each expressing certain aspects of a software system. These diagrams are interrelated and inconsistencies in and between them can arise easily.

In this article, we will restrict ourselves to the subset of the UML metamodel for class diagrams shown in Figure 1[2]. Table 1 lists a set of 13 structural model inconsistency types we will consider based on the elements occurring in this metamodel and on the well-formedness constraints of the UML 2.3 metamodel expressed in OCL [2,6,17,22,25]). Each entry in Table 1 consists of an id followed by the metamodel element on which the constraint is specified in the UML specification. Next, a short description of the inconsistency type is given, followed by the page number of the UML Superstructure document [17] where the inconsistency type can be found.

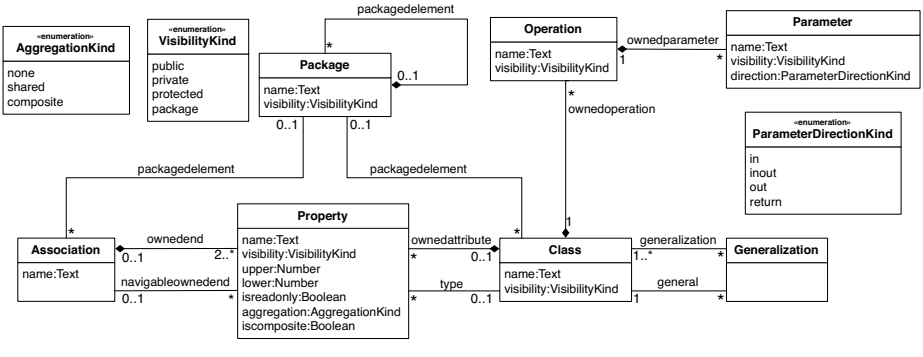


Fig. 1. Simplified fragment of the UML metamodel for class diagrams

[2] Because our approach relies on a metamodel independent representation, it can be used with other structural models as well.

Table 1. List of considered structural model inconsistency types

id	model element	description of the inconsistency type (see [17])
I_1	Association	Only binary associations can be aggregations (p. 39)
I_2	Element	Elements that must be owned must have an owner (p. 65)
I_3	Named Element	If a NamedElement is not owned by a Namespace, it does not have a visibility (p. 101)
I_4	Multiplicity Element	A multiplicity must define at least one valid cardinality that is greater than zero. (p. 97)
I_5	Multiplicity Element	The lower bound must be a non-negative integer literal (p. 97)
I_6	Multiplicity Element	The upper bound must be greater than or equal to the lower bound (p. 97)
I_7	Classifier	The general classifiers are the classifiers referenced by the generalization relationships (p. 54)
I_8	Classifier	Generalization hierarchies must be directed and acyclic. A classifier can not be both a transitively general and transitively specific classifier of the same classifier (p. 54)
I_9	Classifier	A classifier may only specialize classifiers of a valid type (p. 54)
I_{10}	Property	A multiplicity on an aggregate end of a composite aggregation must not have an upper bound greater than 1 (p. 127)
I_{11}	Property	Only a navigable property can be marked as readOnly (p. 128)
I_{12}	Property	The value of isComposite is true only if aggregation is composite (p. 128)
I_{13}	Operation	An operation can have at most one return parameter (p. 107)

Consider as a motivating example the simple class diagram shown in Figure 2. The diagram contains two structural inconsistency occurrences, one of type "I_{10}: Multiplicity composition constraint" and one of type "I_{13}: Operation return constraint" (see Table 1). The occurrence of I_{10} arises because the composite association (represented by a black diamond) between classes `Car` and `Wheel` has an upper multiplicity greater than 1 (namely 2) at the composite end, which is in contradiction with the fact that a part in the composition cannot be shared between multiple components. An occurrence of I_{13} occurs when an operation in a class returns more than one parameter. The operation `getDiameter():float,integer` of class `Wheel` has two return parameters.

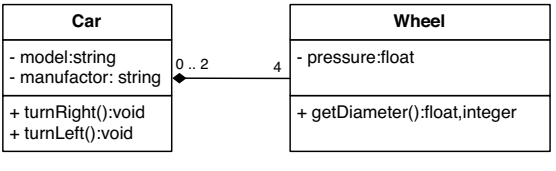


Fig. 2. Class diagram with 2 inconsistency occurrences

Each inconsistency occurrence can be resolved by several resolutions. For example, changing the upper multiplicity from 2 to 1 can resolve the occurrence of I_{10}, while replacing the composite association by a regular association resolves the inconsistency occurrence as well. The occurrence of I_{13} can be resolved by removing one of both return parameters or by changing one of the return parameters into an input parameter.

3 Automated Planning

Our aim is to tackle the problem of inconsistency resolution by generating possible resolutions without the need of manually writing resolution rules or writing any procedures that generate possible resolutions. The approach needs to enable the resolution of multiple inconsistency occurrences at once and to perform the resolution in a reasonable time. In addition, the approach needs to be generic, i.e., it needs to be easy to apply it to different modeling languages. In [24] we explored the usage of model finders for this purpose. In this article, we use *Automated Planning* instead.

Automated planning aims to generate plans, i.e., sequences of actions that lead from an initial state to a state meeting a specific predefined goal. Each planning approach consists of a *representation language* to describe the *problem domain*, a *problem*, an *algorithm* describing the mechanism to solve the problem, and a sequence of *generated plans* produced as output of the algorithm.

The *problem domain* (e.g., model inconsistency resolution) is expressed as a *set of possible actions* (e.g., to change a model). A *possible action* specifies a *valid* way to go from one state to another. The action is composed of a *precondition* that specifies the conditions that must hold in order for the action to be applicable, and an *effect* that specifies the changes to be made to the current state.

The *problem* that needs to be solved in the problem domain (e.g., an inconsistent model such as the one in Figure 2) is expressed by an *initial state* and a *desired goal*. The initial state represents the current state of the world (the inconsistent model). The desired goal is a partially specified state that describes the world that we would like to obtain (a consistent model).

A *generated plan* is a sequence of actions, generated automatically by the planning algorithm, to transform the initial state into a state that satisfies the desired goal.

Many *algorithms* exist to solve planning problems. A first approach consists in translating the problem and its domain into a satisfiability problem, and using a model checker or SAT solver to find a solution [9]. A second way consists in using a state space search algorithm. The state space can be traversed through *progression* planning or *regression* planning. Progression planning performs a forward search that starts in the initial state and tries to find a sequence of actions that reaches a goal state. Regression planning starts from the goal state and searches backwards to find a sequence of actions that reaches the initial state.

4 Badger

We have chosen to implement a regression planner, because it depends on the size of the desired goal and works only with relevant actions. A relevant action is an action that contributes to the achievement of the goal. The search space of a regression planner will be significantly smaller than the one of a progression

planner, as the latter depends mainly on the size of the initial state and does not exclude irrelevant actions.

We implemented the planner algorithm in Prolog, since Prolog's built-in backtracking mechanism allows the planner to easily generate multiple resolution plans among which the user can choose the most suitable one.

4.1 Problem and Problem Domain

The **initial state** is expressed as a conjunction of logic literals that represents the input model. We specify the models using Praxis [2], a language that represents models and model changes as sequences of elementary model operations (`create`, `addProperty`, `addReference`, `delNode`, `remProperty`, `remReference`). Praxis comes with a suite of Eclipse plugins: a plugin to reason about ECore and XMI models; a plugin to generate class diagram models of varying sizes [15]; and a model inconsistency detection engine [2]. As an example, the class *Car* of the class diagram model of Figure 2 is represented as follows[3]:

```
create(c1, class).
addProperty(c1, name, 'Car').
create(att1, property).
addProperty(att1, name,'model').
addReference(att1, type,string).
addReference(c1, ownedattribute, att1).
create(att2, property).
addProperty(att2, name,'manufactor').
addReference(att2, type,string).
addReference(c1, ownedattribute, att2).
create(op1, operation).
addProperty(op1, name,'turnRight').
addReference(c1, ownedoperation,op1).
create(op2, operation).
addProperty(op2, name,'turnLeft').
addReference(c1, ownedoperation,op2).
```

This way of representing models offers several advantages. The elementary model operations are metamodel independent, i.e., they can be used together with any kind of structural metamodel. The second parameter of each model operation refers to an element of the metamodel (e.g., `class`, `ownedattribute`, `parameter`, `ownedoperation`).

The **desired goal** is a partially specified state that represents the objective to be reached, namely the absence of model inconsistencies, as a negation of inconsistency occurrences. An inconsistency occurrence is detected if it matches the pattern defined by the inconsistency type. Table 2 presents all logic operators that are allowed to specify the desired goal, inspired by the list of common constructs found in inconsistency types [5,16,21]. Our approach does the strict

[3] In principle, each of the listed model operations should also have a timestamp that we have left out for the sake of readability.

minimum to accomplish this goal. For example, if the user wants to solve the inconsistency "the lower multiplicity must be greater than 0", *Badger* will proposes 1 as solution to avoid an infinite number of possibilities.

Table 2. Logic Operators. Although the operators value comparison, property comparison and counting are only shown with the $>$ function, the other comparison functions can be used as well : $<$, $\geq$, $\leq$, $=$, $\neq$

Name		
Negative literal	Syntax	not(P)
	Semantics	$\neg P$
	Example	not(lastAddProperty(ae2,iscomposite,'true'))
Conjunction	Syntax	[P, Q]
	Semantics	$P \wedge Q$
	Example	[lastAddProperty(c1,name,'Vehicle'), lastAddProperty(c2,name,'Aircraft')]
Disjunction	Syntax	or [P, Q]
	Semantics	$P \vee Q$
	Example	or [lastAddProperty(c1,name,'Vehicle'), lastAddProperty(c1,name,'Aircraft')]
Universal quantification	Syntax	forall(P,Q)
	Semantics	$\forall x(P(x) \Rightarrow Q(x))$
	Example	forall(lastCreate(X,class), lastAddProperty(X,name,Y))
Existential quantification	Syntax	exists(P)
	Semantics	$\exists x P(x)$
	Example	exists(lastCreate(X,class))
Value comparison	Syntax	compare(P,>,v)
	Semantics	$\forall n \in \mathbb{N}(P(n) \wedge v \in \mathbb{N} \wedge n > v)$
	Example	compare(lastAddProperty(ae1,lower_mult,X),>,0)
Property Comparison	Syntax	compare(P,>,Q)
	Semantics	$\forall n, m(P(n) \wedge Q(m) \wedge n > m)$
	Example	compare(lastAddProperty(ae1,upper_mult,X),>, lastAddProperty(ae1,lower_mult,Y))
Counting	Syntax	count(P,>,v)
	Semantics	$(\mid\{x \mid P(x)\}\mid > v \wedge v \in \mathbb{N})$
	Example	count(lastAddReference(assID,member,X),>,2)
Transitive Navigability	Syntax	nav(From, Kind, To)
	Semantics	$(Kind(From, To) \Rightarrow nav(From, Kind, To)) \vee$ $\exists c(nav(From, Kind, c) \wedge nav(c, Kind, To) \Rightarrow nav(From, Kind, To))$
	Example	nav(c1,generalization,c9)

As an example, the desired goal to resolve an inconsistency occurrence of type I_{10} is specified below as a negation of this inconsistency occurrence, using the logic operators of Table 2. It disallows the upper bound of the multiplicity on the aggregate end of a composite aggregation to be greater than 1.

```
or [not(lastAddProperty(prop1,aggregation,'composite')),
    not(compare(lastAddProperty(prop1,upper,X),>,1))]
```

The use of prefix `last` in model operation `lastAddProperty` is needed to point to those operations in the model that are not followed by other operations canceling their effects [2]. Using the negation of the inconsistency occurrences in the desired goal will only be able to resolve inconsistency occurrences that have already been identified previously. For this detection, we can rely on the detection approach proposed by [2].

A **possible action** specifies a *valid* way to go from one state to another. The action is composed of a *precondition* (`pre`) that specifies the conditions that must hold in order for the action to be applicable, and an *effect* (`eff`) that specifies which Praxis model operations will be added to the current state. The validity of an action (`can`) is verified by using a *metamodel* that imposes constraints on the model. The metamodel needed to validate the set of actions is specified as a set of logic facts in Prolog: fact `mme` represents the metamodel elements; `mme_property` represents the properties of the specified metamodel element and the kind of value that is used (*e.g.*, `text`, `boolean`, `int`); `mme_reference` represents the relationships between two metamodel elements, and the name that this relationship has. The metamodel used in this paper corresponds to the one shown in Figure 1.

The logic rules below specify the possible action `setProperty`. The `pre` rule states that the old property must exist before it can be changed. The `can` rule is used to verify that the new value is correctly typed and that is different from the old value. The `eff` rule expresses the two model operations changing the value of a property.

```
pre(setProperty(Id,MME,Property,OldValue,NewValue),
  [lastAddProperty(Id,Property,OldValue)]).

can(setProperty(Id,MME,Property,OldValue,NewValue)) :-
  mme_property(MME,Property,Type),
  call(Type,NewValue),
  NewValue \== OldValue.

eff(setProperty(Id,Property,OldValue,NewValue),
  [remProperty(Id,Property,OldValue),
    addProperty(Id,Property,NewValue)]).
```

4.2 The Algorithm

The *algorithm* used by *Badger* is based on the ones explained in [3]. *Badger* uses a *recursive best-first search (RBFS)* to recursively generate a state space and search for a solution in that state space. *RBFS* is a *best-first search* that explores the state space by expanding the most promising node. To do this the algorithm needs 3 functions: a *successor function*, an *evaluation function* and a *solution function*. The *successor function* generates the child nodes of a particular node, and is used to generate the state space. It strongly depends on the problem to be solved. The *evaluation function* f evaluates the child nodes to find the most promising one. It is defined as the sum of a *heuristic function* h and a *cost function* g: $f(n) = h(n) + g(n)$ where $h(n)$ is the minimal estimation of the cost to reach a solution from the node n, and $g(n)$ is the actual cost of the path to reach n. The *solution function* checks if a particular node is one of the solutions. These 3 functions are independent of the search algorithm, which means that we can also use other best-first search algorithms (e.g. A*, iterative-deepening A*, memory-bounded A*). We have chosen to use *RBFS* because it

only keeps the current search path and the sibling nodes along this path, making its space complexity linear in the depth of the search tree.

The *heuristic function* used by *Badger* is a known planner heuristic that *ignores the preconditions*. Every action becomes applicable in every state, and a single literal of the desired goal can be achieved in one step. Remember that the desired goal is a conjunction/disjunction of logic literals that represents one or more negations of inconsistency occurrences. This implies that the heuristic can be defined as the number of unsatisfied literals in the desired goal. The cost function used by *Badger* is the user-specified cost of applying each action. These costs affect the order in which the plans are generated. The user can, for example, give more importance to actions that add and modify model elements than to actions that delete model elements.

The *solution function* used by *Badger* checks if there are no more unsatisfied literals in the desired goal.

The *successor function* is the most complex one and is at the heart of the planning algorithm and proceeds as follows: (i) select a logic operator from the desired goal and generate a literal that satisfies this operator; (ii) analyse the effect (the `eff` rule) of each action to find one that achieves this literal; (iii) validate (the `can` rule) if the selected action can be executed; (iv) protect the already satisfied literals by checking if the execution of the selected action does not undo a previously satisfied literal; (v) regress goals through actions by adding the preconditions of the action (the `pre` rule) as new literals in the goal and by removing the satisfied literals from the goal.

4.3 Generated Plans

The *generated plans* produce a sequence of actions that transform the initial, inconsistent model into a model that does not have any of the inconsistency occurrences specified in the desired goal. Moreover, the generated resolution plans do not lead to ill-formed models (that do not conform to their metamodel) as long as all metamodel constraints are given as part of the problem specification.

Two complete resolution plans, each containing only two actions, that solve the inconsistency occurrences of the motivating example are given below:

```
1. setProperty(pro1,upper,2,1)
2. setProperty(par1,direction,‘return’,‘in’)

1. setProperty(pro1,aggregation,‘composite’,‘none’)
2. delNode(par1, parameter)
```

If we unfold the *effects* of each action from a resolution plan, we obtain a sequence of elementary Praxis model operations, that can be applied directly to transform the inconsistent model into a consistent one. For the first plan above, this sequence of operations looks as follows:

```
1. remProperty(pro1,upper,2)
2. addProperty(pro1,upper,1)
```

```
3. remProperty(par1,direction,'return')
4. addProperty(par1,direction,'in')
```

The number of actions proposed to resolve an inconsistency occurrence involving the modification of a reference in the desired goal depends on the size of the initial state (*i.e.*, it depends on the number of model elements). This negatively affects the performance of the algorithm and the number of generated resolution plans. To avoid generating many resolution plans that each refer to a concrete model element (e.g., one of the many classes in a class diagram), we introduced the notion of *temporal elements* as an abstraction of such a set of concrete model elements. A temporal element is represented as a tuple `(+other,X,Y)` where `X` is the model element type (e.g. `class`) and `Y` is the set of model elements of this type that cannot be used as part of the proposed resolution. Once the resolution plan is generated, the user can replace the temporal element by a concrete element that does not belong to `Y`, to avoid re-introducing the same inconsistency occurrence.

In order to assess whether *Badger* generates meaningful resolution plans, we manually verified all plans (between 3 and 10) generated for 5 very small class diagram models. The plans corresponded to what we expected, though we did not always agree with the order in which the plans were generated. By modifying the *cost function* $g(n)$, however, we can easily adapt the order according to the user's preferences. As will be discussed in section 6, carrying out a controlled user study with *Badger* to determine the most suitable order of the generated resolution plans is left as future work. In the next section, we report on the scalability of *Badger* for resolving structural model inconsistencies.

5 Scalability Study

Due to the unavailability of a sufficiently large sample of realistic UML models, we make use of an existing model generator that was proposed, mathematically grounded and validated in [15]. This model generator enables us to study the impact of the size of the models on the approach. It also enables us to apply our approach to a large set of models with a wide range of different sizes.

We used the model generator to create 941 models with model sizes ranging from 21 to 10849 model elements (i.e., elements obtained using the Praxis elementary operation `create`). Obviously, the generated models also contain references (from 21 to 11504) and properties (from 40 to 22903), obtained using the elementary operations `addProperty` and `addReference`, respectively.

These experiments were carried on a 64-bit computer with 2.53GHz Intel Core 2 Duo processor and 4Gb RAM. We used the 64-bit version of SWI-Prolog 6.0.2, running on the Ubuntu 11.04 operating system. All timing results obtained were averaged over 10 different runs to account for performance variations.

5.1 Experimental Results

In a first experiment, we have run *Badger* on all generated models and computed the timing results for generating a single resolution plan. We analysed

the relation between the number of model elements and the time (in seconds) needed to resolve only one inconsistency occurrence of a particular type. In order to compare the timing results for different inconsistency types, we repeated the experiment for each of the 13 considered inconsistency types shown in Table 1.

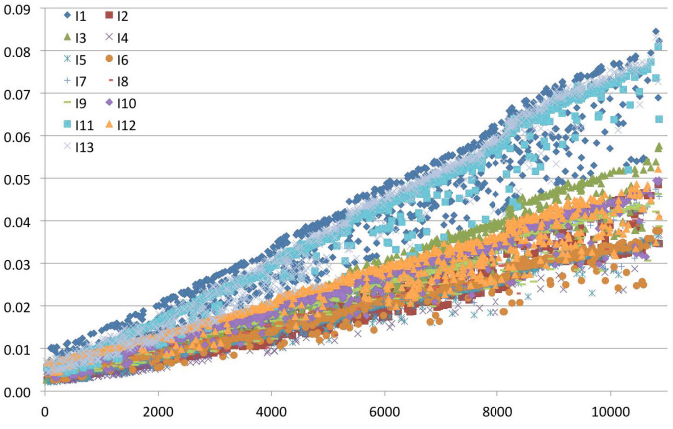


Fig. 3. Comparison of execution time (y-axis, expressed in seconds) per model size (x-axis, expressed as number of model elements) for resolving a single inconsistency occurrence in 941 different models. Different colours and symbols represent different inconsistency types.

The results of the experiment are visualised in Figure 3. The time needed to resolve occurrences of a particular inconsistency type mainly depends on the size of the model and on the number of logic literals in the desired goal. For example, I_{13} requires 4 literals and takes on average 4.2 times longer than I_5 that only uses 1 literal.

We fitted four different types of parametric regression models with 2 parameters to the data: a linear model, a logarithmic model, a power model and an exponential model. The goodness of fit of each type of model was verified using the coefficient of determination R^2. Its value is always between 0 and 1, and a value close to 1 corresponds to a good fit. Table 3 shows the obtained R^2 values. In order to easily distinguish the best regression models, values higher than 0.90 are indicated in *italics*, while values higher than 0.95 are indicated in **boldface**. In addition, per inconsistency type the regression models with the highest R^2 value are marked with (*).

By analysing Table 3 we observe that the logarithmic regression models provide the worst results. In contrast to the three other considered types of regression models, its R^2 values are always lower than 0.8. For these reasons, we exclude this type of regression model from the remainder of the analysis of our results. Based on the R^2 values the linear models appear to be the best in all cases (with an $R^2 > 0.95$ in 10 out of 13 cases). A visual interpretation also confirms that the linear models are the best match. The exponential and power models are also very good fits, with R^2 values that are always close to or above 0.9.

Table 3. R^2 values of four different parametric regression models used to fit the timing results

	Linear $y = a + b\ x$	Log $y = a + b\ \ln(x)$	Power $y = a\ x^b$	Exponential $y = a\ e^{b\ x}$
I_1	*0.935* (*)	0.741	0.891	0.892
I_2	*0.930* (*)	0.668	0.872	*0.927*
I_3	**0.950** (*)	0.687	0.888	*0.910*
I_4	**0.981** (*)	0.789	*0.934*	*0.902*
I_5	**0.987** (*)	0.767	*0.933*	*0.923*
I_6	**0.975** (*)	0.764	*0.918*	*0.926*
I_7	**0.975** (*)	0.737	0.898	*0.943*
I_8	**0.965** (*)	0.686	0.843	*0.935*
I_9	**0.975** (*)	0.736	0.894	*0.941*
I_{10}	**0.975** (*)	0.778	*0.936*	*0.907*
I_{11}	**0.981** (*)	0.752	*0.929*	*0.908*
I_{12}	*0.942* (*)	0.754	*0.906*	*0.905*
I_{13}	**0.977** (*)	0.718	0.888	*0.923*

In a second experiment, we studied how the generation of resolution plans with *Badger* scales up when resolving multiple inconsistencies *of different types* together. For each considered model, we resolved together one occurrence of each of the 13 inconsistency types. Because not all models have at least one occurrence of inconsistency type I_8, during our analysis we distinguished between models containing 12 inconsistency occurrences (excluding I_8) and models containing 13 occurrences.

Figure 4 (top part) presents the results of this experiment. The resolution time only increases slightly as the model size increases. None of the fitted regression models provide an R^2 value higher than 0.25. The execution time is lower for 12 inconsistency occurrences (mean = 0.268, median = 0.265) than for 13 occurrences (mean = 0.341, median = 0.336). Another factor that determines the execution time is the number of actions in the resolution plan. For resolving 12 inconsistency occurrences, we require between 8 and 11 actions (median =10), while for 13 occurrences we need between 9 and 12 actions (median = 11). In addition, the resolution time increases as the number of actions increases, as shown in the box plots of Figure 4 (bottom part).

In a third experiment, we studied how the generation of a resolution plan with *Badger* scales up if we want to resolve multiple inconsistency occurrences *of the same type* together. To test this, we generated a very large model containing more than 10,000 elements and a large number of inconsistency occurrences of each type. We excluded inconsistency type I_8 because the generated model did not contain enough occurrences of this type. For each of the remaining 12 inconsistency types we computed the time required to resolve an increasing number of occurrences (ranging from a single one to 70). Figure 5 visualizes the results. Given the rapid increase of execution time as the number of inconsistency occurrences increases, we fitted quadratic models (second degree polynomial), power models and exponential models to the data. The *adjusted* R^2 (to account for a different number of parameters in the regression models) was very high for the 3 types of models. The quadratic models had the best fit, with an *adjusted* $R^2 > 0.95$ in all cases, followed by the exponential models (> 0.92 in all cases,

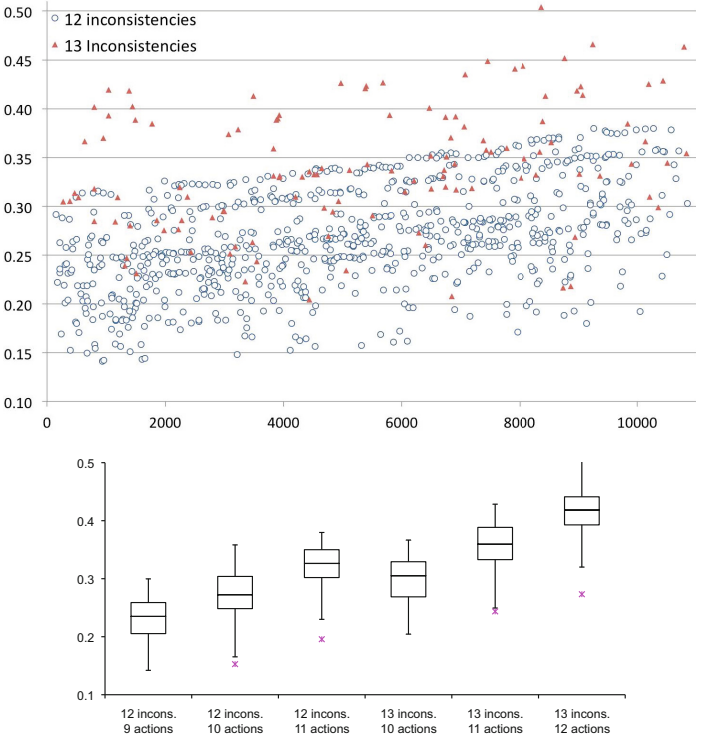


Fig. 4. Top: Time comparison (y-axis, in seconds) per model size (x-axis, in number of model elements) for resolving multiple inconsistencies of different types in 941 different models. **Bottom:** Boxplots showing effect of number of actions on execution time.

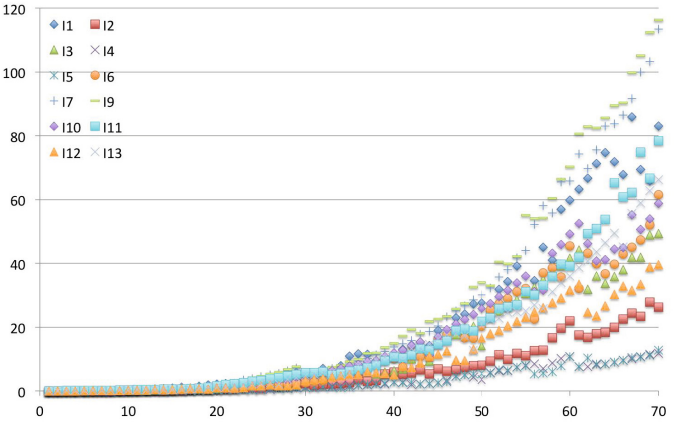


Fig. 5. Execution time (y-axis, in seconds) per number of inconsistency occurrences of the same type (x-axis) for resolving multiple inconsistency occurrences in a very large model. Different colours and symbols represents different inconsistency types.

and > 0.95 in 5 out of 12 cases). The different growth rates observed in Figure 5 reflect the complexity of the inconsistency type to be resolved. For example, the inconsistency types whose resolution only requires the change of property values take less time than those that need changes to references between model elements, because of the additional multiplicity constraints required for the latter.

6 Threats to Validity

Our approach has only been stress-tested on class diagram models. However, the fact that we rely on a metamodel independent representation (using sequences of Praxis elementary model operations) makes it straightforward to apply it to other types of structural models as well. We considered only a limited set of inconsistency types, but tried to include a variety of different expressions and elementary model operations. It remains an open question whether and how the approach can be generalised to non-structural inconsistencies and models.

The regression planner we implemented for doing our experiments may still contain some bugs we are not aware of. To carry out our experiments, we relied on an external model generator [15]. This may cause a bias as the generated models may not look like "real" models. This bias is limited since the model generator relies on the Boltzmann random sampling method that generates, in a scalable way, uniform samplings of any given size.

Our planner generates all possible resolution plans one after another, thanks to Prolog's backtracking mechanism. In this article we only evaluated the scalability for generating a single plan on a wide variety of models containing a wide range of different inconsistencies. In order to make the approach useful in practice, the resolution that the user actually prefers should be one of the first generated resolution plans. The order in which resolution plans are generated can be modified easily by modifying the cost function of the planner algorithm, as explained in section 4. In addition, entire resolution plans can be omitted by attaching an infinite weight to certain actions. Assessing what would be the most suitable parameters for the cost function in practice requires a controlled user study, which is left as future work.

7 Related Work

Several approaches have been proposed to resolve model inconsistencies. In our previous work [14] we specified resolution rules manually, which is an error-prone process. Automatic generation of inconsistency resolution actions aims to resolve this problem. Nentwich *et al.* [16] achieve this by generating resolution actions automatically from the inconsistency types. The execution of these actions, however, only resolves one inconsistency occurrence at a time. As recognised by the authors, this causes problems when inconsistency occurrences and their resolutions are interdependent. Mens *et al.* [13] propose a formal approach based on graph transformation to analyse these interdependencies.

Xiong *et al.* [25] define a language to specify inconsistency rules and the possibilities to resolve the inconsistencies. This requires inconsistency rules to be annotated with resolution information. Almeida da Silva *et al.* [1] propose an approach to generate resolution plans for inconsistent models, by extending inconsistency detection rules with information about the causes of the inconsistency, and by using manually written functions that generate resolution actions. In both approaches inconsistency detection rules are polluted with resolution information.

Instead of explicitly defining or generating resolution rules, a set of models satisfying a set of consistency rules can be generated and presented to the user. Egyed *et al.* [6] define such an approach for resolving inconsistency occurrences in UML models. Given an inconsistency occurrence and using choice generation functions, their approach generates possible resolution choices, i.e., possible consistent models. The choice generation functions depend on the modeling language, i.e., they take into account the syntax of the modeling language, but they only consider the impact of one consistency rule at a time. Furthermore these choice generation functions need to be implemented manually.

In [24] we use *Kodkod*, a SAT-based constraint solver using relational logic, for automatically generating consistent models. While the approach guarantees correctness and completeness (within the considered lower and upper bounds of the relations defined in the problem), a major limitation is its poor performance and lack of scalability.

Küster and Ryndina [11] introduce the concept of side-effect expressions to determine whether or not a resolution introduces a new inconsistency occurrence. They attach a cost to each inconsistency type to compare alternative resolutions for the same inconsistencies. Other authors also use the automatic resolution to solve different kinds of software engineering problems. For example, Jose *et al.* [10] present an algorithm based on a reduction to the maximal satisfiability problem, to automatically locate a set of potential cause of error in C programs. Demsky and Rinard [4] present an approach to automatically detect and resolve errors in data structures. Mani *et al.* [12] collect runtime information for the failing transformation in a model transformation program, and compute repair actions for the input model.

8 Conclusion

In this article we used automated planning, a logic-based approach originating from artificial intelligence, for the purpose of model inconsistency resolution. We are not aware of any other work having used this technique for this purpose.

We implemented a regression planner in Prolog. It requires as input a model and a set of inconsistency occurrences. In contrast to other inconsistency resolution approaches, the planner does not require the user to specify resolution rules manually or to specify information about the causes of the inconsistency. To specify models in a metamodel-independent way, and to be able to reuse an existing model generator, we relied on the Praxis language [2].

We have stress-tested our approach on 941 automatically generated UML class diagram models of varying sizes using a set of 13 structural inconsistency types based on OCL constraints found in the UML metamodel specification. Our approach for resolving inconsistency occurrences appears to be linear in the size of the model, and scales up to models containing more than 10000 model elements. The execution time also increases as the number of actions in the resolution plan increases. With respect to the number of inconsistency occurrences, the approach is quadratic in time. However, controlled user studies are still needed to adapt the cost function and evaluate the preferred order of the generated resolution plans.

Acknowledgments. This work has been partially supported by (i) the F.R.S. – FNRS through FRFC project 2.4515.09 "Research Center on Software Adaptability"; (ii) research project AUWB-08/12-UMH "Model-Driven Software Evolution", an *Action de Recherche Concertée* financed by the *Ministère de la Communauté française - Direction générale de l'Enseignement non obligatoire et de la Recherche scientifique, Belgium*; (iii) the Interuniversity Attraction Poles Programme – Belgian State – Belgian Science Policy.

References

1. Almeida da Silva, M.A., Mougenot, A., Blanc, X., Bendraou, R.: Towards Automated Inconsistency Handling in Design Models. In: Pernici, B. (ed.) CAiSE 2010. LNCS, vol. 6051, pp. 348–362. Springer, Heidelberg (2010)
2. Blanc, X., Mougenot, A., Mounier, I., Mens, T.: Detecting model inconsistency through operation-based model construction. In: Proc. Int'l Conf. Software Engineering, vol. 1, pp. 511–520 (2008)
3. Bratko, I.: Prolog programming for artificial intelligence. Addison-Wesley (2001)
4. Demsky, B., Rinard, M.C.: Automatic detection and repair of errors in data structures. In: Int'l Conf. on Object Oriented Programming, Systems, Languages and Applications, pp. 78–95. ACM (2003)
5. Egyed, A.: Automatically detecting and tracking inconsistencies in software design models. IEEE Trans. Software Eng. 37(2), 188–204 (2011)
6. Egyed, A., Letier, E., Finkelstein, A.: Generating and evaluating choices for fixing inconsistencies in UML design models. In: Proc. Int'l Conf. Automated Software Engineering, pp. 99–108. IEEE (2008)
7. Hoffmann, J.: FF: The Fast-Forward Planning System. The AI Magazine (2001)
8. Hoffmann, J., Nebel, B.: The FF Planning System: Fast plan generation through heuristic search. Journal of Artificial Intelligence Research 14, 253–302 (2001)
9. Jiménez Celorrio, S.: Planning and Learning under Uncertainty. PhD thesis, Universidad Carlos III de Madrid (2010)
10. Jose, M., Majumdar, R.: Cause clue clauses: error localization using maximum satisfiability. In: Proc. Conf. on Programming Language Design and Implementation, pp. 437–446. ACM (2011)
11. Küster, J.M., Ryndina, K.: Improving Inconsistency Resolution with Side-Effect Evaluation and Costs. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 136–150. Springer, Heidelberg (2007)

12. Mani, S., Sinha, V.S., Dhoolia, P., Sinha, S.: Automated support for repairing input-model faults. In: Int'l Conf. on Automated Software Engineering, pp. 195–204. ACM (2010)
13. Mens, T., Van Der Straeten, R.: Incremental Resolution of Model Inconsistencies. In: Fiadeiro, J.L., Schobbens, P.-Y. (eds.) WADT 2006. LNCS, vol. 4409, pp. 111–126. Springer, Heidelberg (2007), doi:10.1007/978-3-540-71998-4_7
14. Mens, T., Van Der Straeten, R., 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)
15. Mougenot, A., Darrasse, A., Blanc, X., Soria, M.: Uniform Random Generation of Huge Metamodel Instances. In: Paige, R.F., Hartman, A., Rensink, A. (eds.) ECMDA-FA 2009. LNCS, vol. 5562, pp. 130–145. Springer, Heidelberg (2009)
16. Nentwich, C., Emmerich, W., Finkelstein, A.: Consistency management with repair actions. In: Proc. 25th Int'l Conf. Software Engineering, pp. 455–464. IEEE Computer Society (May 2003)
17. Object Management Group. Unified Modeling Language: Superstructure version 2.3. formal/2010-05-05 (May 2010)
18. Pinna Puissant, J., Mens, T., Van Der Straeten, R.: Resolving model inconsistencies with automated planning. In: 3rd Workshop on Living with Inconsistencies in Software Development. CEUR Workshop Proceeding (September 2010)
19. Russell, S., Norvig, P.: Artificial Intelligence: A Modern Approach, 3rd edn. Prentice-Hall (2010)
20. Spanoudakis, G., Zisman, A.: Inconsistency management in software engineering: Survey and open research issues. In: Handbook of Software Engineering and Knowledge Engineering, pp. 329–380. World Scientific (2001)
21. Van Der Straeten, R.: Inconsistency management in model-driven engineering: an approach using description logics. PhD thesis, Vrije Universiteit Brussel (2005)
22. Van Der Straeten, R., Mens, T., Simmonds, J., Jonckers, V.: Using Description Logic to Maintain Consistency between UML Models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 326–340. Springer, Heidelberg (2003)
23. Van Der Straeten, R., Mens, T., Van Baelen, S.: Challenges in Model-Driven Software Engineering. In: Chaudron, M.R.V. (ed.) MODELS 2008. LNCS, vol. 5421, pp. 35–47. Springer, Heidelberg (2009)
24. Van Der Straeten, R., Pinna Puissant, J., Mens, T.: Assessing the Kodkod Model Finder for Resolving Model Inconsistencies. In: France, R.B., Kuester, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 69–84. Springer, Heidelberg (2011)
25. Xiong, Y., Hu, Z., Zhao, H., Song, H., Takeichi, M., Mei, H.: Supporting automatic model inconsistency fixing. In: Proc. ESEC/FSE 2009, pp. 315–324. ACM (2009)

Aspect-Oriented Modeling of Mutual Exclusion in UML State Machines

Gefei Zhang*

arvato systems Technologies GmbH
gefeizhang@acm.org

Abstract. Mutual exclusion is a very common requirement in parallel systems. Yet its modeling is a tedious task in UML state machines, one of the most popular languages for behavior modeling. We present HiLA, an aspect-oriented extension of UML state machines, to address this problem. In HiLA, mutual exclusion can be modeled in a highly modular and declarative way. That is, the logic of mutual exclusion is modeled at a dedicated place rather than by model elements scattered all over the state machine, and the modeler only needs to specify which states to mutually exclude rather than how to exclude them.

1 Introduction

UML state machines [9] are widely used for modeling software behavior. They are considered as simple and intuitive, and even deemed to be "the most popular modeling language for reactive components" [3]. Actually, UML state machines also exhibit some modularity problems. In particular, modeling synchronization of parallel regions, e.g. mutual exclusion of states, is often a tedious task using plain UML. The synchronization logic has to be specified imperatively, the involved model elements are often scattered all over the model, thus the resulting state machine is hard to read and prone to error, for an example see [13]. Due to the popularity of UML state machines as well as the importance of parallelism and synchronization, it is desirable to enhance UML state machines by language constructs which allow mutual exclusion to be modeled in a modularized and intelligible way.

Given the cross-region nature of parallelism, Aspect-Oriented Modeling is a promising paradigm for modeling region synchronization in UML state machines. The language of High-Level Aspects (HiLA, [11]) is an aspect-oriented UML extension, which improves the modularity of state machines considerably. Compared with other approaches of aspect-oriented state machines, such as [1,2,7,8], the distinguishing feature of HiLA is that HiLA aspects are *semantic*. That is, HiLA aspects are defined as modification of the *behavior* of the base machine rather than its (abstract) *syntax*. This way, the modeler only needs to specify *what* to do instead of *how* to do it.

* Sponsored by Ludwig-Maximilians-Universität München and the EU project ASCENS, 257414.

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 162–177, 2012.

The semantic approach of HiLA is also valuable for modeling region synchronization. In previous work [11,13], we showed how HiLA supports highly modular modeling of *passive waiting*, i.e., the region about to enter a critical state passively waiting for the "blocking" state (in another region) to be left *ordinarily*, as designed in the base machine. In this paper, we extend HiLA to cover another kind of mutual exclusion, which we call *active commanding*, where the blocking state is deactivated immediately, so that the waiting region can enter its critical state at once. Both strategies have realistic use cases, see e.g. [13,14] In both cases, using HiLA reduces the complexity of mutual-exclusion modeling considerably.

The rest of this paper is organized as follows: in the following Sect. 2 we give an overview of syntax and semantics of UML state machines, and show why the support for mutual-exclusion modeling provided by plain UML is insufficient. A brief overview of HiLA is given in Sect. 3.1, before in Sect. 3.2 the HiLA solution of modeling mutual exclusion is presented. Our implementation of HiLA aspects (weaving) is outlined in Sect. 4. Finally, we discuss some related work and draw conclusions.

2 UML State Machines

A UML state machine provides a model for the behavior of an object or component. Figure 1 shows a state machine modeling (in a highly simplified manner) the behavior of a player during a part of a game.[1] The player—a magician—starts in a state where she has to chose a NewLevel. Upon completion of the preparations she is transferred into the Play state which contains two concurrent regions, modeling two different *concerns* of the magician's intelligence. The upper region describes her possible movements: in each level the player initially starts in an entrance hall (Hall), from there she can move to a room in which magic crystals are stored (CrystalRoom) and on to a room containing a Ladder. From this room the player can either move back to the hall or, after fighting with some computer figure and winning, exit the level.

The lower region specifies the magician's possible behaviors. She may be Idle, gathering power for the next fight, Spelling a hex, or Fighting. She may escape from the fight and try to spell another hex, or, if she wins the fight in the Ladder room, wins the level and move on to another level. Any time while Playing, she can leave the game and quit.

2.1 Syntax and Informal Semantics

We briefly review the syntax and semantics of UML state machines according to the UML specification [9] by means of Fig. 1. A UML state machine consists of *regions* which contain *vertices* and *transitions* between vertices. A vertex is either a *state*, which may show hierarchically contained regions; or a *pseudo state* regulating how transitions are compound in execution. Transitions are triggered by

[1] This example is inspired by [15].

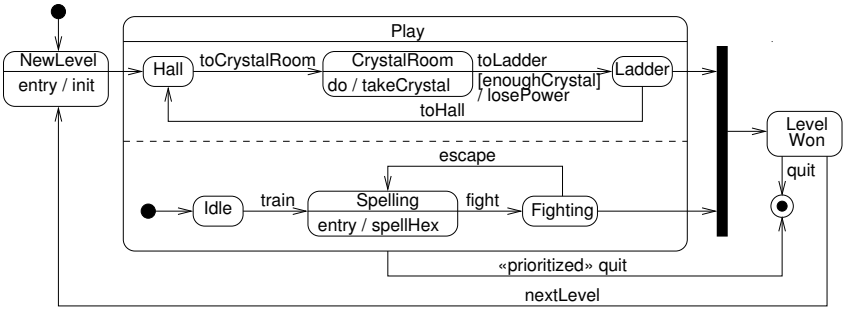


Fig. 1. Example: UML state machine

events and describe, by leaving and entering states, the possible state changes of the state machine. The events are drawn from an *event pool* associated with the state machine, which receives events from its own or from different state machines.

A state is *simple*, if it contains no regions (such as NewLevel in Fig. 1); a state is *composite*, if it contains at least one region; a composite state is said to be *orthogonal* if it contains more than one region, visually separated by dashed lines (such as Play). A region may also contain state and other vertices. A state, if not on the top-level itself, must be contained in exactly one region. A composite state and all the states directly or recursively contained in it thus build a tree.

Each state may[2] show an *entry* behavior (like spellHex in Spelling) and an *exit* behavior (not shown here), which are executed on activating and deactivating the state, respectively; a state may also show a *do activity* (like in CrystalRoom), which is executed while the state machine sojourns in this state. Transitions are triggered by events (toCrystalRoom, fight), show guards (enoughCrystal), and specify effects to be executed when a transition is fired (losePower). Completion transitions (e.g., the transition leaving NewLevel) are triggered by an implicit *completion event* emitted when a state completes all its internal activities. Events may be *deferred* (not shown here), that is, put back into the event pool if they are not to be handled currently. By executing a transition its source state is left and its target state entered; transitions may also be declared to be *internal* (not shown), thus skipping the activation-deactivation scheme. An *initial* pseudo state, depicted as a filled circle, represents the starting point for the execution of a region. A *final* state, depicted as a circle with a filled circle inside, represents the completion of its containing region; if all top-level regions of a state machine are completed then the state machine terminates. Transitions to and from different regions of an orthogonal composite state can be synchronized by *fork* (not shown here) and *join* pseudo states, presented as bars. For simplicity, we omit the other pseudo state kinds (entry and exit points, shallow and deep history, junction, choice, and terminate).

[2] In the following, we require that each state does have an entry and an exit action, which, however, may be NOP, doing nothing.

At run time, states get activated and deactivated as a consequence of transitions being fired. The active states at a stable step in the execution of the state machine form the active *state configuration*. Active state configurations are hierarchical: when a composite state is active, then exactly one state in each of its regions is also active; when a substate of a composite state is active, so is the containing state too. The execution of the state machine can be viewed as different active state configurations getting active or inactive upon the state machine receiving events. Note that for any given region, at most one direct substate of the region can be active at any given time, because a state configuration can contain at most one direct substate of the region.

For example, an execution trace, given in terms of active state configurations, of the state machine in Fig. 1 might be (NewLevel), (Play, Hall, Idle), (Play, Hall, Spelling), (Play, Hall, Fighting), (LevelWon), followed by the final state, which terminates the execution.

Note that events received by a state machine are stored in an *event pool*. The UML Specification [9] does not enforce any order to dispatch the events. The concrete dispatching strategy is deliberately delegated to the concrete implementation. Events may be prioritized, though a standard notation for priorities is not defined. In HiLA, we assume the concrete implementation to be able to handle two priorities: high and normal, and notate high-priority events with stereotype ≪prioritized≫. For example, in Fig. 1, when an event quit is received, and state Play is active, then the event is handled immediately (and the game is over), even though there may be other events waiting in the pool.

2.2 Mutual Exclusion in UML State Machines

Mutual exclusion is a very common feature in parallel systems. Yet it is fairly difficult to model in plain UML.

To prevent two states (from two different regions, see above) from being simultaneously active, we can actually distinguish two strategies: passive waiting and active commanding. Suppose states s_1 and s_2 are to be mutually excluded, the containing region being r_1 and r_2, $r_1 \neq r_2$, respectively. Suppose s_1 is active, and, if not for the mutual exclusion requirement, the current event would now cause a transition to be fired and, as a consequence, s_2 would be activated. Now we can actually apply two strategies to achieve mutual exclusion:

- In passive waiting, region r_2 would wait passively for s_1 to become inactive. Mutual exclusion is achieved by delaying an ordinary transition of the original state machine. The waiting region has no influence on the time it has to wait.
- In active commanding, an additional event (not available in the original state machine) is sent, so that s_1 will immediately and be left, and s_2 is activated right thereafter. This strategy activates s_2 as soon as possible, interrupting whatever s_1 may be undertaking.

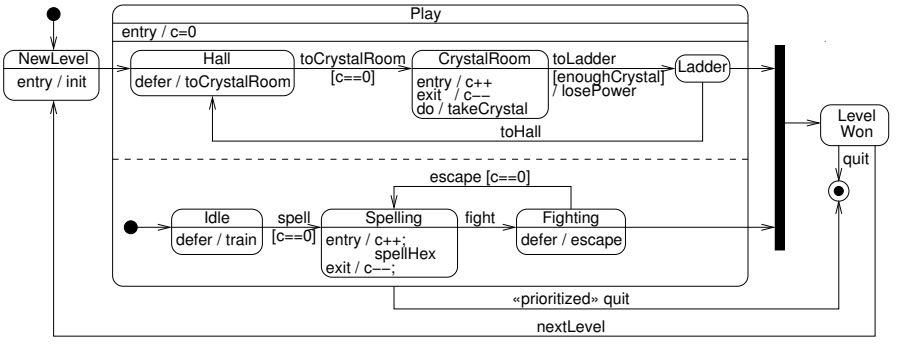


Fig. 2. Mutual exclusion in plain UML: passive waiting

Both strategies have realistic use cases, see e.g. [13,14]. Unfortunately, both are hard to model using plain UML.

As an example, assume for the above example a mutual-exclusion rule that requires the magician not to spell a hex in the crystal room. That is, in Fig. 1, the states CrystalRoom and Spelling must not be simultaneously active. A possible implementation of passive waiting to achieve the mutual exclusion is modeled in Fig. 2. A variable c is introduced and used to control the access to the two critical states: it is initialized as 0 in the entry action of Play, increased whenever CrystalRoom or Spelling is activated, and decreased whenever one of the two states is deactivated. The three transitions that activate the two states (from Hall to CrystalRoom, from Idle to Spelling, and from Fighting to Spelling), are extended by a guard, such that they are only fired when c equals 0, which means that the other critical state is currently inactive and the mutual exclusion rule is satisfied. A subtle point is that we have to declare the events toCrystalRoom, spell, and escape to be deferrable in the states Hall, Idle, and Fighting respectively. In this way the transitions are only postponed if the other critical state is active, and will be automatically resumed without requiring the events to be sent again. Otherwise the events would be lost in case exactly one of the critical states were active, since the event would then be taken from the event pool without firing a transition. Overall, the model elements we introduced make one region passively wait until some state in another region is left.

The other strategy, active commanding, is modeled in Fig. 3. In addition to variable c and the related entry and exit actions introduced above, new junctions, states and prioritized signals are also necessary. In the upper region, the (compound) transition from Hall to CrystalRoom is only enabled if c equals to 0, which means that the other critical state, Spelling, is inactive hence activating CrystalRoom would not break the mutual-exclusion rule. On the other hand, if c is not 0, i.e. Spelling is active, then Wait1 is activated. Within its entry action, a signal stopS is sent to the state machine itself and, due to the high priority (indicated by stereotype ≪prioritized≫), handled immediately. Since Spelling is active, the transition to Idle is fired, which means Spelling now becomes inactive. The transition sends then an event cntn1, which is also handled immediately in

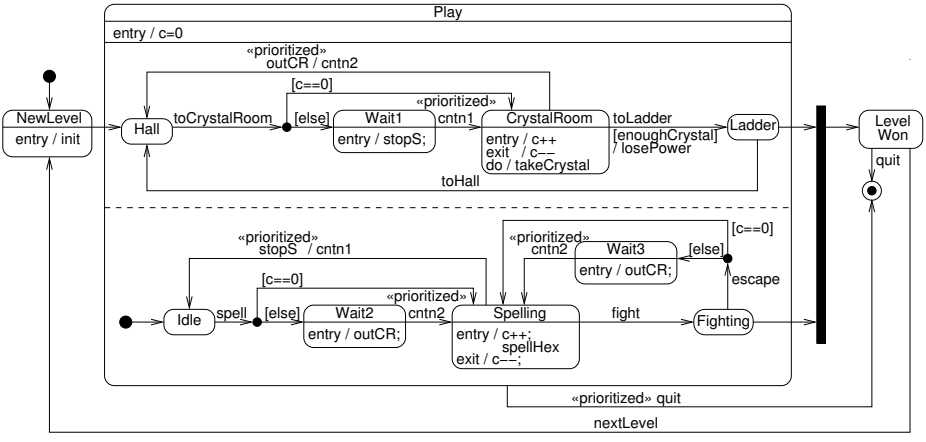


Fig. 3. Mutual exclusion in plain UML: active commanding

the upper region, where the transition from Wait1 to CrystalRoom is fired. Now CrystalRoom is active. The active state configuration consists of Play, CrystalRoom, and Idle. The function of Wait2 and Wait3 is similar. Overall, the Wait* states, the case distinctions, and the additional, prioritized events build up a mechanism for transitions in one region to send events which can actively influence the behavior of another region.

In both cases, it is obviously unsatisfactory that modeling even such a simple mutual exclusion rule requires modification of many model elements, scattered all over the state machine. The modification of the behavior is even in such a relatively simple example hard to understand. Furthermore, it is easy to introduce errors which are hard to find. Such modeling makes maintenance difficult, the models are complex and prone to errors.

3 Modeling Mutual Exclusion with HiLA

As a possible solution of UML state machines' modularity problems, the language High-Level Aspects (HiLA, [11,16]) was defined as an aspect-oriented extension for UML state machines. HiLA provides high-level constructs for declarative, as opposed to imperative, behavior modeling.

3.1 HiLA in a Nutshell

The concrete syntax of a HiLA aspect is shown in Fig. 4 and explained in the following. Syntactically, a HiLA aspect is a UML template containing at least a name, a pointcut and an advice. The template parameters allow easy customization, so that aspects for functionalities such as logging, transactions or mutual exclusions can easily be reused in many places.

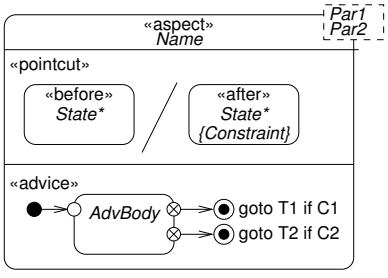


Fig. 4. HiLA: concrete syntax

An aspect is applied to a UML state machine, which is called the *base machine*. An aspect defines some additional or alternative behavior of the base machine at some points in time during the base machine's execution. The behavior is defined in the *advice* of the aspect; the points in time to execute the advice are defined in the *pointcut*. The advice (stereotype ≪advice≫ in Fig. 4) also has the form of a state machine, except that 1) transitions may carry out a special kind of action and 2) the final states may carry a label. The "body" of the advice, i.e. the part without the initial vertex, the final states, and the connecting transitions, models the behavior to carry out.

The special actions are called *commanding actions*. A commanding action has the form @X goto Y, and means that if state X (in the base machine) is active, then it should be deactivated, and state Y (in the base machine) should be activated. A label consists of a state, which should be activated when the advice is finished and the execution of the base machine should be resumed. We refer to this state as the *resumption state* of the final state. The label may optionally be guarded by a *resumption constraint*, which is indicated by the keyword if and has the form (like|nlike) StateName* where StateName is the set of the qualified names of the base machine's states. like S is true iff after the resumption all states contained in S will be active, otherwise nlike S is true.

The pointcut (≪pointcut≫) specifies the "interesting" points in time when the advice is executed. The points in time may be 1) when a certain state of the base machine is just about to become active or 2) a set of states has just been left. The semantics of a pointcut can be regarded as a selection function of the base machine's transitions: a pointcut ≪before≫ s selects all transitions in the base machine whose firing makes state s active, and a pointcut ≪after≫ S selects those transitions whose firing deactivates S.

Overall, an aspect is a graphical model element stating that at the points in time specified by the pointcut the advice should be executed, and after the execution of the advice the base machine should resume execution by activating the state given by the label of the advice's final state, when the conditions given there are satisfied. For ≪before≫ and ≪after≫ pointcuts, this "point in time" is always the firing of a transition; we say that this transition is *advised* by the advice.

3.2 Modeling Mutual Exclusion with HiLA

Since states to mutually exclude are always in different, parallel regions, mutual exclusion is actually a special case of cross-region communication. HiLA provides elegant solutions for modeling cross-region communication in general and mutual exclusion in particular. For the very common feature of mutual exclusion, reusable templates for its modeling are defined.

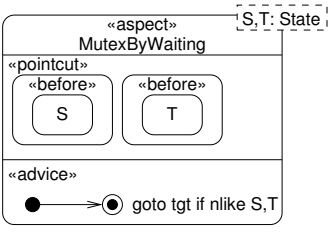


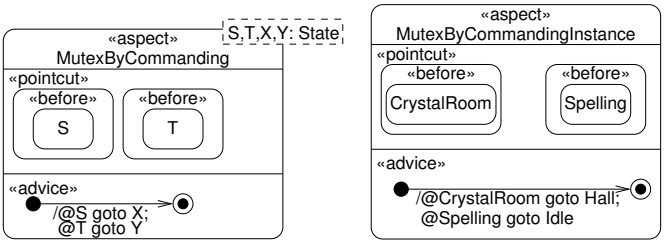
Fig. 5. HiLA template modeling mutual exclusion by passive waiting

Passive Waiting. The basic idea of aspect-oriented modeling of passive waiting is to define an aspect to interrupt the execution of the transition that would otherwise break the mutual-exclusion requirement, and to delay the resumption from the aspect until the other critical state is inactive again.

In HiLA, passive waiting is modeled by (instances of) the template shown in Fig. 5. The template takes two State parameters, S and T. The pointcut is a shortcut of "≪before≫ S or ≪before≫ T", and specifies all the points in time when either S or T is just about to get active. In such moments the advice is executed, which contains an empty body and simply conducts the base machine to resume the advised transition (by going to its target), when after the resumption the states S and T would not be both active (condition nlike S, T). Compared with the UML solution, the imperative details of mutual exclusion are now transparent for the modeler, the modeling is non-intrusive, the semantics of the aspect (template) is much easier to understand hence less error-prone. Instantiating the template by binding S to Enchanted and T to CrystalRoom elegantly prevents our magician from entering the crystal room while being enchanted and also from becoming enchanted while in the CrystalRoom.

Active Commanding. Active commanding not only modifies the behavior of the region that has to wait, but the blocking state (contained in another region) is also deactivated immediately. In HiLA, we use a commanding action (see Sect. 3.1) to achieve this behavior.

Figure 6(a) shows an aspect template for modeling mutual exclusion by active commanding. Before state S or T becomes active, the advice is executed, in which state S is told to move to X and T to move to Y. Note that this actually means "before S becomes active tell T to move to Y" (because in this moment S cannot be active, and the @S event will be simply discarded) and "before T becomes active tell S to move to X". Instantiating the template by binding S to



(a) Template (b) Instance

Fig. 6. HiLA: mutual exclusion by active commanding

CrystalRoom, T to Fighting, X to CrystalRoom, and Y to Spelling yields the concrete aspect shown in Fig. 6(b), which models the active-commanding implementation of mutual exclusion in our example. Again, compared with the UML solution, the HiLA solution is higher-level and much more easier to construct and to comprehend.

4 Weaving

Weaving is the process of transformation the base machine to incorporate the behavior modeled in aspects. In the following, we show the weaving techniques to implement the execution resumption from aspect to base machine and to implement commanding actions. These techniques are essential for the implementation of mutual-exclusion aspects. Note however, the techniques were designed for very general HiLA aspects, of which mutual-exclusion aspects are just special cases. See also [15] for other techniques used in the weaving process of HiLA, such as tracking the last active state configuration (used in implementing ≪after≫ aspects) or removing pseudo-states (used to handle with syntax variations of UML state machines).

In the following, we first present a simple transformation of the base machine to allow us to track active states during the base machine's execution. We then describe in Sect. 4.2 how an aspect is woven if it is the only aspect advising a given transition. Even on this simplistic stage, correct implementation of execution resumption and commanding events necessitates some elaborate techniques. Then, in Sect. 4.3, we expand the basic-weaving techniques to cover weaving of multiple aspects as well.

Notation. Given an aspect α, we refer to the set of the advice's top-level final states as $\mathcal{F}(\alpha)$. We assume that the final states contained in $\mathcal{F}(\alpha)$ are numbered. For each $f \in \mathcal{F}(\alpha)$, $\mathsf{num}(f)$ returns its number, $\mathsf{cond}(f)$ returns its resumption condition, and $\mathsf{label}(f)$ returns its label. Given a number $\mathsf{num}(f)$, we write $F(\mathsf{num}(f))$ to get f. Given a transition t, we refer to its source, target, guard, and action as $\mathsf{src}(t)$, $\mathsf{tgt}(t)$, $\mathsf{guard}(t)$, and $\mathsf{eft}(t)$, respectively.

4.1 Tracking Active States

In plain UML, the information of which states are currently active is not provided by any built-in language construct. On the other hand, this information is essential for the implementation of HiLA aspects modeling cross-region communication. We therefore extend the entry and exit action of each state in the base machine before actual weaving. The purpose of the extension is to store for each state in a variable if it is currently active or not. That is, for each state s in the base machine, we define a boolean variable a_s, which is initialized to be false, and set it to be true and false in the entry and exit action of s, respectively. This way, a_s is true iff s is active. This idea is shown in Fig. 7.

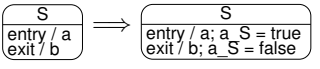


Fig. 7. Transformation for tracking active states

With this transformation, it is very easy to determine for a given set of states if all states contained in it are active. We ignore in this paper the details, and simply use impl(cond(f)) to notate the correct implementation of checking if the resumption condition of a final state is satisfied.

4.2 Weaving a Single Aspect

In HiLA, aspects are woven as a composite state containing (an implementation of) the advice. The composite state is inserted into each transition advised by the aspect.

Basic Idea. The basic idea of weaving a ≪before≫ aspect[3] is shown in Alg. 1. For each transition τ of the base machine, we assume it is advised by at most one aspect, which we call α. We first remove τ from the base machine, and replace it by the model elements we are going to insert. We insert a new state Asp into the base machine, insert an implementation of the advice into (the only region of) Asp (line 5), and set the completion event, which we notate as *, as deferrable (line 6). The "implementation" is a copy of the advice, except that commanding actions need a more involved implementation, see below. We also insert a junction j, and connect it with Asp by a transition. For each $f \in \mathcal{F}(\alpha)$, we still have to resolve its label since labels are not defined in UML. To this end, we extend the effect of the transition t leading to f by an action to store num(f) in a variable gt (line 10), insert a transition t'' from j to the label state of f (line 11), and guard (line 12) this transition so that it is only enabled when gt is equal to the number of f (which means f was the final state where the execution of the advice terminated) and the resumption condition is satisfied.

[3] In this work it suffices to consider only ≪before≫ aspects, because only these are needed to model mutual exclusion. In the case of ≪after≫ aspects, state Asp is slightly more involved, see [15].

Algorithm 1. Weaving a single aspect to transition

Require: each transition advised by at most one aspect

1: **for** each transition τ **do**
2: **if** τ advised by aspect α **then**
3: removeTransition(τ)
4: Asp $\leftarrow$ insertState
5: insertAdvice(α, Asp)
6: setDefer(Asp, *)
7: $j \leftarrow$ insertJunction; $t' \leftarrow$ insertTransition(Asp, j);
8: **for** $f \in \mathcal{F}$(Asp) **do**
9: $t \leftarrow$ transition σ, such that tgt(σ) = f
10: eft(t) $\leftarrow$ eft(t) $\cdot$ $'gt \leftarrow$ num(f)$'$ ▷ store the number of final state
11: $t'' \leftarrow$ insertTransition(j, label(f))
12: guard(t'') $\leftarrow$ $'gt =$ num(f) $\wedge$ impl(cond(f))$'$
13: **end for**
14: **end if**
15: **end for**

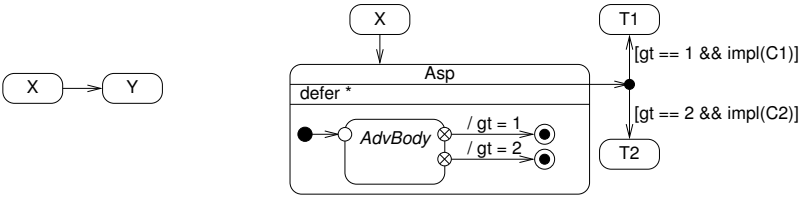
As an example, Fig. 8(b) shows the result of weaving the very general aspect (we assume the pointcut to be a ≪before≫ pointcut, though) given in Fig. 4 to a transition as given in Fig. 8(a). At run time, after the execution of *AdvBody*, the transition to one of the final states inside Asp is fired, and the final state is activated. Then, depending on the value of gt, the transition from Asp to T1, T2 or Y is activated in the right moment, i.e., when the resumption condition is also satisfied.

Commanding Actions. In HiLA, the advice of an aspect may contain commanding actions. Basically, the "command" is implemented by an event sent to the base machine itself, to be handled by the addressee. To ensure that the event is handled immediately, we need to prioritize the handling of the event.

Commanding actions may be carried out by transitions within the advice. Recall that we insert an implementation of the advice into (a region) of the state Asp (Alg. 1, line 5). Actually the implementation is simply a copy of the advice, except that transitions sending commanding events need a more involved implementation, as will be explained in the following.

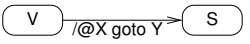
We require that the transition executing an commanding action have a state as its target. Note that we do not lose generosity due to this requirement, since all pseudo-states as transition targets can be eliminated by a semantic-preserving normalization process, see [11, Ch. 5]. Recall that a commanding action has the form @X goto Y, and that it causes state X to go to Y (both defined in the base machine) immediately. We refer to the source vertex of the transition as V, which may or may not be a state, and refer to the target state as S, see Fig. 9(a), where we assumed V is also a state.

While weaving, this transition is not simply copied into the state Asp, but transformed as follows: we insert into the region containing V and S a junction j, a state Waiting and connection transitions. Note trigger cntn of the transition



(a) Advised transition (b) Result

Fig. 8. Weaving a single ≪before≫ aspect



(a) Transition in the advice

(b) Weaving result: transition (c) Weaving result: addressee

Fig. 9. Weaving: commanding action

from Wait to S has a high priority. Additionally, in the region containing X and Y, a transition t from X to Y is also inserted. The trigger of transition t is an event toY, which has high priority and will be handled as soon as received.

At run time, when the transition leaving V is fired, one of the two following cases applies: if X is active (i.e. if a_X is true), then state Wait is activated, and, within its entry action, the signal toY is sent. Since X is active, and toY has a high priority, the transiton to Y is handled immediately, which means X becomes inactive, a signal cntn is sent, and Y gets active. Back in the upper region, Wait is active, cntn has a high priority, therefore the transition to T is fired, and the execution of the advice is finished. On the other hand, if X is not active when t_1 is fired, then Y is simply activated, just "as usual", and the mutual-exclusion requirement is not violated.

Applying the above algorithms to our example, weaving an instance of Fig. 5 with S bound to CrystalRoom and T bound to Spelling to the base machine given in Fig. 1 yields a result that is very similar to Fig. 2, and weaving Fig. 6(b) to the base machine yields a result very similar to Fig. 3. Since the weaving techniques are designed for general HiLA aspects, the weaving result of very simple advice may exhibit some overhead. We consider optimization to be an interesting piece of future work.

4.3 Multiple Aspects

Obviously, only in trivial systems it applies (as assumed above) that a transition is advised by at most one aspect. In any realistic system, multiple aspects may be interacting, i.e. advising the same transition. In such cases, it is essential for the weaving algorithm to help determine or even reconcile potential conflicts.

Algorithm 2. Weaving multiple aspects

```
 1:                                  ▷ General case: multiple aspects advising one transition
 2: for each transition τ do
 3:     A ← {α | α advising τ}
 4:     if A ≠ ∅ then
 5:         removeTransition(τ)
 6:         Asp ← insertState
 7:         setDefer(Asp, *)
 8:         j ← insertJunction; t′ ← insertTransition(Asp, j);
 9:         for each α advising τ do
10:             insertAdvice(α, Asp);
11:             for f ∈ F(Asp) do
12:                 t ← transition σ, such that tgt(σ) = f
13:                 eft(t) ← eft(t) · 'gt(α) ← num(f)' ▷ store the number of final state
14:             end for
15:         end for
16:         F ← ⋃_{α∈A} F(α)
17:         for g ∈ ⋃_{f∈F} label(f) do
18:             t″ ← insertTransition(j, g)
19:             guard(t″) ← ' ⋀_{α∈A} F(gt(α)) = g ∧ impl(cond(F(gt(α))))'
20:         end for
21:         Err ← insertState
22:         t_e ← insertTransition; guard(t_e) ← 'else'
23:     end if
24: end for
```

Resumption. When a transition is advised by multiple aspects, it is important to ensure that all aspects (i.e. their advice) are executed, and that the aspects specify the same resumption state to go to after the execution. If they specify different resumption states, it is a conflict. To this end, we extend the algorithm presented in Sect. 4.2 to Alg. 2. For each advising aspect, we insert an implementation of its advice into a region of Asp (line 10). Therefore, Asp is in general an orthogonal state, containing a multitude of regions, to be executed in parallel at run time. Instead of inserting a transition from junction j to $\mathsf{label}(f)$ for each final state f (Alg. 1, line 7), we now insert a transition to each state g such that g is the resumption state of some final state of the advice of any of the aspects (line 18). The guard of the transition ensures that the transition is enabled iff in each the region, the actual resumption state is really the target of the transition (line 19).[4] Otherwise, if different resumption states are specified by different aspects, this is a conflict situation. In this situation, state Exception is activated, inserted to the base machine by in lines 21 and 22.

Commanding Actions. The idea of weaving commanding actions shown in Sect. 4.2 also works for multiple aspects. If a state is addressee of multiple commanding actions, then multiple transitions will be introduced. It is important,

[4] Recall that $F(gt(\alpha))$ returns the $gt(\alpha)$-th final state of (the advice of) aspect α.

though, to generate different event names for the different transitions. This way, conflicting is eliminated.

5 Related Work

HiLA is a semantic approach, i.e., HiLA aspects define modifications of the semantics of the base machine. Therefore, the semantics of an aspect can be described in a purely behavioral manner. The modeler only needs to specify what to do (in this paper: what to mutually exclude), and no longer has to specify how do to it in imperative details, since the details are hidden behind the weaving algorithms and thus transparent to the modeler. In comparison, prevalent approaches of incorporating aspect-orientation into UML state machines, such as [1,2,7,8], are mainly syntactic. Aspects typically define transformations of the (abstract syntax) of the base model. It is therefore the modeler's job to define aspects (modifications of the syntax of the base machine) such that the overall behavior of the modified base machine is the desired one.

The semantics of such syntactic aspects are usually defined by by graph transformation systems, such as Attributed Graph Grammar (AGG, [10]). Consistency checks are supported by a confluence check of the underlying graph transformation, see e.g. [7]. Due to the syntactic character of the aspects, this check is also syntactic: there may be false alarms if different weaving orders lead to syntactically different but semantically equivalent results. In contrast, in our approach described above, the error state is only entered when the resumption variables are really conflicting.

The pointcut language JPDD [6] also allows the modeler to define "stateful" pointcuts. Compared with HiLA, a weaving process is not defined. State-based aspects in reactive systems are also supported by the Motorola WEAVR tool [17]. Their aspects can be applied to the modeling approach Rational TAU[5], which supports flat, "transition-centric" state machines. In comparison, HiLA is also applicable to UML state machines, that in general include concurrency. Ge et al. [5] give an overview of an aspect system for UML state machines. They do not give enough details for a thorough comparison, but it appears that the HiLA language is significantly stronger the theirs, and that the issues presented in this paper are not addressed by their solution.

To the author's knowledge, declarative modeling of mutual exclusion is not directly supported by the above approaches.

6 Conclusions and Future Work

We have presented how HiLA, an aspect-oriented extension of UML, can be applied to model mutual exclusion in UML state machines, as well as weaving techniques used to implement mutual-exclusion aspects. Using HiLA, both passive waiting and active commanding can be modeled highly modularly and

[5] http://ibm.com/software/awdtools/tau/

declaratively. That is, mutual-exclusion is modeled at a dedicated place rather than by model elements scattered all over the state machine, and the modeler only has to specify which states to mutually exclude rather than how to do it. Moreover, our implementation minimizes potential conflicts between aspects by weaving interacting aspects into parallel regions of a composite state.

Currently we are working on an extension of the tool Hugo/HiLA [11] to automate the weaving process of commanding aspects. Future work includes investigation of how HiLA can help model interactive user interfaces [12] more modularly, as well as techniques of factorizing aspects out of plain UML state machines.

References

1. 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)
2. Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design: the Theme Approach. Addison-Wesley (2005)
3. Drusinsky, D.: Modeling and Verification Using UML Statecharts. Elsevier (2006)
4. Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.): MODELS 2007. LNCS, vol. 4735. Springer, Heidelberg (2007)
5. Ge, J.-W., Xiao, J., Fang, Y.-Q., Wang, G.-D.: Incorporating Aspects into UML State Machine. In: Proc. Advanced Computer Theory and Engineering (ICACTE 2010). IEEE (2010)
6. Hanenberg, S., Stein, D., Unland, R.: From Aspect-Oriented Design to Aspect-Oriented Programs: Tool-Supported Translation of JPDDs into Code. In: Barry, B.M., de Moor, O. (eds.) Proc 6th Int. Conf. Aspect-Oriented Software Development (AOSD 2007), pp. 49–62. ACM (2007)
7. Jayaraman, P.K., Whittle, J., Elkhodary, A.M., Gomaa, H.: Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis. In: Engels et al. [4], pp. 151–165
8. Mahoney, M., Bader, A., Elrad, T., Aldawud, O.: Using Aspects to Abstract and Modularize Statecharts. In: Proc. 5th Int. Wsh. Aspect-Oriented Modeling, Lisboa (2004)
9. OMG, Unified Modeling Languague Superstructure, Version 2.4.1. Specification, Object Management Group (2011), http://www.omg.org/spec/UML/2.4.1/Superstructure/
10. Taentzer, G.: AGG: A Graph Transformation Environment for Modeling and Validation of Software. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 446–453. Springer, Heidelberg (2004)
11. Zhang, G.: Aspect-Oriented State Machines. PhD thesis, Ludwig-Maximilians-Universität München (2010)
12. Zhang, G.: Aspect-Oriented UI Modeling with State Machines. In: Van den Bergh, J., Sauer, S., Breiner, K., Hußmann, H., Meixner, G., Pleuss, A. (eds.) Proc. 5th Int. Wsh. Model-Driven Development of Advanced User Interfaces (MDDAUI 2010), pp. 45–48 (2010)

13. Zhang, G., Hölzl, M.: HiLA: High-Level Aspects for UML State Machines. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 104–118. Springer, Heidelberg (2010)
14. Zhang, G., Hölzl, M.: Improving the Modularity of Web-Application Models with Aspects (submitted, 2012)
15. Zhang, G., Hölzl, M.: Weaving Semantic Aspects in HiLA. In: Hirschfeld, R., Tanter, É., Sullivan, K.J., Gabriel, R.P. (eds.) Proc. 11th Int. Conf. Aspect-Oriented Software Development (AOSD 2012), pp. 263–274. ACM (2012)
16. Zhang, G., Hölzl, M., Knapp, A.: Enhancing UML State Machines with Aspects. In: Engels et al. [4], pp. 529–543
17. Zhang, J., Cottenier, T., van den Berg, A., Gray, J.: Aspect Composition in the Motorola Aspect-Oriented Modeling Weaver. Journal of Object Technology 6(7), 89–108 (2007)

TexMo: A Multi-language Development Environment

Rolf-Helge Pfeiffer and Andrzej Wąsowski

IT University of Copenhagen, Denmark
{ropf,wasowski}@itu.dk

Abstract. Contemporary software systems contain a large number of artifacts expressed in multiple languages, ranging from domain-specific languages to general purpose languages. These artifacts are interrelated to form software systems. Existing development environments insufficiently support handling relations between artifacts in multiple languages.

This paper presents a taxonomy for multi-language development environments, organized according to language representation, representation of relations between languages, and types of these relations. Additionally, we present TexMo, a prototype of a multi-language development environment, which uses an explicit relation model and implements visualization, static checking, navigation, and refactoring of cross-language relations. We evaluate TexMo by applying it to development of a web-application, JTrac, and provide preliminary evidence of its feasibility by running user tests and interviews.

1 Introduction

Maintenance and enhancement of software systems is expensive and time consuming. Between 85% to 90% of project budgets go to legacy system operation and maintenance [6]. Lientz et. al. [19] state that 75% to 80% of system and programming resources are used for enhancement and maintenance, where alone understanding of the system stands for 50% to 90% percent of these costs [25].

Contemporary software systems are implemented using multiple languages. For example, PHP developers regularly use 1 to 2 languages besides PHP [1]. The situation is even more complex in large enterprise systems. The code base of OFBiz, an industrial quality open-source ERP system contains more than 30 languages including General Purpose Languages (GPL), several XML-based Domain-Specific Languages (DSL), config files, property files, and build scripts. ADempiere, another industrial quality ERP system, uses 19 languages. ECommerce systems Magento and X-Cart utilize more than 10 languages each.[1] Systems utilizing the model-driven development paradigm additionally rely on multiple languages for model management, e.g., meta-modelling (UML, Ecore, etc.) model transformation (QVT ATL, etc.), code generation (Acceleo, XPand, etc.), and model validation (OCL, etc.).[2]

[1] See `ofbiz.apache.org`, `adempiere.com`, `magentocommerce.com`, `x-cart.com`

[2] See `uml.org`, `eclipse.org/modeling/emf`, `omg.org/spec/QVT`, `eclipse.org/atl`, `eclipse.org/acceleo`, `wiki.eclipse.org/Xpand`, `omg.org/spec/OCL` respectively.

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 178–193, 2012.

(a) Declaration of the *translate* command attached to a button.

(b) JavaScript code that is executed whenever the button is pressed.

Fig. 1. Declaration of a command and its use

We call software systems using multiple languages, *Multi-Language Software Systems (MLSS)*. Obviously, the majority of modern software systems are MLSSs.

Development artifacts in MLSS can be models, source code, property files, etc. To simplify presentation, we refer to all these as *mograms* [18]. Mograms in MLSS are often heavily interrelated. For example, OFBiz contains many hundreds of relations across its languages [23,13]. Unfortunately, relations across language boundaries are fragile. They are broken easily, as development environments neither visualize nor statically check them.

Consider the following scenario. For simplicity of presentation we use a small example. Our work, though, is not tight to a particular selection of languages, or the particular example system.

Example Scenario. Bob develops a *Safari* web browser *extension*. The extension contributes a button to Safari's menu bar. Pressing the button translates the current web-page to English using *Google translate* and presents it in a new tab. Browser extensions are usually built using HTML, CSS and JavaScript. Bob's extension consists of three source code files: *Info.plist*, *button.js*, and *global.html*.

Plist files serve as an interface for the extension. They tell Safari what the extension contributes to the UI. In Bob's extension, the *Plist* file contains the declaration of a `translate` command attached to a toolbar button (Fig.1a). *JavaScript* code contains logic attached to buttons, menus, etc. Bob's *button.js* forwards the current URL to Google's translation service whenever the corresponding button is pressed (Fig.1b). Every extension contains a *global.html* file, which is never displayed. It contains code which is loaded at browser start-up or when the extension is enabled. It is used to provide code for extension buttons, menus, etc. Bob's *global.html* file (not shown here) contains only a single `script` tag pointing to *button.js*.

In Fig.1a the `translate` command for the button is defined. Fig.1b shows how the translate command is used in *button.js* in a string literal. This is an example of a *string-based reference* to *Info.plist*. Such string-based references are common in development of MLSSs.

Now, imagine Bob renaming the command in *Info.plist* from `translate` to its Danish equivalent `oversæt`. Obviously, the browser plugin will not work anymore since the JavaScript code in *button.js* is referring to a non-existing

command. Symmetrically, the reference is broken whenever the "`translate`" string literal is modified in the `button.js` file, without the corresponding update to *Info.plist*. □

Existing Integrated Development Environments (IDE) do not directly support development of MLSSs. IDEs do not visualize cross-language relations (markers left to line numbers and gray highlighting in Fig. 1). Neither do they check statically for consistency of cross-language relations, or provide refactorings across mograms in multiple languages. We are out to change this and enhance IDEs into *Multi-Language Development Environments (MLDE)*.

This paper introduces a taxonomy of design choices for MLDEs (Sec. 2). The purpose of this taxonomy is twofold. First, it serves as requirements list for implementing MLDEs, and second it allows for classification of such. We argue for the validity of our taxonomy by a survey of related literature and tools.

As the second main contribution, the paper presents TexMo (Sec. 3), an MLDE prototype supporting textual GPLs and DSLs. It implements actions for *visualization* of, *static checking* of, *navigation* along, and *refactoring* of inter-language relations, and facilities to declare inter-language relations. Additionally, TexMo provides standard editor mechanisms such as syntax highlighting. We position TexMo in our taxonomy and evaluate it by applying it to development of an MLSS and user tests followed by interviews.

2 Taxonomy of Multi-language Development Environments

Popular IDEs like Eclipse or NetBeans implement separate editors for every language they support. A typical IDE provides separate Java, HTML, and XML editors, even though these editors are used to build systems mixing all these languages. Representing languages separately allows for an easy and modular extension of IDEs to support new programming languages. Usually, IDEs keep an *Abstract Syntax Tree (AST)* in memory and automatically synchronize it with modifications applied to concrete syntax. IDE editors exploit the AST to facilitate source code navigation and refactorings, ranging from basic renamings to elaborate code transformations such as *method pull ups*.

Inter-language relations are a major problem in development of MLSS [23,13,12]. Since they are mostly implicit, they hinder modification and evolution of MLSS. An MLDE is an IDE that addresses this challenge by not only integrating tools into a uniform working experience, as IDEs do, but also by integrating languages with each other. MLDEs support across language boundaries the mechanisms implemented by IDEs for every language separately.

We surveyed IDEs, programming editors[3], and literature to understand the kind of development support they provide. We realized that 4 features, that

[3] IDEs: Eclipse, NetBeans, IntelliJ Idea, MonoDevelop, XCode. Editors: MacVim, Emacs, jEdit, TextWrangler, TextMate, Sublime Text 2, Fraise, Smultron, Tincta, Kod, gedit, Ninja IDE. (See project websites at: `www.itu.dk/~ropf/download/list.txt`)

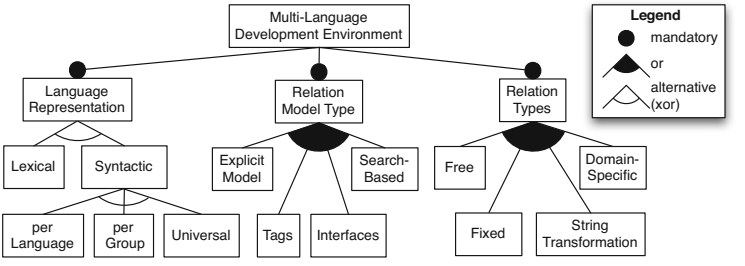


Fig. 2. Taxonomy for multi-language development environments

visualization, *navigation*, *static checking*, and *refactoring* are implemented by all IDEs and by some programming editors. Thus, in order to support developers best, MLSS need to consider delivering these features across language boundaries as their essential requirements:

1. *Visualization.* An MLDE has to highlight and/or visualize inter-language references. Visualizations can range from basic markers, as for instance in the style of Fig. 1 to elaborate visualization mechanisms such as treemaps [7].
2. *Navigation.* An MLDE has to allow navigating along inter-language relations. In Fig. 1, the developer can request to automatically open *button.js* and jump to line 8, when editing *Info.plist*. All surveyed IDEs allow to navigate source code. Further, IDEs allow for source code to documentation navigation, a basic multi-language navigation.
3. *Static Checking.* An MLDE has to statically check the integrity of inter-language relations. As soon as a developer breaks a relation, the error is indicated to show that the system will not run error free. All surveyed IDEs provide static checking by visualizing errors and warnings.
4. *Refactoring.* An MLDE has to implement refactorings, which allow easy fixing of broken inter-language relations. Different IDEs implement a different amount of refactorings per language. Particularly, rename refactorings seem to be widely used in current IDEs [21,31].

To address these requirements one needs to make three main design decisions: **a)** *How to represent different programming languages?* **b)** *How to inter-relate them with each other?* **c)** *Using which kind of relations?*

Systematizing the answers to these questions led us to a domain model characterizing MLDEs. We present this model in Fig. 2 using the feature modeling notation [5,16]. An MLDE always represents mograms based on the their language (*Language Representation*). Furthermore, an MLDE has to represent inter-language relations (*Relation Model Type*). This feature is essential for augmenting an IDE to an MLDE. Finally, an MLDE associates types to inter-language relations (*Relation Types*). An IDE first becomes an MLDE if it supports inter-language relations, i.e., as it implements an instance of this model.

The following subsections detail and exemplify the fundamental MLDE characteristics of our taxonomy. References to the surveyed literature are inlined.

2.1 Language Representation Types

We consider two main types of language representation, lexical and syntactic language representation. The former always works on an artifact directly without constructing a more elaborate representation, whereas the latter is always based on a richer data-structure representing mograms in a certain language. Syntactic language representation can represent mograms per language, per language group, or universally.

Lexical Representation. Most text editors, such as EMacs, Vim, and jEdit, implement lexical representation. Mograms are loaded into a buffer in a language agnostic manner. Syntax highlighting is implemented solely based on matching tokens. Due to lack of sufficient information about the edited mogram such editors provide limited support for static checking, code navigation, and refactoring.

Syntactic Representation. Per Language. Typical IDEs represent mograms in any given language using a separate AST, or a similar richer data structure capturing a mogram's structure; for instance Eclipse, NetBeans, etc. Unlike lexical representation, a structured, typed representation allows for implementation of static checking and navigation within and between mograms of a single language but not across languages. The advantage using per language representation, compared to per language group and universal representation, is that IDEs are easily extensible to support new languages.

Using models to represent source code is getting more and more popular[4]. This is facilitated by emergence of language workbenches such as EMFText, XText, Spoofax, etc.[5] The MoDisco [4] project, a model-driven framework for software modernization and evolution, represents Java, JSP, and XML source code as EMF models, where each language is represented by its own distinct model. These models are a high-level description of an analyzed system and are used for transformation into a new representation. The same principle of abstracting a programming language into an EMF model representation is implemented in JaMoPP [11]. Similarly, JavaML [3] uses XML for a structural representation of Java source code. On the other hand, SmartEMF [12] translates XML-based DSLs to EMF models and maps them to a Prolog knowledge base. The EMF models realize a per language representation. Similarly, we represent OFBiz' DSLs and Java using EMF models to handle inter-component and inter-language relations [23].

Syntactic Representation. Per Language Group. A single model can represent multiple languages sharing commonalities. Some languages are mixed or embedded into each other, e.g., SQL embedded in C++. Some languages extend others, e.g., AspectJ extends Java. Furthermore, languages are often used together, such as JavaScript, HTML, XML, and CSS in web development. Using

[4] Language workbenches mostly use modeling technology to represent ASTs. Therefore, we use the terms AST and model synonymously in this paper.

[5] See `www.languageworkbenches.net` for the annual language workbench competition.

a per language group representation allows increased reuse in implementation of navigation, static checks and refactoring in MLDEs, because support for each language does not need to be implemented separately.

For example, the IntelliJ IDEA IDE (`jetbrains.com/idea`), supports code completion for SQL statements embedded as strings in Java code. X-Develop [28,27] implements an extensible model for language group representation to provide refactoring across languages. AspectJ's compiler generates an AST for Java as well as for AspectJ aspects simultaneously. Similarly, the WebDSL famework represents mograms in its collection of DSLs for web development in a single AST [8]. *Meta*, a language family definition language, allows the grouping of languages by characteristics, e.g., object-oriented languages in *Meta(oopl)* [14]. The Prolog knowledge base in [12] can be considered as a language group representation for OFBiz' DSLs, used to check for inter-language constraints.

Syntactic Representation. Universal. Universal representations use a single model to capture the structure of mograms in any language. They can represent any version of any language, even of languages not invented yet. Universal representations use simple but generic concepts to represent key language concepts, such as blocks and identifiers or objects and associations. A universal representation allows the implementation of navigation, static checking, and refactoring only once for all languages. Except for TexMo, presented in Sec.3, we are not aware of any IDE implementing a universal language representation.

The per group and the universal representations are generalizations of the per language representation. Both represent multiple languages in one model. Generally, there are two opposing abstraction mechanisms: *type abstraction* and *word abstraction* [29]. Type abstraction is a unifying abstraction, whereas word abstraction is a simplifying abstraction.

For example, both Java and C# method declarations can include modifiers, but the set of the actual modifiers is language specific. The synchronized modifier in Java has no equivalent in C#. Under the type abstraction, Java and C# method declarations can be described by a *Method Declaration* type and an enumeration containing the modifiers. In contrast, under the word abstraction, Java and C# method declarations would be described by a common simple *Method Declaration* type that neglects the modifiers. Obviously, in the type abstraction Java and C# method modifiers are distinguishable, whereas in the more generic word abstraction this information is lost.

Type abstraction is preferable for per group representations. Word abstraction is preferred for universal representations. The choice of abstraction influences the specificity of the representation, affecting the tools. Word abstractions are more generic than type abstractions. For instance, more cross-language refactorings are possible with the per group representation, while the refactorings in the systems relying on the universal representation automatically apply to a wider class of languages.

2.2 Relation Model Types

Software systems are implemented using multiple mograms. At the compilation stage, and often only at runtime, a complete system is composed by relating all the mograms together. Each mogram can refer to, or is referenced by, other mograms. An MLDE should maintain information about these relations. We observe four different techniques to express cross-language relations:

Explicit model. For example, *mega-models* [15], *trace models* [22,9], *relation models* [23], or *macromodels* [24]. All these are models linking distributed mograms together.

Tags. Hypertext systems, particularly HTML code links substructures or other artifacts with each other by tags. Tags define anchors and links within an artifact [10]. Hypertext systems interpret artifacts, anchors, and links. first after interpretation a link is established.

Interfaces. Interfaces are anchors decoupled from artifacts. An interface contains information about a development artifact's contents and corresponding locations. For example, OSGi manifest files or model and meta-model interfaces describe component and artifact relations [13].

Search-based. There is no persistent representation of relations at all. Possible relation targets are established after evaluating a search query. Search-based relations are usually used to navigate in unknown data. For example, in [30] relations across documents in different applications are visualized on user request by searching the contents of all displayed documents.

2.3 Relation Types

Here we elaborate on relations between mograms in different languages. Since we consider only textual languages all the following relation types relate strings.

Free relations are relations between arbitrary strings. They rely solely on human interpretation. For example, natural language text in documentation can be linked to source code blocks highlighting that certain requirements are implemented or that a programmer should read some documentation. Steinberger et. al. describe a visualization tool allowing to interrelate information across domains, even across concrete syntaxes [26]. Their tool visualizes relations between diagrams and data.

Fixed relations: Relations between equal strings are fixed relations. Fixed relations occur frequently in practice. For example, the relation between an HTML anchor declaration and its link is established by equality of a tag's argument names. Figure 1 shows an example of a fixed relation across language boundaries.

Waldner et. al. discuss visualization across applications and documents [30]. Their tool visualizes relations between occurrences of a search term matched in different documents.

String-transformation relations are relations between similar strings, or functionally related strings. For example, a Hibernate configuration file (XML) describes how Java classes are persisted into a relational database. The Hibernate framework requires that a field specified in the XML file has a corresponding get and set method in the Java class. A string fieldName in a Hibernate configuration file requires a getter with name getFieldName in the corresponding Java class. Depending on the direction, a string-transformation relation either attaches or removes get and capitalizes or decapitalizes fieldName.

Domain-Specific Relations (DSRs) are relations with semantics specific to a given domain or project. DSRs are always typed. Additionally, DSRs can be free, fixed or string-transformation relations. For example, a requirements document can require a certain implementation artifact, expressing that a certain requirement is implemented. At the same time, some Java code can require a properties file, meaning that the code will only produce expected results as soon as certain properties are in place. We consider any relation type hierarchy domain-specific, e.g., trace link classification [22].

The first three relation types, free, fixed, and string-transformation relations are untyped. They are more generic than DSRs, since they only rely on physical properties of relation ends. Fixed, string-transformation, and domain-specific relations can be checked automatically, which allows to implement tools supporting MLSS development, such as error visualization and error resolution.

3 TexMo as an MLDE Prototype

TexMo[6] addresses the requirements listed in Sec. 2 and it implements an instance of our MLDE taxonomy. TexMo uses a *key-reference* metaphor to express relations. In the example of Fig. 1, the command *declaration* takes the role of a *key* (Fig. 1a) and its uses are *reference* (Fig. 1b). TexMo relations are always many-to-one relations between *references* and *keys*. We summarize how TexMo meets the requirements presented in Sec. 2:

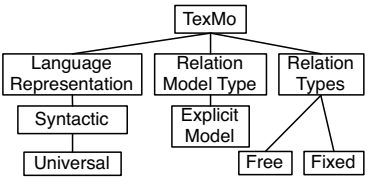
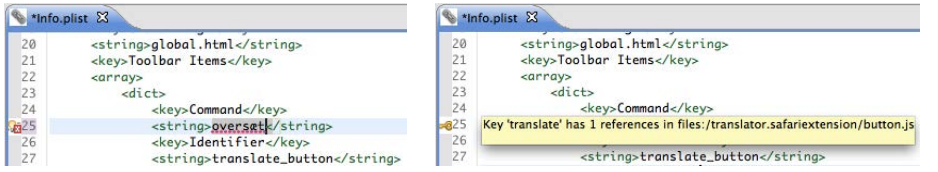


Fig. 3. The feature model instance describing TexMo in our taxonomy of MLDEs

1. *Visualization.* TexMo highlights keys and references using gray boxes, see line 25 in Fig. 1a and line 8 Fig. 1b. Keys are labeled with a key icon and references are labeled by a book icon; see Fig. 1 left to line numbers. Inspecting markers reveals detailed information, e.g., how many references in which files refer to a key, see Fig. 4b.

[6] TexMo's source code including the text model and the relation model is available online at: `www.itu.dk/~ropf/download/texmo.zip`



(a) A broken relation between command declaration and its use, see Fig. 1b.

(b) Detailed relation information attached to a key marker.

Fig. 4. Visualization and information for inter-language relations

2. *Navigation.* Users can navigate from any reference to the referred key and from a key to any of its references. Navigation actions are called via the context menu.
3. *Static checking.* Fixed relations in TexMo's relation model (RM) are statically checked. Broken relations, i.e., fixed relations with different string literals as key and reference, are underlined red and labeled by a standard error indicator in the active editor, see Fig. 4a.
4. *Refactoring.* Broken relations can be fixed automatically using quick fixes. TexMo's quick fixes are key centric rename refactorings. Applying a quick fix to a key renames all references to the content of the key. Contrary, applying a quick fix to a reference renames this single reference to the content of the corresponding key.

On top of these multi-language development support mechanisms, TexMo provides syntax highlighting for 75 languages. GPLs like Java, C#, and Ruby, as well as DSLs like HTML, Postscript, etc. are supported. Standard editor mechanisms like undo/redo are implemented, too.

Universal Language Representation. The Text Model. TexMo implements a universal language representation since such an MLDE is easily applicable for development of any MLSS.

All textual languages share a common coarse-grained structure. The text model (Fig. 5), an AST of any textual language, describes blocks containing paragraphs, which are separated by new lines and which contain blocks of words. Words consist of characters and are separated by white-spaces. The only model elements containing characters are word-parts, separators, white-spaces, and line-breaks. Blocks, paragraphs, and word blocks describe the structure of a mogram. Separators are non-letters within a word, e.g., '/','.', etc., allowing represent of typical programming language tokens as single words.

TexMo treats any mogram as an instance of a textual DSL conforming to Fig. 5. For example, a snippet of JavaScript code `if(event.command ==` , line 8 in Fig. 1b, looks like: Block(Paragraph[WordBlock(Word[WordPart("if"), SeperatorPart(content:"("), WordPart("event"), SeperatorPart("."), WordPart("command"), WhiteSpace(" ")]), ...]) (using Spoofax [17] AST notation).

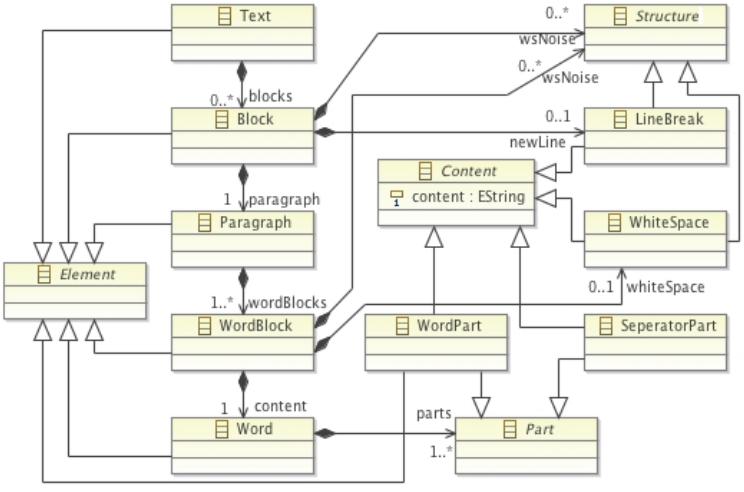


Fig. 5. The open universal model for language representation

An Explicit Relation Model. TexMo uses an instance of the Relation Model (RM) presented in Fig. 6 to keep track of relations between multi-language mogram code. Our RM allows for relations between mogram contents (ElementKey and ElementReference), between mogram contents and files (Artifacts) or components (Components), and between files and components. This allows for example to express relations in case mogram code requires another file, which occurs frequently, e.g., in HTML code.

The RM instance is kept as a textual artifact. The textual concrete syntax is not shown here, since the RM is not intended for human inspection. TexMo automatically updates the RM instance whenever developers modify interrelated mograms. That is, TexMo supports evolution of MLSS. Currently, the RM is created manually. TexMo provides context menu actions to establish relations between keys and references. Future versions of TexMo will integrate pattern based mining mechanism [23,9] to supersede manual RM creation.

Relation Types. TexMo's RM currently implements fixed and explanatory relations. Explanatory relations are free relations in our taxonomy. Keys and references of fixed relations contain the same string literal. Figure 1 shows a fixed relation and Fig. 4 shows a broken fixed relation. Explanatory relations allow to connect arbitrary text blocks with each other, for example documentation information to implementation code.

4 Evaluation

In this section we discuss TexMo's applicability. First, we evaluate TexMo's language representation mechanism, i.e., its representation of mograms as text

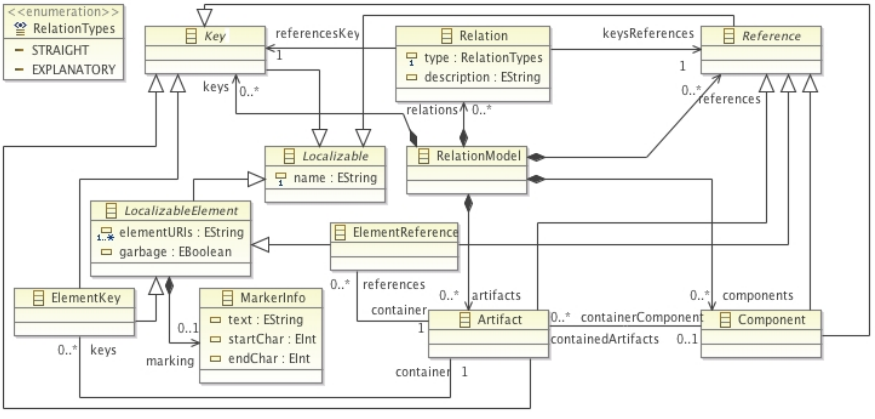


Fig. 6. TexMo's explicit relation model

models. Second, we provide preliminary evidence on the feasibility of TexMo by testing user acceptance. Furthermore, we discuss applicability of TexMo's relation model with respect to keeping inter-language relations while testers are using TexMo.

The subject used for this evaluation is the open-source web-based bug-tracking system, JTrac. JTrac's code base consists of 374 files. The majority of files, 291, contain source code in Java (141), HTML (65), property files (32), XML (16), JavaScript (8), and 29 other source code files such as Shell scripts, etc. Similar to many web-applications, JTrac implements the model-view-controller (MVC) pattern. This is achieved using popular frameworks: Hibernate (`hibernate.org`) for OR-Mapping and Wicket (`cwiki.apache.org/WICKET/`) to couple views and controller code. The remaining 83 files are images and a single jar file. We did not consider these files in our evaluation since they do not contain information in a human processable, textual syntax. Clearly, JTrac is an MLSS.

4.1 Universal Language Representation

To evaluate TexMo's universal language representation, we manually opened all 291 mograms with the TexMo editor to check if a correct text model can be established. By correct we mean that any character and string in a source code artifact has a corresponding model element in the text model, which in turn allows the RM to interrelate mograms in different languages. The files used are available at: `www.itu.dk/~ropf/download/jtrac_experiment.zip`.

We concluded that all 291 source code files can be opened with the TexMo editor. For all files a correct text model has been established.

4.2 User Test

To test user acceptance, we let 11 testers perform three typical development tasks. The testers included 4 professional developers, 3 PhD students, and 4

undergraduate students, with median 3 years of working experience as software developers.

Using only a short tutorial, which explains TexMo's features the testers had to work on the JTrac system. First, they had to find and remove a previously injected error, a broken fixed relation. Second, they had to rename a reference and fix the now broken relation. Third, they had to replace a code block, which removes two keys. We captured the screen contents and observed each tester. After task completion, each tester filled out a questionnaire. Questions asked for work experience, proficiency in development of MLSS using Java, HTML, and XML. Additionally, two open questions on the purpose of the test and on the usefulness of TexMo where asked. After the completion of questionnaires we had a short, open discussion about TexMo where we took notes on tester's opinions.

We conclude that the testers understand and use MLDE concepts. Seven testers applied inter-language navigation to better understand the source code, i.e., to inspect keys and references whenever an error was reported. Furthermore, another seven used rename refactorings to securely evolve cross-language relations in JTrac. All testers were able to find all errors and to fix them. In the following we quote a selection of the testers arguing about usefulness of TexMo (we avoid quoting complete statements for the sake of brevity). Their statements indicate that visualization, static checking, navigation and refactoring across language boundaries are useful and that such features are missing in existing IDEs.

> **Q**: *"Do you think TexMo could be beneficial in software development? Why?"*
> $\mathbf{A}_1$: *"TexMo's concepts are really convincing. I would like to have a tool like this at work."*
> $\mathbf{A}_2$: *"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."*
> $\mathbf{A}_3$: "[TexMo] *improves debugging time by keeping track of changes on source code written in different programming languages that are strongly related. I do not know any tool like this."*
> $\mathbf{A}_4$: *"I see* [TexMo] *useful, especially when many people work on the same project, and, of course, in case the projects gets big."*
> $\mathbf{A}_5$: *"I did development with Spring and a tool like TexMo would solve a lot of problems while coding."*
> $\mathbf{A}_6$: *"In large applications it is difficult to perform renaming or refactoring tasks without automated tracking of references. . . . If there would be such a reference mechanism between JavaScript and C#, it would save us a lot of work.*
> $\mathbf{A}_7$:"[TexMo] *solves* [a] *common problem experienced when software project involves multiple languages."*

Robustness of the Relation Model. To run the user test and to demonstrate that the RM can express inter-language relations in an MLSS, we established a RM relating 9 artifacts containing 51 keys, 87 references, via 87 fixed relations with each other. The RM relates code in Java, HTML, and properties files with each other. We did not aim for a *complete* RM, since we focus on demonstrating TexMo's general applicability. After the testers had finished their development

tasks, we inspected the RMs manually to verify that they still correctly interrelate keys and references.

We conclude that TexMo's RM is robust to modifications of the MLSS. After modification operations, all relations in the RM correctly relate keys and references across language boundaries.

A common concern of the testers related to replacing a code block containing multiple keys with a new code block, where TexMo complains about a number of created dangling references in corresponding files. We did not implement a feature to automatically infer possible keys out of the newly inserted code, since we consider this process impossible to automate completely.

4.3 Threats to Validity

The code base of JTrac might be to small to allow to generalize that any textual mogram in any language can be represented using TexMo's text model. However, we think that nearly 300 source code files in 15 languages gives a rather strong indication. The RM used for the user tests might be to small and incomplete. We were not interested in creating a complete RM, but only concerned about its general applicability.

To avoid direct influence on the testers in an oral interview, we used a written questionnaire. All quotes in the paper are taken from this written data.

5 Related Work

Strein et. al. argue that contemporary IDEs do not allow for analysis and refactoring of MLSS and thus are not suitable for development of such. They present X-Develop an MLDE implementing an extensible meta-model [28] used for a syntactic per language group representation. The key difference between X-Develop and TexMo is the language representation. TexMo's universal language representation allows for its application in development of any MLSS regardless of the used languages. Similarly, the IntelliJ IDEA IDE implements some multi-language development support mechanisms. It provides multi-language refactorings across some exclusive languages, e.g., HTML and CSS. Unlike in TexMo, these inter-language mechanisms are specific to particular languages since IntelliJ IDEA relies on a per language representation.

Some development frameworks provide tools to enhance IDEs. Our evaluation case, JTrac relies on the web framework Wicket. *QWickie* (`code.google.com/p/qwickie`), an Eclipse plugin, implements navigation and renaming support between inter-related HTML and Java files containing Wicket code. The drawback of framework-specific tools is their limited applicability. QWickie cannot be used for development with other frameworks mixing HTML and Java files.

Chimera [2] provides hypertext functionality for heterogeneous *Software Development Environments (SDE)*. Different programs like text editors, PDF viewers and browsers form an SDE. These programs are viewers through which developers work on different artifacts. Chimera allows for the definition of anchors on views. Anchors can be interrelated via links into a hyperweb. TexMo is

similar in that models of mograms can be regarded as views where each model element can serve as an anchor for a relation. Chimera is not dynamic. It does not automatically evolve anchors while mograms are modified. Subsequent to modifications, Chimera users need to manually reestablish anchors and adapt the links to it. Contrary, TexMo automatically evolves the RM synchronously to modifications applied to mograms. Only after deleting code blocks containing keys, users need to manually update the dangling references.

Meyers [20] discusses integrating tools in multi-view development systems. One can consider language integration as a particular flavor of tool integration. Meyers describes basic tool integration on file system level, where each tool keeps a separate internal data representation. This corresponds to the per language representation in our taxonomy. Meyers' *canonical* representation for tool integration corresponds to our universal language representation. Our work extends Meyers work by identifying a per language group representation.

6 Conclusion and Future Work

We have presented a taxonomy of multi-language development environments, and TexMo, an MLDE prototype implementing a universal language representation, an explicit relation model supporting free and fixed relations. The taxonomy is established by surveying related literature and tools. We have also argued that implementation of TexMo meets is design objectives and evaluated adequacy of its design. By itself TexMo demonstrates that design of useful MLDEs is feasible and welcomed. We reported very positive early user experiences.

To gather further experience, we plan to extend TexMo with string-transformation and domain-specific relations and compare it to an MLDE using a per language representation. We realized that it is costly to keep an explicit RM updated while developers work on a system, especially the larger a RM grows. Therefore, we will experiment with a search-based relation model. This will also overcome the vulnerability of an explicit RM to changes applied to mograms outside the control of the MLDE.

Note, TexMo's RM does not only allow the interrelation of mograms of different languages but also of mograms in a single language. We do not focus on this fact in this paper. However, this ability can be used to enhance and customize static checks and visualizations beyond those provided by current IDEs without extending compilers and other tools.

While working with TexMo we realized that a universal language representation is favorable if an MLDE has to be quickly applied to a wide variety of systems with respect to the variety of used languages. Furthermore, there is a trade-off between the language representation mechanism and the richness of the tools an MLDE can provide. Basic support, like visualization, highlighting, navigation and rename refactorings, can be easily developed on any language representation, with very wild applicability if the universal representation is used. More complex refactorings require a per group or a per language representation.

In future we plan to build support to automatically infer inter-language relations. Fixed and string-transformation relations can be automatically established

by searching for equal or similar strings. This process is not trivial as soon as a language provides for example scoping. Then inferring inter-language relations has to additionally consider language specific scoping rules. Inferring domain-specific relations has to rely on additional knowledge provided by developers, for example as patterns [23], which explicitly encode domain knowledge. Inferring free relations is probably not completely automatable but relying on heuristics and search engines could result in appropriate inter-language relation candidates.

Acknowledgements. We thank Kasper Østerbye, Peter Sestoft and David Christiansen for discussion and feedback on models for representation of language groups and for feedback on the TexMo prototype. EMFText developers have provided technical support during TexMo's development. Chris Grindstaff has developed the Color Editor (`gstaff.org/colorEditor`), parts of which were reused for TexMo's syntax highlighting. Last but not least, we also thank all the testers participating in the experiment.

References

1. Zend Technologies Ltd.: Taking the Pulse of the Developer Community, `http://static.zend.com/topics/zend-developer-pulse-survey-report-0112-EN.pdf` (February 2012)
2. Anderson, K.M., Taylor, R.N., Whitehead Jr., E.J.: Chimera: Hypermedia for Heterogeneous Software Development Enviroments. ACM Trans. Inf. Syst. 18 (July 2000)
3. Badros, G.J.: JavaML: A Markup Language for Java Source Code. Comput. Netw. 33 (June 2000)
4. Bruneliere, H., Cabot, J., Jouault, F., Madiot, F.: MoDisco: A Generic and Extensible Framework for Model Driven Reverse Engineering. In: Proc. of the IEEE/ACM International Conference on Automated Software Engineering (2010)
5. Czarnecki, K., Eisenecker, U.W.: Generative Programming: Methods, Tools, and Applications (2000)
6. Erlikh, L.: Leveraging Legacy System Dollars for E-Business. IT Professional 2 (May 2000)
7. 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)
8. Groenewegen, D.M., Hemel, Z., Visser, E.: Separation of Concerns and Linguistic Integration in WebDSL. IEEE Software 27(5) (2010)
9. 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)
10. Halasz, F.G., Schwartz, M.D.: The Dexter Hypertext Reference Model. Commun. ACM 37(2) (1994)
11. Heidenreich, F., Johannes, J., Seifert, M., Wende, C.: Closing the Gap between Modelling and Java. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 374–383. Springer, Heidelberg (2010)
12. Hessellund, A.: SmartEMF: Guidance in Modeling Tools. In: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (2007)

13. Hessellund, A., Wąsowski, A.: Interfaces and Metainterfaces for Models and Metamodels. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 401–415. Springer, Heidelberg (2008)
14. Holst, W.: Meta: A Universal Meta-Language for Augmenting and Unifying Language Families, Featuring Meta(oopl) for Object-Oriented Programming Languages. In: Companion to the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (2005)
15. 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)
16. 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 (1990)
17. Kats, L.C.L., Visser, E.: The Spoofax Language Workbench: Rules for Declarative Specification of Languages and IDEs. In: OOPSLA (2010)
18. Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels (2008)
19. Lientz, B.P., Swanson, E.B., Tompkins, G.E.: Characteristics of Application Software Maintenance. Commun. ACM 21 (June 1978)
20. Meyers, S.: Difficulties in Integrating Multiview Development Systems. IEEE Softw. 8 (1991)
21. Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. In: Proc. of the 31st International Conference on Software Engineering (2009)
22. 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)
23. Pfeiffer, R.-H., Wąsowski, 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)
24. 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)
25. Standish, T.A.: An Essay on Software Reuse. IEEE Trans. Software Eng. (1984)
26. 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)
27. Strein, D., Kratz, H., Lowe, W.: Cross-Language Program Analysis and Refactoring. In: Proc. of the 6th IEEE International Workshop on Source Code Analysis and Manipulation (2006)
28. Strein, D., Lincke, R., Lundberg, J., Löwe, W.: An Extensible Meta-Model for Program Analysis. IEEE Trans. Softw. Eng. 33 (September 2007)
29. Wagner, S., Deissenboeck, F.: Abstractness, Specificity, and Complexity in Software Design. In: Proc. of the 2nd International Workshop on the Role of Abstraction in Software Engineering (2008)
30. Waldner, M., Puff, W., Lex, A., Streit, M., Schmalstieg, D.: Visual Links Across Applications. In: Proc. of Graphics Interface (2010)
31. Xing, Z., Stroulia, E.: Refactoring practice: How it is and how it should be supported — an Eclipse case study. In: Proc. of the 22nd IEEE International Conference on Software Maintenance (2006)

On-the-Fly Emendation of Multi-level Models

Colin Atkinson, Ralph Gerbig*, and Bastian Kennel

University of Mannheim, Mannheim, Germany
{atkinson,gerbig,kennel}@informatik.uni-mannheim.de

Abstract. One of the main advantages of multi-level modeling environments over traditional modeling environments is that all ontological classification levels are treated in a uniform way and are all equally available for immediate, on-the-fly modification. However, such flexibility is a two-edged sword, since a minor change in a (meta-) ontological level can have a dramatic impact on other parts of the ontology (i.e. collection of ontological levels) - requiring a large number of "knock-on" changes to keep the overall ontology correct. To effectively exploit the modeling flexibility offered by multi-level modeling environments therefore, modelers need semi-automated support for emending ontologies to keep them consistent in the face of changes. In this paper we describe a model emendation architecture and illustrate how it can help modelers maintain the correctness of an ontology.

Keywords: multi-level emendation, orthogonal classification architecture, ontological classification, linguistic classification.

1 Introduction

Although meta-modeling is now a widely practiced activity in software engineering, and meta-models play a pivotal role in advanced model-driven development projects, most contemporary modeling environments still place unnecessary limitations on the way meta-models can be defined, evolved and applied. This is because most environments are still based on the idea of applying a two-level physical platform to a logical multi-level modeling hierarchy in a fixed and unchangeable way. Traditional modeling environments apply the physical platform to the M_2 and M_1 levels of the OMG model hierarchy so that the M_2 level is the frozen type level (i.e. hardwired into the platform) and the M_1 level is the editable instance data. Meta-modeling environments (such as language engineering environments) apply the platform to the M_3 and M_2 levels so that M_3 is the frozen type level and the M_2 level is the evolvable instance data. This approach is fine as long as users are content to work within the single evolvable level supported by a tool, but it requires complex transformations and recompilations to be performed as soon as they wish to make the results of their work applicable

* Ralph Gerbig was supported by Deutsche Forschungsgemeinschaft (DFG) as part of SPP 1496 "Reliably Secure Software Systems".

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 194–209, 2012.

to lower logical levels not supported by the tool. These transformations and recompilations required to "deploy" a model are not only cumbersome and time consuming, they are also error prone. To support such scenarios tools like Edapt [4] in the EMF universe exist.

Multi-level modeling aims to overcome this problem by making all logical classification levels editable and evolvable as linguistic instance data in a uniform and level-agnostic way. It is still based on a two-level physical platform, but the frozen type model (the linguistic model) is specially designed to support multiple logical (a.k.a ontological) levels at the linguistic instance level below. This so called Orthogonal Classification Architecture (OCA) therefore supports two distinct forms of classification, organized in two orthogonal dimensions - linguistic classification which is supported directly by the underlying physical platform and ontological classification which is supported within the evolvable linguistic level. By arranging for all end user modeling and meta-modeling services (e.g. DSL definition and application capabilities) to be supported within the ontological levels all modeling capabilities at all classification levels become equally accessible, editable and evolvable as instance data within the tool.

This so called "real-time" (meta-)modeling capability provides tremendous flexibility and evolvability advantages for model-driven development, and makes it easier to use models at run-time to drive the execution of systems in an adaptable and knowledge-driven way. However, it also creates tremendous problems for modelers to keep a model up-to-date whenever changes are made, and to ensure that an ontology (the collection of ontological data across all the ontological levels) remains consistent. Since the number of ontological levels is unlimited, a change to a model element in one ontological level (i.e. a model) could have an impact on a large number of elements over an unlimited number of lower and higher ontological levels. The effective use of this extra flexibility is therefore contingent on the multi-level modeling environment providing dynamic, real-time (i.e. on-the-fly) support for propagating the effects of a change to the affected parts of the ontology. Sometimes this may be performed automatically, but in most cases the tool needs to obtain further input from modelers about the intended effects of changes.

In terms of today's software engineering environments, this capability most closely resembles the idea of refactoring that is used to improve the quality of software engineering artifacts [5]. Refactoring involves the enactment of various kinds of enhancements to a software artifact to change it into a new form. It has traditionally been applied to code or architecture artifacts but there is increasing awareness of the value of applying it at the level of models [12,3] and ontologies [9,7]. However, refactoring as recognized in traditional software engineering environments differs from the on-the-fly changes required in multi-level modeling environments in one important and fundamental aspect - the former are performed with the specific goal of retaining the original meaning of the artifact concerned, while the latter are performed with the specific goal of changing the meaning of a currently invalid ontology to make it valid again. We therefore use the term "emendation" rather than "refactoring" to characterize the process

of evolving a multi-level model, on the fly, to restore it to a state of validity since this precisely captures the intent and nature of the process. Emendation is defined as "an alteration designed to correct or improve" in the Miriam-Webster English dictionary [10]. We believe this therefore represents the most accurate term for describing the process of (and associated techniques for) making changes to an ontology (i.e. a set of ontological levels) in order to bring it back to a state of correctness after a user-induced change.

As well as introducing the notion and goals behind model emendation the contribution of this paper is to present an architecture for automatic emendation support and an initial prototype we have developed to support it. The remainder of this paper is structured as follows: in the next section we first introduce multi-level modeling (Section 2). In the section after that ontology consistency and the resulting requirements for an emendation service are outlined (Section 3). After identifying the ontology consistency requirements, an architecture which can support emendation of multi-level models and our prototypical implementation are presented (Section 4). Afterwards, we show how semi-automatic emendation support can help a modeler on a small example of an online pet store (Section 5). The paper then closes with a discussion of future work (Section 6) and a conclusion (Section 7).

2 Multi-level Modeling

Multi-level modeling supports the creation of ontologies containing an arbitrary number of classification levels (i.e. models) unlike traditional modeling environments like MOF or Ecore, where the number of classification (i.e. meta-) levels is fixed and limited. This is an advantage when a modeler is modeling a domain with more than two inherent classification levels. With traditional approaches a modeler can only capture two levels, the meta-model and the meta-model instance. The key to supporting multi-level modeling is the so called Orthogonal Classification Architecture (OCA) in which (the majority of) model elements have two fundamental types rather than one as illustrated in Figure 1 - an ontological type, defined by the modeled problem domain at L_1, and a linguistic one defined by the level-spanning modeling language at L_2. Traditional modeling environments mix up these two kinds of classification. For example, in Ecore the meta-level M_3 usually describes the available language constructs which are then instantiated at level M_2 and M_1 to capture the domain of interest. In contrast, in a multi-level modeling environment, linguistic and ontological (i.e. domain) classification are separated into two distinct classification dimensions as illustrated in Figure 1. Linguistic classification is indicated through vertically dotted instantiation arrows whereas ontological classification is indicated through horizontally dashed classification arrows. This orthogonality gives the architecture on which multi-level modeling bases its name [1](Orthogonal Classification Architecture). Linguistic classification does not only provide a linguistic type but also linguistic attributes (e.g. potency) called traits. Ontological attributes provided through a model element's ontological type are called attributes.

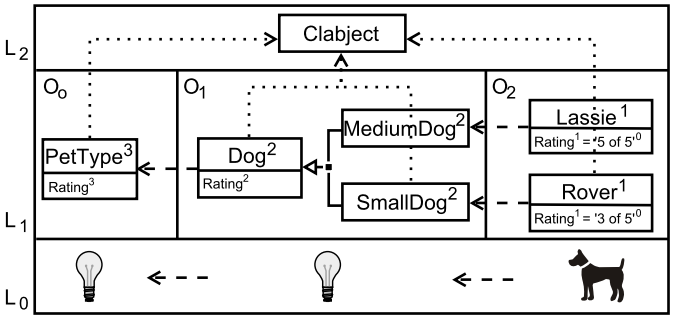


Fig. 1. The pet store web shop ontology

From the diagram in Figure 1 it is clear that model elements residing in the middle levels of ontologies, e.g. Dog, are simultaneously an instance of a type at a higher ontological level, PetType, and types for model elements on lower ontological levels, Lassie and Rover, at the same time. To capture this class/object duality of model elements the term "clabject" which is a composite of the words "class" and "object" is introduced.

Every ontological model element in L_1, the so called ontology, has an annotation in the form of a superscript. This number is the element's potency. Features (attributes and methods) and attribute values also have a potency. The notion of potency allows a clabject to be instantiated over a predefined number of ontological levels, depending on the properties of the subject of the model. Potencies can have either a non-negative integer value or $*$ value representing infinity or unlimited. If a clabject has an integer potency, p, every instance of that clabject at the lower level must have potency p-1. Thus, a clabject with potency 0 can have no instances. A clabject with $*$ potency can have instances with any integer potency or with $*$ potency. The example in Figure 1 shows the potencies of all model elements. All elements at level O_0 have a potency of 3, their instances at level O_1 have potency 2 and their instances at level O_2 have potency 1. Feature potency specifies over how many levels a feature can endure over, i.e. be passed on to instances. That is why this value is also called durability. A feature with durability 1 exists in the following level only, a feature with durability 2 in the following 2 levels and so on. The value potency (mutability) determines over how many levels an attribute can be changed. A value with mutability 0 cannot be changed on the next level while a value with mutability 1 can be changed on the next level. In terms of traditional modeling languages like UML, clabjects with potency 1 correspond to traditional types and clabjects with potency 0 correspond to traditional objects. No equivalent in the UML can be found for clabjects with potencies higher than 1.

Another major difference between multi-level modeling environments and traditional modeling technologies is that one can change all levels of a multi-level model at the same time. Changes on one level immediately effect all other levels. We usually refer to this concept as real-time (meta-)modeling. In traditional meta-modeling environments the meta-model is fixed when editing a meta-model

instance (i.e. a model). Moreover, when editing a meta-model, instances of that model are usually not directly accessible. Because only two levels are ever available for modeling (usually the meta-model and model), we refer to such environments as two-level modeling environments. The limitations of such environments make it difficult for a language engineer to alter more than one level on-the-fly during model creation. When all model levels are equally available for modeling at all times new possibilities for creating, debugging and extending domain-specific modeling languages are created.

Figure 1 shows an ontology which classifies different kinds of pets and their instances. Level O_0 defines the ontological type PetType with Potency 3 which is the basis for all types of pets, e.g. dogs or cats. Having a potency of 3 enables PetType to be instantiated over the next three levels. PetType owns the attribute Rating which expresses the rating of a pet. The durability of 3 states that the rating attribute must exist on the next three levels. The mutability of the attribute is not shown because it is the same as the durability (the default). Thus the mutability of Rating is also 3. The category Dog is instantiated on O_1 as instance of PetType. Dog is further subdivided into SmallDog and MediumDog as subtypes of Dog. These ontological types from O_1 are used to create Rover and Lassie as instances at level O_2. The potency 1 of the elements on O_2 states that these can be instantiated on one more level which is not shown here. In addition to their ontological type every model element in the ontological levels also has a linguistic type, clabject, indicated through the dotted vertical instantiation arrows.

3 Ontology Consistency Semantics

The central premise when emending an ontology after a user-induced change is that the ontology was *correct* before the change occurred and shall be *correct* again afterwards. Two questions are of fundamental importance for the following discussion:

1. What does it mean for an ontology to be "correct"?
2. How can the ontology "break", i.e. which aspects of the correctness can be violated by a change?

We define two concepts for the informal meaning of correctness. First, an ontology is **consistent** if there is no information inside the ontology that contradicts another statement in the ontology. Second, an ontology is **complete** if it is consistent and all the statements inside the ontology are true. The detailed formal definition of all concepts presented in this paper can be found in [6].

The difference between consistency and completeness stems from the maturity of the ontology. If a clabject has potency two, the statement is that there are (or will be) instances of the clabjects instances. If those 2^{nd} order instances are not present yet, the ontology is not *complete*. If there are no contradictions otherwise, it is nevertheless *consistent*. The focus of this paper is to restore ontology **consistency** by applying emendation operations.

3.1 Ontology Consistency

Building up on the informal definition of consistency above, the property of ontology consistency is split up in two parts:

REQ 1: All classification relationships have to be correct, i.e. an instance has to be an instance of its type according to multi-level classification semantics.

REQ 2: All generalization relationships have to be correct, i.e. the classified model (i.e. ontological level) has to respect the claims implied by the boolean traits of the generalization.

Classifications point from the instance to the type and generalizations are statements about the instances of the sub- and supertypes. So for the consistency of one model, the classifications of the models itself as well as the generalizations of the classifying model are relevant. If the subject model is at the top of the model stack, it has no classifications or classifying model. So the top model is always consistent by definition. Figure 2 gives an overview of the constraints.

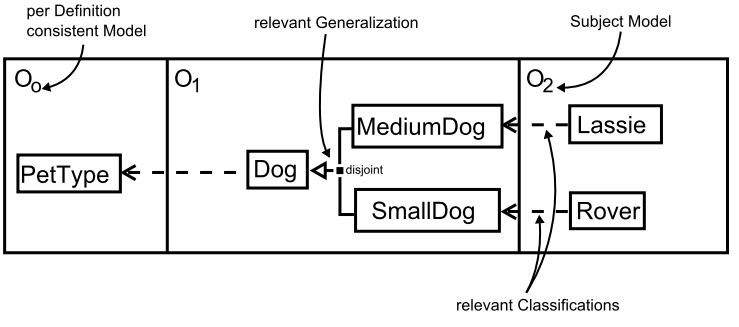


Fig. 2. Ontology consistency dependencies

3.2 Classification Correctness

The correctness of a classification requires the instance to be an instance of the type according to multi-level classification semantics. For an instance to be an instance of the type, it has to define all the properties defined by the type[1] and have a conforming potency. Conforming potency means that the potency is one lower than the type's potency. In case the type's potency is $*$ the instance's potency has to be $*$ or any positive natural number. Having all properties means that for every feature its type defines, the instance has to define a conforming feature. For every mandatory connection the type takes part in, the instance has to take part in a corresponding connection. Connection participation is mandatory if the connection has a potency greater than zero and the multiplicity of the

[1] The notion of instance actually has a more refined meaning, including different kinds of instances depending on whether they define more properties than needed by the type or not. See [6] for details. Strictly speaking the rules presented here are valid only for isonymic classification relationships

other end of the connection is greater than zero. Both features and connection participation are properties that can be inherited from supertypes. To conform a feature has to match the name and conform to the durability of the type's corresponding feature. If the feature is an attribute, the value potency has to conform and if the type's attribute has 0 mutability the value has to be the same. So the requirements of classification correctness can be summarized as:

REQ 1.1: The potency of the instance has to conform to the type's potency.
REQ 1.2: For every feature of the type the instance has to have a conforming feature.
REQ 1.3: For every mandatory connection the type participates in, the instance has to participate in a conforming one.

3.3 Generalization Correctness

The correctness of a generalization requires that the classified domain respects the constraints imposed by the boolean traits of the generalization. Formally, the correctness of a generalization also requires that every instance of the subtype[2] is also an instance of the supertype. This constraint is true by definition as the properties of the supertype are always a subset of the properties of the subtype. A generalization can have three boolean traits: *disjoint, complete* and *intersection.* Although their type is boolean, these traits do not need to be set, this means that a generalization can choose between three alternatives[3]:

1. it can state that it is disjoint,
2. it can state that it is not disjoint or
3. it can make no statement about disjointness.

The difference between the second and the third is that the third does not impose any constraints whereas the second states that the opposite of disjointness is true. So detailed requirements of generalizations are:

REQ 2.1: If the generalization is disjoint there must not be an instance of the supertype that is an instance of more than one of the subtypes. If the generalization is not disjoint, there has to be an instance of the supertype that is an instance of more than one of the subtypes.
REQ 2.2: If the generalization is complete, there must not be an instance of the supertype that is not an instance of any of the subtypes. If the generalization is not complete, there has to be an instance of the supertype that is not an instance of any of the subtypes.
REQ 2.3: If the generalization is an intersection, there must not be an instance of all the supertypes that is not an instance of the subtype. If the generalization is not an intersection, there has to be an instance of all the supertypes that is not an instance of the subtype.

[2] There may be more than one subtype, or more than one supertype.
[3] On the example of disjoint.

4 Suggested Emendation Service Architecture

The architecture for context sensitive ontology emendation support, displayed in Figure 3, consists of 3 components, the multi-level model, the emendation service and the impact analyzer. The emendation service subscribes to changes in the ontology. It is unimportant how the changes to the ontology are actually performed (e.g. via code, graphical editor, tree structure editor etc.). Once the emendation service is notified of a change to the ontology, it asks the impact analyzer to compute all model elements which are effected by this change. For impact analysis, the classification semantics and the resulting requirements presented earlier in this work are used. If the computation reveals an impact on more than the changed model element, the emendation service is triggered. This then suggests operations which can be performed to rectify the situation and asks the user to select which is the most appropriate. Alternatively the user can cancel the actions that are about to be executed by the emendation service. After configuration, the emendation service executes the emendation operations which are needed to keep the model consistent. If no impact on other than the changed model element is calculated by the impact analyzer the emendation services does not act. The architecture shows that the configuration information for the impact analyzer and the emendation service is loosely coupled and can be changed on the fly while using a multi-level modeling environment.

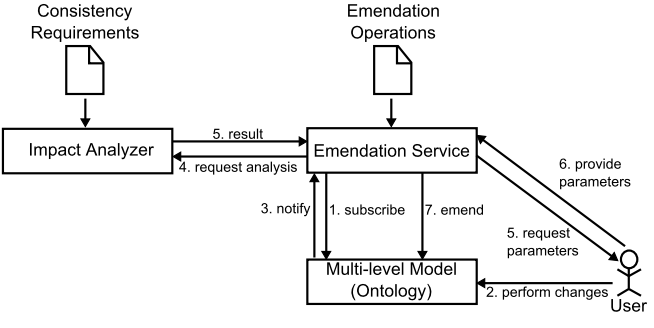


Fig. 3. The suggested emendation service architecture

Some changes to an ontology are hard to support with an automatic emendation service in an appropriate way. One of these cases is the introduction of a connection with a multiplicity with a lower bound higher than “1”. In such a case all instances of the clabject at the opposite end of this connection end must have added a connection to an instance of the clabject at the connection end with the lower bound higher than “1”. It is very hard to automatically guide a user through such a process because, for example, the clabjects to which the instance clabject needs to be connected may not exist at the time of the change. In such cases automated emendation is not recommended and model validation is preferred. After saving the multi-level model, all model elements with inconsistencies are marked and in some cases automatic fixes are provided for the

indicated problems. However, such model validation approaches are beyond the scope of this paper.

At the time of writing, a prototype implementation of this proposed emendation architecture is implemented in the Melanie [13] multi-level modeling tool. The consistency rules and emendation operations are currently hard coded into the tool but this will be changed in future versions. Melanie provides automatic and context sensitive emendation support and a limited number of classification related emendation operations. However, the number of these operations is continually growing. Support for automatically applying recorded emendation operations to other, deployed ontologies (e.g. [9]) is not implemented at the moment. The user does not explicitly need to invoke the emendation mechanism to get assistance while editing a model. All changes to the edited ontology are permanently tracked and evaluated. As soon as a user performs an operation on a classification relationship that requires mandatory changes to more than the currently edited model element, he/she automatically receives assistance from Melanie's emendation service. A dialog or sophisticated wizard, depending on the operation, can be displayed to guide the user through the process of emendation. The collected input from the user is then used to configure the emendation service before execution. If the user does not want any assistance he can simply switch off the emendation support via the user interface. In addition to the emendation service Melanie provides basic model validation services and operations to fix errors which are invoked when a multi-level model is saved.

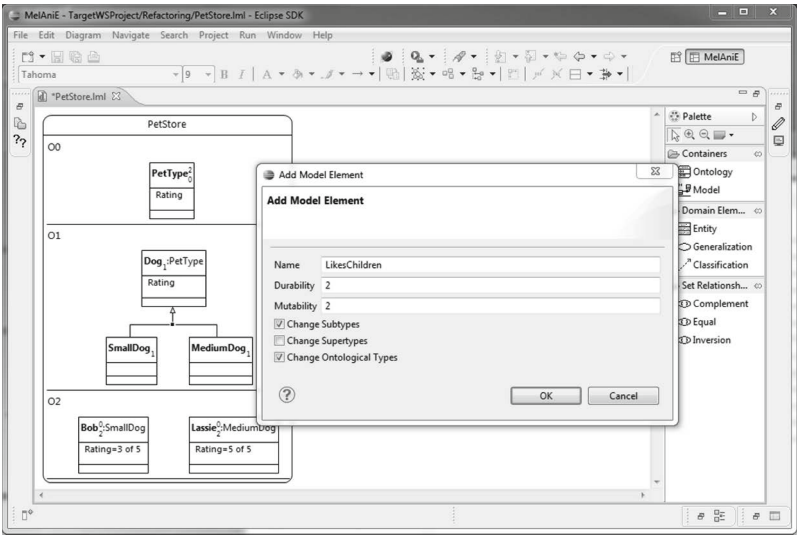


Fig. 4. Screenshot of the emendation support offered by Melanie. The attribute LikesChildren is added to MediumDog.

Figure 4 shows the dialog that a user receives, after performing an operation which effects more than the directly changed model-element. Here the user added the feature LikesChildren to the clabject MediumDog. The impact analyzer has noticed that this change can potentially effect the instances of MediumDog which is Lassie in this case. Additionally, this could also effect the superclass Dog. If the user also wants to change the clabject Dog its ontological type PetType, and the instances of SmallDog namely Rover are effected as well. Hence, the displayed dialog offers the options "Change Subtypes", "Change Supertypes", "Change Ontological Types" to configure the emendation service. In this case the user decides to not change the supertype which is indicated by the not selected "Change Supertypes" option in the "Add Model Element" dialog. After selecting "OK" the emendation operations are performed. If the user selects "Cancel" and explicitly states by doing so that no action is desired, the emendation service will perform no action. The example shown here is quite simple. More extensive support in the form of wizards guiding a user through a multi-staged emendation process can be provided if appropriate.

5 Case Study: Emendation of an Online Pet Store

In this section we show how multi-level model emendation services can help a modeler keep an online pet store [2] correct in the face of ongoing changes. The chapter is organized according to refactoring operation categories proposed by Opdyke [11]: "Creating a Program Entity", "Deleting a Program Entity", "Changing a Program Entity" and "Moving a Member Variable". The following list of emendation operations is not exhaustive but rather a starting point illustrating the need and use of emendation operations. However, the operations in the following subsections where chosen to cover the most common operations which are executed on an ontology during evolution. Figure 5 shows the pet store ontology described in the previous multi-level modeling introduction. A shop sells different PetTypes. In the beginning only Dogs are sold. These are divided into MediumDog and SmallDog with two instances - Lassie and Rover. This ontology builds the basis for the web store because all content of the web store is generated out of this ontology. Therefore, changing the ontology changes the content in the web store. Throughout the case study changes in the pet store's environment need to be reflected in the ontology. After a short explanation of the reason for the change, the nature of the applied abstract change

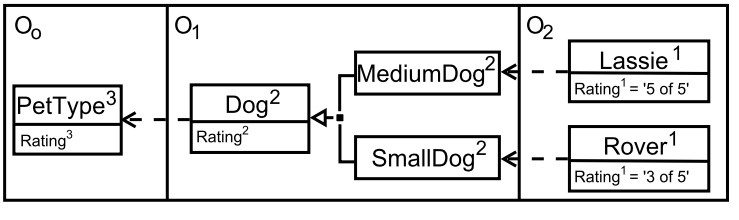


Fig. 5. The initial pet store ontology

operation (highlighted in italics) applied is described, the violated requirements are enumerated and the automated steps suggested by the emendation service are elaborated.

5.1 Adding New Pets and LikesChildren Attribute - Creating a Program Entity

The web store decides to sell dogs as well as cats. Thus on O_1 Cat is instantiated from PetType and two subclasses HairyCat and GroomedCat are added. From these two subclasses Max and Moritz are instantiated. *Adding a new clabjects* does not violate any ontology consistency rules as the new model elements do not participate in a classification relationship or effect any model elements participating in a classification relationship. *Adding new clabjects through instantiation* does not effect the classification relationship between a type and the new instance as all rules for classification are automatically satisfied by the instantiation operation. Thus, the impact analyzer calculates no impact on the changed ontology and does not invoke the semi-automated emendation support. However, the process of instantiating an instance from a type can be interpreted as an emendation operation itself, because a new model element together with a classification relationship are created and setup to automatically fulfill all subrequirements of REQ1.

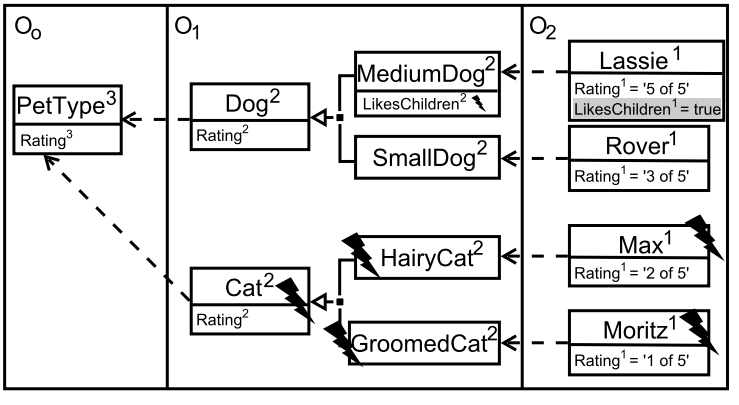


Fig. 6. The pet store ontology after adding cats and the LikesChildren attribute

Furthermore, some customers reported that it is useful to know if a MediumDog is suitable for parents. Hence, the pet store owner decides to add the LikesChildren attribute to the MediumDog model element. The *introduction of new features* can violate the classification relationship consistency rule REQ1.2 which states that an instance needs to have one conforming feature for each feature of the type. Changing the number of features, whether by adding or removing them, breaks this requirement for the clabject in the role as instance and type. The clabject's types need to have the feature added so that the clabject

has one conforming feature for each feature of the type and the instances need to have the new feature added for the same reason. In this case the impact analyzer detects a violation between MediumDog and Lassie because MediumDog now owns the feature LikesChildren which is not owned by Lassie. The user is automatically assisted in fixing this by adding the attribute to all instances of MediumDog. MediumDog is not involved in any other classification relationships, either as type or instance so no further changes are needed. Figure 6 shows the resulting ontology with cats and the LikesChildren attribute added.

5.2 Changing the Potency of PetType, LikesChildren and Rating - Changing a Program Entity

Later the pet store decides that no instances of model elements at level O_2 are needed. Hence, the potency of PetType is changed from 3 to 2 to fix the number of available model levels to three. A *change to the potency of a clabject* violates REQ1.1 which defines that an instance's potency has to conform to the potency of its type. Such a change means the model element's potency does not conform to the type's potency anymore and the instance's potency does not conform to the model element's anymore. The impact analyzer detects this change to the potency of PetType and calculates that a change to the potency violates the classification relationships it takes part in. Thus, it offers to change the potency of all model elements which are instances of PetType - Dog, Cat, MediumDog, SmallDog, HairyCat, GroomedCat, Lassie, Rover, Max, Moritz.

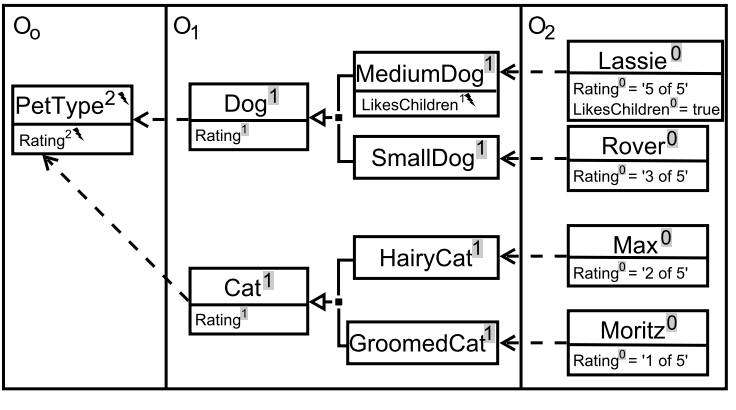


Fig. 7. The Ontology after changing the potency of PetType, Rating and LikesChildren

Additionally the pet store owner decides to also change the durabilities of Rating and LikesChildren. *Changing traits of a feature* violates REQ1.2 in relation to name, durability, mutability and value because these are used to define the conformance of features. By changing one of these the number of conforming features between a model element and its types and instances changes. This can

be fixed by changing all features that conformed before the change so that these still conform after the change. The potencies are recalculated and changed as shown in Figure 7.

5.3 Deleting Rating and HairyCat - Deleting a Program Entity

After having added cats to the pet store, the owner notices that these are very poorly rated by users which is not good for his business. This leads to the decision to remove the Rating attribute from the pet store. To do so the attribute is removed from the PetType model element in the store's ontology. *Deleting a feature* can violate the classification relationship requirement REQ1.2 as the number of features changes. The clabject now has fewer features than its type and the instances have more than the changed clabject. Again this is detected by the impact analyzer which calculates that the consistency rules for all instances of PetType are violated. To restore consistency after the change to the ontology, the emendation service offers to automatically remove the attribute from Dog, Cat, Lassie, Max and Moritz.

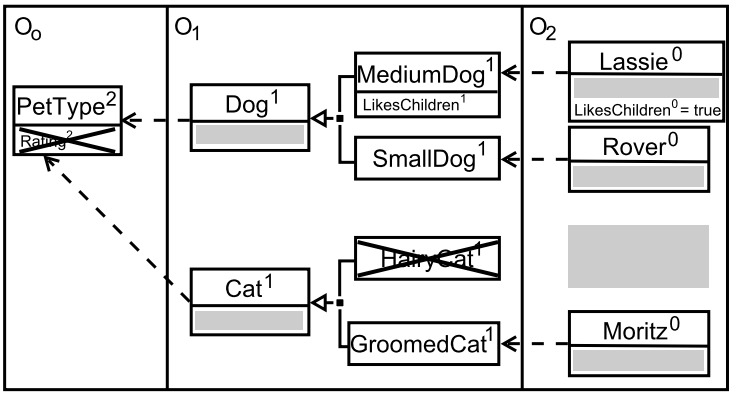


Fig. 8. The pet store ontology after removing the Rating attribute

To avoid the danger of accidents during the approaching New Years Eve the pet store owner decides to temporarily remove hairy cats from the web store. This leads to the removal of HairyCat. *Deleting a clabject* does not violate any classification requirements as long as it does not provide attributes to instances by taking part in an inheritance relation ship. This would lead to a violation of REQ1.2 which could for instance be fixed by deleting the feature provided through the clabject to the instances. In this case HairyCat does not provide any attributes through inheritance thus no emendation operation for preserving classification semantics need to be enacted. However, the emendation service could notice the deletion of HairyCat and offer to also delete its instances to save the modeler the manual editing effort. Figure 8 shows the new ontology without the Rating attribute, the HairyCat clabject and its instance Max.

5.4 Moving LikesChildren - Moving a Member Variable

After various complaints about incidents between cats bought in the pet store and children of customers, the pet store owner sees the need to also capture whether cats are suitable for children. Hence the already existing attribute LikesChildren is moved from MediumDog to PetType. A *move operation of a feature or clabject* is a delete operation on the source and an add operation at the target. Thus the previous presented emendation operations for creation and deletion of a program entity can be applied as the same requirements for classification relationships are violated. The impact analyzer detects the violations introduced through the delete and add operation and notifies the emendation service that all instances of PetType violate at least one of the classification relationship consistency requirements. To fix this, the emendation service offers to add the attribute LikesChildren to Dog, Cat, Rover, Max and Moritz. The resulting ontology is shown in Figure 9.

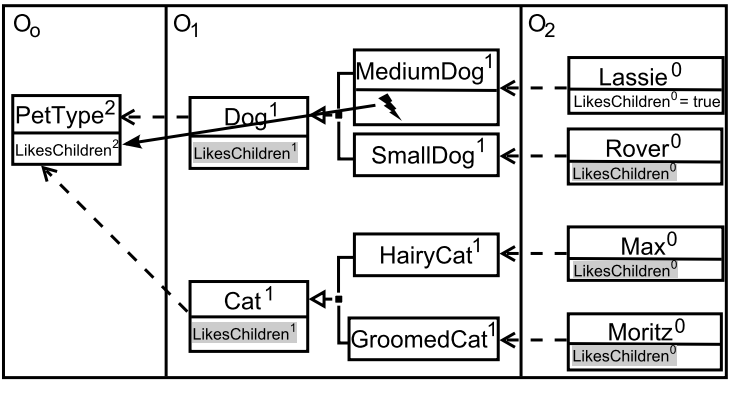


Fig. 9. The pet store ontology after moving the LikesChildren attribute and Lassie clabject

6 Future Work

This paper has presented an overview of our initial realization of the idea of on-the-fly emendation support for multi-level models. A method to flexibly configure the emendation service and impact analyzer needs to be developed. Furthermore, a study on how to most effectively support modelers during the change of a model is required. It is important to provide modelers with as much support as possible but not in a way that the automated emendation service decreases their efficiency.

When considering the evolution of a model the question of how to transport these changes to other models arises. We believe the multi-level modeling approach described in this paper comes closer to the notion of "mogram" defined by Kleppe [8] than any other meta-modeling approach available today. In short,

a mogram is a software program written using a modeling language instead of a programming language. This claim is based on the fact that our multi-level models are completely self contained DSL-based packages of information which can exist side by side in the same execution environment. The traditional modeling approach with a centralized meta-model requires models to be "deployed" to a central model repository and reconfigured before they can be interpreted. In our approach a DSL can be used in a multi-level model environment without configuration or deployment of any separate meta-model. This is similar to starting e.g. a Java program in the Java Virtual Machine. In this case no prior setup of the VM is needed and different versions of a program can run side by side. In this sense multi-level models are closer to mograms than traditional two-level models. However, treating every model as a closed piece of software also has a disadvantage. All models basing on the same version of a language can differ from each other as they do not have a central and fixed meta-model. Thus a mechanism to transport the refactoring operations to other deployed multi-level models is needed. We are currently investigating the recording of refactoring changes and automatic application to other multi-level models.

7 Conclusions

The paper presents initial investigations into the provision of on-the-fly emendation support for multi-level models (i.e. ontologies) based on the orthogonal classification architecture. In contrast to today's model refactoring technologies which only support the changing of meta-models and the subsequent application of these changes to model instances the approach presented here modifies all levels of a model simultaneously. A second novelty of the approach is that a model, whilst being edited, is continuously monitored for changes so that emendation support operations can be proactively suggested by the emendation service. In contrast, traditional model refactoring approaches require modelers to explicitly request refactoring support. These changes are then recorded and transported to other model instances. To do so current approaches need to generate a cumbersome model transformation which updates the dependent model levels in a subsequent, decoupled refactoring step. In conclusion, we hope these initial investigations into on-the-fly emendation will provide the foundation for a much more sophisticated automated emendation support that can make the vision of truly real-time meta-modeling become a reality.

References

1. Atkinson, C., Gutheil, M., Kennel, B.: A Flexible Infrastructure for Multilevel Language Engineering. IEEE Transactions on Software Engineering (2009)
2. Basler, M., Brydon, S., Nourie, D., Singh, I.: Introducing the Java Pet Store 2.0 Application (2007), http://java.sun.com/developer/technicalArticles/J2EE/petstore/

3. Brosch, P., Seidl, M., Wieland, K., Wimmer, M., Langer, P.: The operation recorder: specifying model refactorings by-example. In: OOPSLA Companion, pp. 791–792 (2009)
4. Eclipse Foundation: Edapt (2012), http://www.eclipse.org/edapt/
5. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, Boston (1999)
6. Kennel, B.: A Unified Framework for Multi-Level Modeling. Ph.D. thesis, University Mannheim (2012)
7. Klein, M., Noy, N.F.: A component-based framework for ontology evolution. In: Workshop on Ontologies and Distributed Systems at IJCAI 2003 (2003)
8. Kleppe, A.: Software Language Engineering: Creating Domain-specific Languages Using Metamodels. Addison-Wesley (2009)
9. Maynard, D., Peters, W., Sabou, M., dÁquin, M.: Change management for metadata evolution. In: International Workshop on Ontology Dynamics (IWOD) ESWC 2007 Workshop (2007)
10. Miriam-Webster: Definition of Emendation (2012), http://www.merriam-webster.com/dictionary/emendation
11. Opdyke, W.F.: Refactoring object-oriented frameworks. Ph.D. thesis, Champaign, IL, USA, uMI Order No. GAX93-05645 (1992)
12. Reimann, J., Seifert, M., Aßmann, U.: Role-Based Generic Model Refactoring. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part II. LNCS, vol. 6395, pp. 78–92. Springer, Heidelberg (2010)
13. University of Mannheim - Software Engineering Group: MelaniE - Multi-level modeling and ontology engineering Environment (2012), http://www.eclipselabs.org/p/melanie

Specifying Refinement Relations in Vertical Model Transformations*

Jan Rieke** and Oliver Sudmann

University of Paderborn, Heinz Nixdorf Institute,
Zukunftsmeile 1, 33102 Paderborn, Germany
{jrieke,oliversu}@uni-paderborn.de

Abstract. In typical model-driven development processes, models on different abstraction levels are used to describe different aspects. When developing a mechatronic system, an abstract system model is used to describe everything that is relevant to more than one of the disciplines involved in the development. The discipline-specific implementation is then carried out using different concrete discipline-specific models.

During the development, changes in these discipline-specific models may affect the abstract system model and other disciplines' models. Thus, these changes must be propagated to ensure the overall consistency. Bidirectional model transformation and synchronization techniques aim at automatically resolving such inconsistencies.

However, most changes are discipline-specific refinements that do not affect other disciplines. Therefore, vertical model transformations also have to take into account that these refinements must not be propagated. Current model transformation techniques, however, do not provide sufficient means to specify and detect whether a change is just a refinement.

In this paper, we propose a way to formally define such refinements. These definitions are then used by the model transformation engine to automatically synchronize models of different abstraction levels.

Keywords: Vertical Model Synchronization, Triple Graph Grammars (TGG), Refinement/Abstraction, Mechatronic System Design.

1 Introduction

The development of mechatronic systems, from modern household appliances to transportation systems, requires the close collaboration of multiple disciplines, such as mechanical engineering, electrical engineering, control engineering, and software engineering. First, an abstract, discipline-spanning *system model* is created by an interdisciplinary team of engineers. Next, this system model is transformed into different concrete, *discipline-specific models*, which engineers from

* This work was developed in the course of the Collaborative Research Center 614 – Self-optimizing Concepts and Structures in Mechanical Engineering – University of Paderborn, funded by the Deutsche Forschungsgemeinschaft.

** Supported by the International Graduate School Dynamic Intelligent Systems.

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 210–225, 2012.

each discipline now alter to implement the system. As changes to a discipline-specific model may affect the discipline-spanning system model and other disciplines' models, avoiding inconsistencies is crucial.

To automatically synchronize the different models used during the development, a concept is needed to bidirectionally propagate changes between the different models. If, for instance, one discipline-specific model is significantly changed, these changes must be propagated to the system model, and from there to the other discipline-specific models. Bidirectional model-to-model transformation techniques are a promising approach for such scenarios.

However, not all changes affect the overall consistency. If an engineer performs a change to their discipline-specific model, such a change may either be a *discipline-specific refinement*, which must not be propagated to other models, or a *discipline-spanning relevant change*, which also affects the system model and other disciplines' models. This is due to the differing levels of abstraction between the system model and the discipline-specific models: For one abstract model, there exist several consistent concrete models. In other words, in a consistency mapping from a system model to a discipline-specific model, discipline-specific refinements are all changes that can be performed on the discipline-specific model so that it still corresponds to the system model. Thus, such a mapping from an abstract model to a more concrete model is a 1-to-n mapping.

In a conceptual view, we have an abstract language A (the system model's meta-model) and a concrete language B (the discipline-specific meta-model).[1] To transform a word $a \in A$ to a word $b \in B$, we use an initial transformation function $I \subseteq (A \to B)$. However, as B is more concrete than A, a consistency relation R contains more elements than I and is not a function: $I \subseteq R \subseteq (A \times B)$. An operation $op \in (B \to B)$ is a consistency-preserving refinement iff $\forall a \in A, b \in B : (a, b) \in R \Rightarrow (a, op(b)) \in R$, i.e., both the concrete model before and after the operation map to the same abstract model.

Therefore, when defining such a vertical[2] model transformation, we also have to consider non-functional consistency relations. Existing model transformation approaches (e.g., [16,7,11]), however, do not provide sufficient support for that, because they mostly work for functional relations only. Even if the transformation language allows specifying non-functional mappings, it is not well supported by the synchronization algorithm. Furthermore, it is time-consuming and error-prone to define all possible refinements directly in the consistency relation by hand, and doing so makes the consistency relation difficult to maintain.

Thus, we suggest an inductive approach: We take the functional transformation relation I as a fixed input, manually define a set of consistency-preserving refinement operations, and combine both to compute the consistency relation R. In practical scenarios, such an approach is more flexible, because the

[1] We use "language" and "meta-model" (as well as "word" and "model") interchangeably here. For a more formal comparison of both concepts, see Amelunxen and Schürr [1].

[2] Horizontal transformations map between models of the same abstraction level, vertical transformations map between models of different abstraction levels [15].

consistency-preserving refinement operations can be defined by discipline experts who do not need to know the transformation language. Only the initial, functional transformation I is defined by a transformation engineer.

To sum up, our approach works as follows. First, we formally define an initial transformation function I, and discipline-specific refinements in terms of in-place model transformation rules. Each of these so-called *refinement rules* describes a change to a model that is considered to be a refinement operation. Second, we apply these refinement rules on the initial, functional transformation relation I, generating an altered consistency relation R that also covers these refinement operations. To perform model synchronization with such non-functional consistency relations, we present an improved synchronization algorithm based on Triple Graph Grammars (TGGs) [17], a rule-based formalism for declaratively specifying relations between models.

The paper is structured as follows. The running example is presented in Sec. 2. Furthermore, we give details about the development of mechatronic systems and the models and tools in use. In Sec. 3, we describe the foundations of the model synchronization technique we use. In Sec. 4, we introduce the language to define refinements and explain how to derive the consistency relation R. The required extensions to the model synchronization algorithm are described in Sec. 5. Finally, we discuss related work in Sec. 6 and conclude the paper in Sec. 7.

2 Running Example

As an example, we consider the *RailCab* research project[3]. Its vision is that, in the future, the schedule-based railway traffic will be replaced by small, autonomous RailCabs, which transport passengers and goods on demand, being more energy efficient by dynamically forming convoys.

When developing a mechatronic system, a team of engineers from all involved disciplines (mechanical engineering, electrical engineering, control engineering, and software engineering) starts developing an abstract system model. Here, an interdisciplinary specification language called CONSENS [5] is used.

Fig. 1 shows parts of the RailCab's *active structure*, which is part of the system model and shows of which elements the system consists and how these system elements interact. RailCabs can communicate with each other using the Communication Module, allowing negotiating the formation of convoys. When forming a convoy, all following RailCabs have to change the control strategy for the velocity to avoid collisions: Instead of using the Velocity Controller that uses a reference speed v^*_{RailCab} and the actual speed v_{RailCab} to calculate the acceleration force F^*, RailCabs now use the Distance Controller that uses a reference distance d^* and the actual distance d to the preceding RailCab as input [12].

A state diagram in the system model specifies the communication protocol to negotiate convoys (Fig. 2). When a RailCab in noConvoy state receives a createConvoy message from another RailCab approaching from behind, it switches to the convoyLeader state in at most 500 ms. Vice versa, a RailCab may form a

[3] Neue Bahntechnik Paderborn/RailCab: http://www-nbp.uni-paderborn.de/

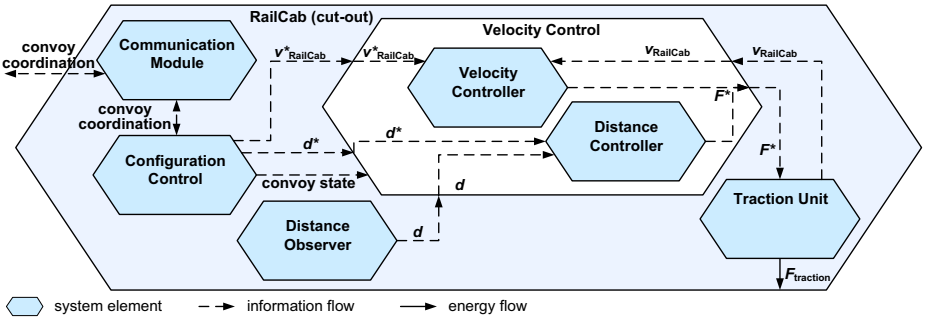


Fig. 1. Parts of the active structure of the RailCab system

convoy with a RailCab in front by sending a createConvoy message and switching to convoyFollower. Convoys may be canceled by a breakConvoy message.

The different disciplines use this system model as the basis for their discipline-specific refinements and implementation. During the development, however, inconsistencies between the system model and the different discipline-specific models may arise. Consider the following process as an example.

1. The discipline-specific models are generated from the system model by different *initial* model transformations. Initial MATLAB/Simulink and Stateflow models are derived for the control engineering. Software engineers use MechatronicUML models [10] for defining the structure and behavior for the discrete parts of the software, especially the communication behavior.
2. The disciplines' engineers start refining their models. E.g., the control engineer defines how to switch between the two control strategies. Due to safety and comfort reasons, sudden steps in the acceleration force F^* must be avoided. Thus, this reconfiguration of controllers requires some time. Therefore, so-called fading states are introduced in the control engineering models in which the actual reconfiguration takes place. This change does not affect other disciplines. Therefore, it must not be propagated to the system model.
3. The software engineer identifies a weakness within the original behavior: The convoy negotiation protocol does not allow the leader RailCab to reject a convoy proposal for safety reasons, e.g., when transporting dangerous cargo. Thus, the behavior in the software model is extended by modifying the corresponding state diagram, now allowing the rejection of convoys.

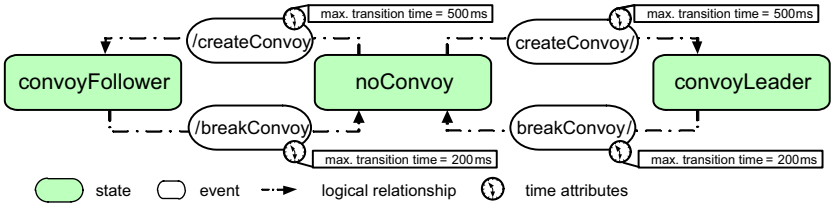


Fig. 2. State diagram describing the convoy communication behavior of the RailCab

4. Next, this modification is propagated to the system model.
5. Finally, this modification is propagated to the control engineering models, retaining the discipline-specific refinements of step 2.

There are two challenges in this process. First, the added fading states in step 2 are just discipline-specific refinements, as they do not affect other disciplines. Thus, they must not be propagated to the abstract system model. However, existing model transformation techniques would simply propagate these changes, as they do not recognize them as refinements. Second, in step 5 some parts of the control engineering model have to be modified according to the changes in the system model. However, the changed state diagram of the system model cannot be simply copied to the control engineering model, as this target model has already undergone some changes in the meantime which must not be overwritten (the addition of fading states in step 2). The challenge is to update the model in a way that these discipline-specific refinements are not destroyed or become invalid, but are reasonably integrated with the changes from the system model.

Fig. 3 shows in detail how the different behavioral models evolve during this development process (the MechatronicUML model and the transformation to it are not important for the comprehension of this paper and have been removed for presentation purposes). First, discipline-specific models are generated from the system model using Triple Graph Grammar transformations (step 1, also marked with ① in Fig. 3). For software engineering, a MechatronicUML model is generated, which contains a Real-Time Statechart that specifies the behavior for the convoy management (① left). For control engineering, we generate initial MATLAB/Simulink and Stateflow models, e.g., a Stateflow chart for the convoy management (① right). As the meta-modeling concepts for state-based behavior are similar in the CONSENS language and the Stateflow language, this transformation is straightforward.

Before we explain the rest of this process in Sect. 4 and 5, we give an introduction to Triple Graph Grammars, the model transformation language we use to define the mapping to the disciplines' models.

3 Foundations of Triple Graph Grammars

Bidirectional model transformation techniques are a promising approach for automatically synchronizing the different models during the development. Here, we use a concept called *Triple Graph Grammars* (TGGs) [17]. TGGs are a rule-based formalism that allows us to specify how corresponding graphs or models can be produced "in parallel" by linking together two graph grammar rules from two different graph grammars. More specifically, a TGG rule is formed by inserting a third graph grammar rule to produce the so-called *correspondence* graph that links the nodes of the other two graphs. Thus, a TGG is a graph grammar that defines a language of corresponding graph triples. TGGs can be interpreted for different transformation and synchronization scenarios. Before we describe these scenarios, let us consider the structure of TGG rules.

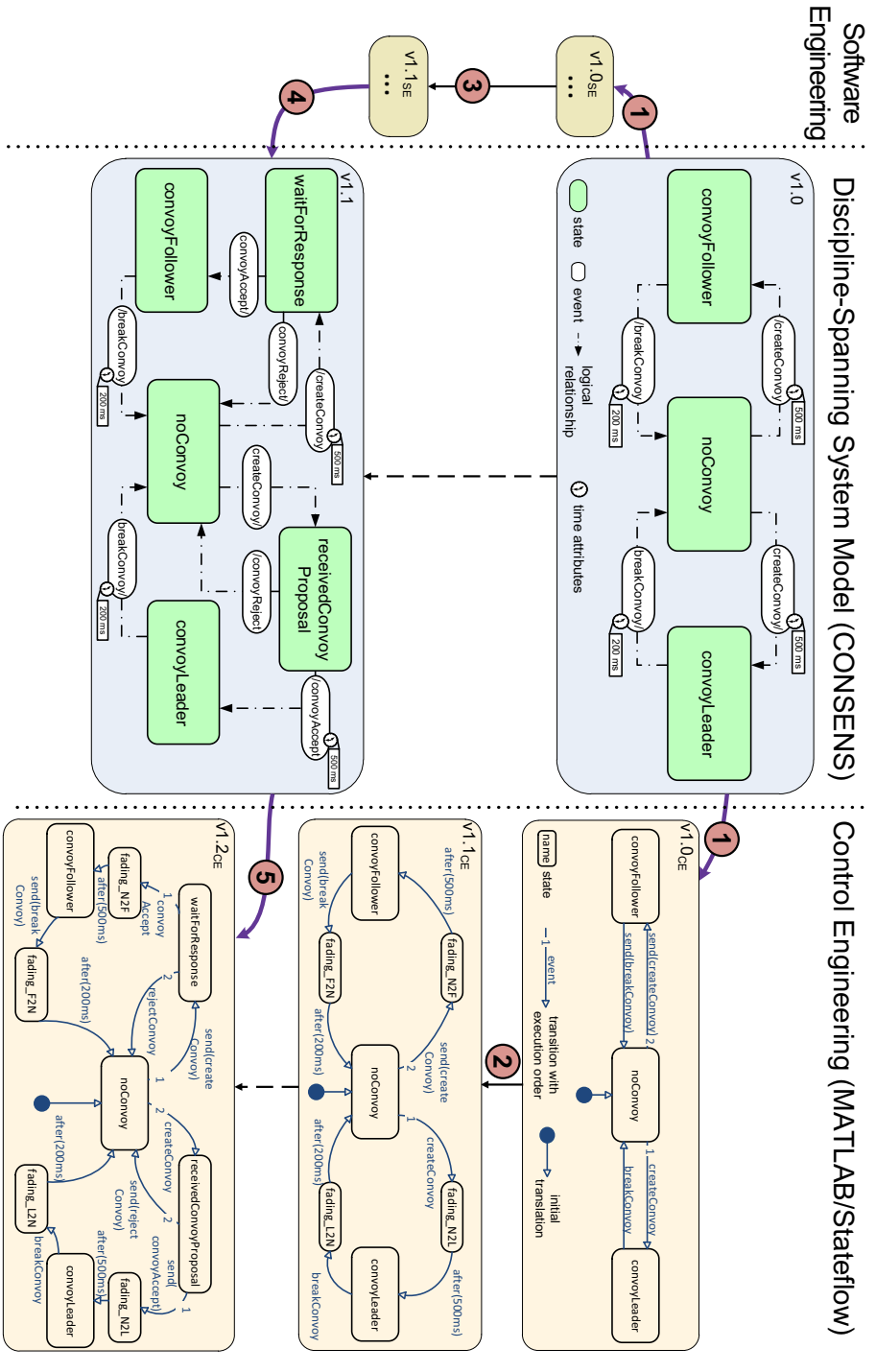


Fig. 3. Evolution of the different models during the development process

3.1 Triple Graph Grammar Rules

Fig.4 illustrates a TGG rule, State to State, which is taken from a TGG that defines the mapping between CONSENS and MATLAB/Stateflow.

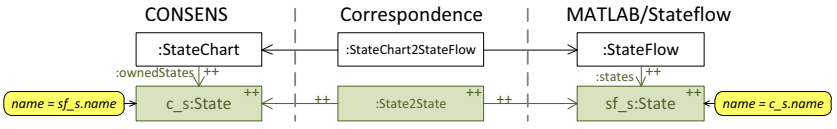


Fig. 4. TGG Rule State to State

TGG rules are non-deleting graph grammar rules that have a left-hand side (lhs) and a right-hand side (rhs) graph pattern. The nodes appearing on the lhs and the rhs are called *context nodes*, displayed by white boxes. The nodes appearing on the rhs only are called *produced nodes*, displayed by green boxes, labeled by "++". Accordingly, there are *context edges*, displayed by black arrows, and *produced edges*, displayed by green arrows and "++" labels.

In TGGs, graphs are *typed* and *attributed*. When working with models and meta-models in terms of MOF, this means that the host or *instance* model contains objects and links that are instances of classes and references of a given meta-model. Accordingly, the nodes and edges in the rules are typed over the classes and references in a meta-model. Nodes are labeled in the form "*Name*:*Type*". For instance, the nodes in the left column of rule State to State are typed by the classes StateChart and State from the CONSENS meta-model. The edge is typed over the reference ownedStates.

The columns of a TGG rule describe model patterns of different meta-models and are called *domains*. The left-column production states that when there is a StateChart in CONSENS, we can add a State and a link between them. The right column of the rule represents the graph grammar production for creating States in MATLAB/Stateflow. In the middle, there is the production of the correspondence structure between the models.

Our TGG rules further introduce the concept of *attribute constraints* and *application conditions* (depicted by yellow, rounded rectangles in Fig.4). Attribute constraints are attached to nodes and have expressions of the form $\langle prop \rangle = \langle expr \rangle$, where $\langle prop \rangle$ is a property of the node's type class, and $\langle expr \rangle$ is an OCL expression that must conform to the type of $\langle prop \rangle$. Node names can be used as variables in the OCL expression. Attribute constraints constrain the attribute value of an object. E.g., rule State to State has two constraints that express that a state's name has to be equal to the name of the opposite state.

We defined a set of TGG rules to transform between CONSENS and MATLAB/Stateflow. Rule State to State (Fig.4) defines how CONSENS states correspond to Stateflow states. Rule Transition to Transition (Fig.5) describes how transitions between states in CONSENS map to transitions in Stateflow. The maximum duration of a transition is represented by an annotation in Stateflow.

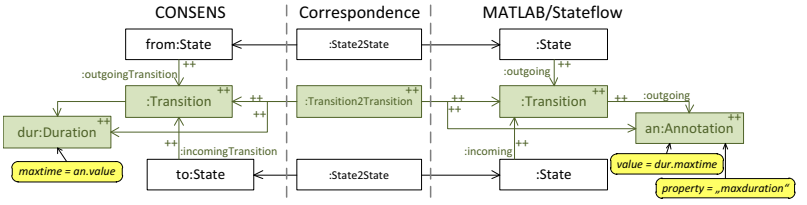


Fig. 5. TGG Rule Transition to Transition

3.2 Application Scenarios

A TGG defines a language of corresponding graph (or model) triples. However, we want to use TGGs for a model-to-model transformation. To do so, we can interpret TGGs for different *application scenarios*. One scenario, called *forward transformation*, is to create one "target" graph corresponding to a given "source" graph. In this case, to apply a TGG rule, it is *interpreted* as follows: First, the context pattern of the rule is matched to *bound* model elements, which are objects and links that were previously matched by another rule application. Second, the source produced pattern is matched to yet *unbound* parts in the source model. If a matching respecting these conditions is found, the produced target and correspondence patterns are created. Doing so, the final transformation result is a valid graph/model triple that is an element of the language defined by the TGG. The *backward* direction works accordingly, reversing the notion of source and target. We refer to Greenyer and Kindler [8] for further details on TGGs and the binding semantics.

Our TGG engine interprets attribute constraints (of the form $\langle prop \rangle = \langle expr \rangle$) as assignments in the target domain. If a TGG rule shall support both transformations directions, assignments must be specified for both directions.

In a situation where a triple of corresponding models is given and a change occurs in one domain model, this change can be propagated by *incrementally updating* only the affected parts of the model. This is also called *model synchronization*, and in general works in two steps: First, for every changed or deleted element in the source model, we check whether the rule application that translated this element is still valid. If it is not valid any more, the rule application is revoked by deleting the produced elements in the target model and removing the bindings of the source produced elements. Second, for every source model element that is not bound yet (i.e., elements that were added or whose bindings have been removed in the first step), we try to apply new rules as in a regular forward transformation. This is necessary to ensure that the synchronization result is again a valid graph/model triple. The backward direction works accordingly.

4 Defining Refinements

In step 2 of the process, the control engineers implement the controllers using MATLAB/Simulink and Stateflow. Especially, they modify the Stateflow model

by incorporating additional states which describe the fading behavior when switching between the controller configurations (② in Fig.3). Such a change is considered a discipline-specific refinement, as it does not affect other disciplines. Therefore, it must neither be propagated to the discipline-spanning system model nor to the other disciplines. However, when using existing model transformation techniques, these additional states would be nevertheless propagated back to the system model: When synchronizing the models, the TGG Rule State to State (see Fig.4) is applicable for the new intermediate states. The TGG rule Transition to Transition is also applicable for the new transitions.

A transformation engine can deal with hierarchical refinements (like adding sub-states or subcomponents) by simply ignoring everything "below" an existing element. We described in our previous work [6] how this can be achieved using *relevance annotations* to mark elements subject to the transformation. However, for complex, non-hierarchical refinements as described above, this is not sufficient. Thus, we need another means to specify refinements. We propose that discipline experts define a set of *refinement rules* that describe which kinds of changes to a discipline-specific model are regarded as discipline-specific refinements. Generally, a refinement rule formally describes a refinement by a precondition (left-hand side) and a replacement (right-hand side).

Fig.6 shows a refinement rule in concrete syntax which defines that adding an intermediate state is a discipline-specific refinement. It describes that a transition may be replaced by a combination of a transition, a state and another transition. In addition, it is specified by a constraint that the new state and transitions must not violate the maximum duration of the original transition. Furthermore, no other incoming or outgoing transitions are allowed for the intermediate state.

This refinement rule covers the addition of the fading states. Using this rule, we can add this refinement to the consistency relation R, so the model synchronization can detect that adding the fading states is a refinement. However, as described later, it is important to store the information that a refinement took place, i.e., that the transition createConvoy/ in the system model (v1.0 in Fig.3) now corresponds to the transition-state-transition combination in the control engineering model ($v1.1_{CE}$).

Basically, a refinement rule is a graph transformation rule. When choosing the language to define refinements, we sought to cover as many refinements as possible on the one hand and, on the other hand, not making the language

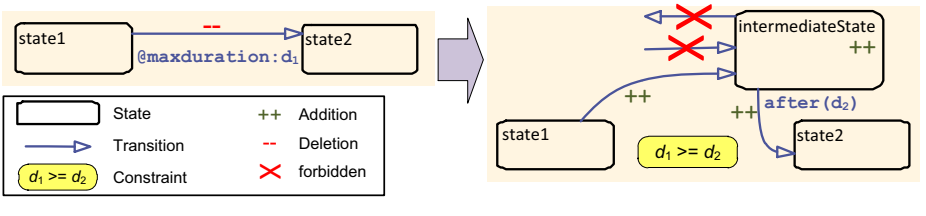


Fig. 6. Refinement rule (concrete syntax) for adding intermediate states in the Stateflow control engineering model

too complex to make analyses impractical. We identified several different refinements from different disciplines (e.g., fault-tolerance patterns like triple modular redundancy, functional partitioning of components, load balancing) which can be described in terms of such graph transformation rules. However, it remains to be investigated further whether we may need a more sophisticated language for other refinements which we have not identified, yet.

Our goal was that these refinement rules should be integrated into the consistency relation R, so that the model transformation engine itself can deal with refinements without fundamental changes to the synchronization algorithm. In this way, formal properties of TGGs like correctness or completeness are still valid and we do not have to heavily modify the existing synchronization tool. We therefore add the information from the refinement rules to the TGG rule set that defined the initial transformation I, creating an altered TGG rule set for the consistency relation R.

The basic idea is to check where refinement rules match in the original TGG rules in the target domain. Whenever a refinement rule's precondition can be found in a TGG rule, we create a copy of that TGG rule and apply the refinement rule in this TGG rule copy. In this way, we derive new TGG rules which map the same source pattern to the refined target pattern. Consider the refinement rule from Fig. 6. This refinement rule's precondition (left-hand side) matches in the target domain of the TGG rule Transition to Transition (Fig. 5). We now copy that TGG rule and apply the refinement rule onto its target domain. That means that we delete every node and edge from the TGG rule which match deleted elements in the refinement rule, and create new nodes and edges for everything that is created by the refinement rule. Furthermore, we create constraints in the new TGG rule for constraints in the refinement rules. Fig. 7 shows the resulting refined TGG rule. This new TGG rule now matches whenever a refinement according to the refinement rule took place in the target model. This rule matches at the respective refined model elements; thus, the models are consistent in terms of the new synchronization rule set.

Next, we describe how the improved synchronization applies this relation R and deals with subsequent incremental updates that may affect refinements.

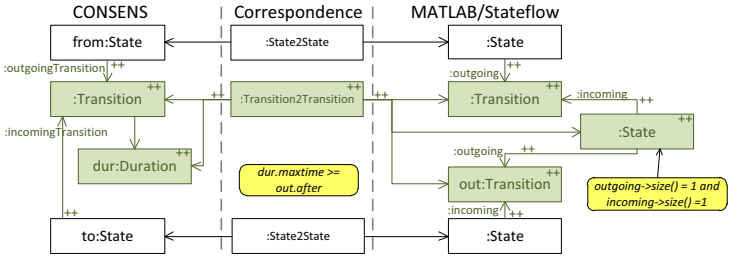


Fig. 7. TGG Rule Transition to Transition (refined, with intermediate state)

5 Model Synchronization with Refinements

Let us have a look at how the improved model synchronization algorithm deals with refinements that are introduced to the models. Fading states were added to the Stateflow model (② in Fig.3). As the model was changed, a (backward) synchronization is triggered to propagate the change to the system model, and from there to all other affected disciplines.

As described in Sect.3, model synchronization algorithms work in two steps: First, for everything that has been deleted (or inconsistently changed) in one model, the corresponding elements in the other model are also deleted. Second, for everything that has been added (or inconsistently changed), new corresponding elements are created. In this case, the control engineer deleted the transition from the noConvoy to the convoyLeader state and added a new fading state fading_N2L and two transitions. Thus, the synchronization would also delete the corresponding transition from the system model and then add new elements, which is what we want to avoid.

In our previous work, we proposed an improved model synchronization approach [9]. The main idea is not to delete such corresponding parts right away, but to mark them for deletion first, so they can be reused later, i.e., in subsequent rule applications. Previous model synchronization approaches would simply create new corresponding parts when new elements are transformed. Our improved synchronization tries to reuse elements marked for deletion instead: it performs a search in the set of elements marked for deletion and tries to reuse fitting elements; if they fit, they are not deleted. Only if no fitting elements that are marked for deletion can be found, new elements are created. Finally, elements marked for deletion that could not be reused are actually destroyed. For details of the improved synchronization algorithm, please refer to Greenyer et al. [9].

In this case, this improved algorithm works as follows. First, as the transition from the noConvoy to the convoyLeader state has been deleted from the Stateflow model, the corresponding transition in the system model is marked for deletion. Next, we try to apply new rules. Here, the new, refined TGG rule (Fig.7) is applicable: It matches the transition in the system model that has been marked for deletion, and it also matches the new fading state and the new transitions in the Stateflow model. We can now apply this rule: In the CONSENS model, we reuse the transition marked for deletion, and we bind the elements of the refinement in the Stateflow model. As result, we have applied the refined TGG rule in backward direction without performing any changes to the CONSENS model just by reusing elements marked for deletion. The models are now consistent in terms of the TGG. This is exactly what we wanted to achieve: We have derived a new TGG rule which covers the refinement case described by the refinement rule. Furthermore, when later changes make the refinement invalid, e.g., when the time constraint is violated, the model transformation engine can detect this by checking the validity of the application of the refined TGG rule.

Note that we have to add precedences to the TGG rules. When propagating changes to the abstract model, we want to use these refined rules primarily, as applying the original, non-refined TGG rules would propagate the refinement

to the abstract model, which we wanted to avoid. When propagating to the concrete model, we do *not* want to use the new rules, as we need a functional, deterministic transformation. Thus, we only use the initial rule set.

Let us have a look at the next steps in the development process. As explained before, the software engineers work on their model, too. They change the behavior of the software by adding the possibility to reject a convoy proposal (③ in Fig. 3). This is a discipline-spanning relevant change, as it also affects, for instance, the controller implementation in the control engineering. Thus, it is propagated back to the system model (④): The state diagram is extended by two states waitForResponse and receivedConvoyProposal and new transitions and messages (see v1.1 of the system model in Fig. 3). Instead of switching to the state convoyFollower directly after a createConvoy message is send, the follower RailCab switches to the new state waitForResponse. There, it waits for the leader RailCab to accept or to reject the convoy proposal. The leader RailCab receives the createConvoy message and changes to the new state receivedConvoyProposal, in which it decides whether it accepts or rejects the proposal. If the convoy proposal is accepted, the leader RailCab changes its state to convoyLeader and the follower RailCab changes to the state convoyFollower. If the proposal is rejected, both RailCabs return to the state noConvoy.

These changes then must be propagated to other affected disciplines. Thus, the control engineering model also has to be updated to reflect the changed communication behavior (⑤). In the example, the createConvoy/ transition was changed in the system model during the synchronization in ④. To transform this change, a naïve synchronization would first revoke the respective rule application by deleting the corresponding elements in the Stateflow model, and then try to retransform the affected elements. However, this createConvoy/ transition in the system model corresponds to a refinement introduced in $v1.1_{CE}$ (the combination of the transition createConvoy, the state fading_N2L and the transition to the convoyLeader state in the control engineering model, which are bound by the refined TGG rule). Revoking the rule would destroy the complete refinement (see Fig. 8 b)).

As such an information loss must be prevented, we again use our improved synchronization. First, we revoke rules by marking for deletion. For instance, the fading_N2L state and its incoming and outgoing transition are marked for deletion due to the revocation of the refined TGG rule. Next, we transform the new elements in the system model to the control engineering model.

In general, there may be several possibilities to reuse elements previously marked for deletion, which leads to differently updated models; all of them are consistent according to the consistency relation. The synchronization may not be able to decide automatically which is the most reasonable. In our example, the question is where the newly added states waitForResponse and receivedConvoyRequest should be added: before (Fig. 8 c)) or after the fading states (Fig. 8 d))? Of course, an expert can quickly see that d) is the correct way of updating, as the controller strategy must not be switched before every RailCab has actually

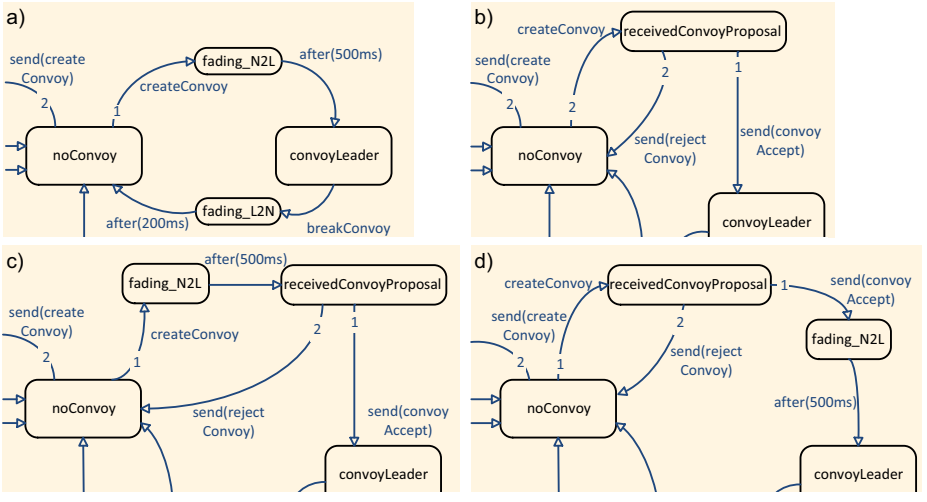


Fig. 8. Excerpts from Stateflow model: a) before updating; updated in different ways: b) lost fading state, c) "wrong" propagation of the change, d) correctly updated

approved the formation of a convoy. An automatic synchronization, however, cannot decide this.

Thus, our improved synchronization algorithm explicitly computes all reuse possibilities, rates them with respect to information loss, and asks the user in ambiguous cases which of the update possibilities is the correct one [9]. In the example, the refinement in the control engineering model that has been marked for deletion (consisting of the transition createConvoy, the state fading_N2L and the transition to the convoyLeader state) may be reusable as the corresponding control engineering part for three new transitions in the system model v1.1 (createConvoy/, /rejectConvoy, and /convoyAccept). However, the deleted refinement is not reusable as is. Some additional modifications have to be made to make it reusable in a certain case. For instance, when reusing elements marked for deletion as corresponding part for the new transition createConvoy/ (which would result in Fig. 8 c)), the target of the outgoing transition must be modified to point to the state receivedConvoyProposal.

We can sort the different update possibilities by the amount of modifications that must be made to reuse the elements: the less modifications must be made, the more likely is that this is a reasonable reuse possibility. In the example, we can reuse the refinement for the transition createConvoy/ (see Fig. 8 c)), as the source of the transition (the noConvoy state) is the same as before, but we must alter the target state. We can also reuse the refinement for the transition /convoyAccept (see Fig. 8 d)), as the target of the transition is the same (the convoyLeader state). It is, however, unreasonable to reuse the refinement for the transition /convoyReject, as neither the source nor the target state is the same as before. Thus, the expert can now decide between two reasonable reuse possibilities that are depicted in Fig. 8 c) and d).

6 Related Work

In MDA, a platform description model (PDM) is used to define the model transformation from an abstract PIM to a concrete PSM. In general, one PIM element may be mapped to more than one corresponding representation in the PSM. So-called mark models annotate the PIM and tell the model transformation which mapping to use. In contrast, our solution provides a way to capture alternative implementations using refinement rules in terms of the PSM. These alternatives are then automatically integrated into the model transformation. However, right now, we only use these refinements rules to derive model transformation rules which are not used in initial model transformations, but only match manually performed refinements. Therefore, we plan on extending our approach so that the refinement rules can be used for a) automatically deriving a mark meta-model and using a mark model to trigger alternative mappings (refinements), and b) to actively propose such possible refinements to the user.

If a refined TGG rule is applicable, its original rule will also be applicable. Here, we solve this by adding precedences. However, when having several model transformation rules applicable at the same elements in the source model, we have to determine which one to execute in general. Thus, we need to identify these conflicting rules. Therefore, we plan to include approaches like the critical-pair analysis, as described by Hermann et al. [13].

However, most solutions which implement an MDA-like process are limited to the concrete application scenario and/or models and tools. Only few publications deal with developing general solutions, e.g., to improve the usage of mark models, and only few model transformation solutions exist which deal with similar aspects as described in this paper. Körtgen [14] developed a synchronization tool for the case of a simultaneous evolution of both models. Although it does not incorporate a concept to define refinement operations, it also allows having several conflicting rules, i.e., rules that are all applicable at the same position. In a step-by-step, highly interactive process, the user may decide which alternatives should be applied. Our aim is to avoid unnecessary user interaction where that is possible. For ambiguous cases, however, we would also like to incorporate better means for user interaction into our synchronization tool.

A different approach in general is to create a single *unified modeling notation*, which includes all modeling means from all disciplines. This avoids defining model-to-model mappings, but requires a) view definitions and b) complex static semantics to ensure "internal" model consistency. Thus, the general model consistency problem remains. Several publications deal with inconsistency handling in a unified notation, e.g., Egyed et al. [4] or Atkinson et al. [2]. Although there is no fundamental conceptual drawback in using such a unified notation, our approach seems more reasonable from a practical and technical perspective in our setting, because most tools use their own model formats, anyway. Furthermore, such a unified notation is difficult to maintain. However, a more extensive discussion about this is outside the scope of this paper.

7 Conclusion and Discussion

In many model-based development processes, vertical model transformations map between models of different abstraction levels. If changes occur to a concrete model, these changes may be either relevant changes that affect the abstract model or model-specific refinements. Existing model synchronization approaches do not provide sufficient means to specify such refinement relations. In this paper, we have shown how in-place model transformation rules can be used to define refinements. We use this set of rules as input to a TGG-based synchronization approach, so that refinements can be automatically detected and will not be propagated. We have shown how this technique helps ensuring model consistency in a synchronization scenario in systems engineering.

At the moment, the refinement rules have to be defined manually, which can be a significant effort. Solutions exist which are able to identify and generalize actual model changes, e.g. Brosch et al. [3]. We plan on incorporating such approaches to (semi-)automatically derive refinement rules from examples.

We evaluated the approach using different disciplines' models from the RailCab research project. We identified and successfully modeled several refinements with our approach, e.g., redundancy patterns in structural models like UML component diagrams or block-based diagrams like Matlab/Simulink, and refined behaviors in behavioral models like statecharts or activity diagrams. However, we have yet to further evaluate the suitability and the efficiency of the approach in other domains, e.g., by performing industrial case studies.

Our approach only works if a refinement rule affects only one TGG rule. If this is not the case, more sophisticated computations are necessary to derive the refined TGG rules; this is subject to future research. As this occurs especially in complex TGGs that map between models that are unsimilar in its modeling principles, our technique will currently meet its limits in such a case.

There may be more than one reasonable solution to prevent the loss of existing refinements. In such ambiguous cases, user interaction is required. As future work, we plan on investigating how this user interaction can become more intuitive and easy to use. Furthermore, by storing and analyzing previous user decisions, we may be able to further improve the automation in ambiguous cases.

References

1. Amelunxen, C., Schürr, A.: Formalising model transformation rules for UML/MOF 2. IET Software 2(3), 204–222 (2008)
2. Atkinson, C., Stoll, D., Bostan, P.: Orthographic Software Modeling: A Practical Approach to View-Based Development. In: Maciaszek, L.A., González-Pérez, C., Jablonski, S. (eds.) ENASE 2008/2009. CCIS, vol. 69, pp. 206–219. Springer, Heidelberg (2010)
3. Brosch, P., Langer, P., Seidl, M., Wieland, K., Wimmer, M., Kappel, G., Retschitzegger, W., Schwinger, W.: An Example Is Worth a Thousand Words: Composite Operation Modeling By-Example. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 271–285. Springer, Heidelberg (2009)

4. Egyed, A., Letier, E., Finkelstein, A.: Generating and evaluating choices for fixing inconsistencies in UML design models. In: ASE 2008, pp. 99–108 (2008)
5. Gausemeier, J., Frank, U., Donoth, J., Kahl, S.: Specification technique for the description of self-optimizing mechatronic systems. Research in Engineering Design 20(4), 201–223 (2009)
6. Gausemeier, J., Schäfer, W., Greenyer, J., Kahl, S., Pook, S., Rieke, J.: Management of cross-domain model consistency during the development of advanced mechatronic systems. In: Proc. of the 17th Int. Conf. on Engineering Design (2009)
7. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Software and Systems Modeling 8(1) (2009)
8. Greenyer, J., Kindler, E.: Comparing relational model transformation technologies: Implementing Query/View/Transformation with Triple Graph Grammars. Software and Systems Modeling 9(1), 21–46 (2010)
9. Greenyer, J., Pook, S., Rieke, J.: Preventing Information Loss in Incremental Model Synchronization by Reusing Elements. In: France, R.B., Kuester, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 144–159. Springer, Heidelberg (2011)
10. Greenyer, J., Rieke, J., Schäfer, W., Sudmann, O.: The Mechatronic UML development process. In: Tarr, P.L., Wolf, A.L. (eds.) Engineering of Software, pp. 311–322. Springer, Heidelberg (2011)
11. Hearnden, D., Lawley, M., Raymond, K.: Incremental Model Transformation for the Evolution of Model-Driven Systems. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 321–335. Springer, Heidelberg (2006)
12. Henke, C., Tichy, M., Schneider, T., Böcker, J., Schäfer, W.: Organization and control of autonomous railway convoys. In: Proc. of the 9th Int. Symposium on Advanced Vehicle Control (2008)
13. Hermann, F., Ehrig, H., Orejas, F., Golas, U.: Formal Analysis of Functional Behaviour for Model Transformations Based on Triple Graph Grammars. In: Ehrig, H., Rensink, A., Rozenberg, G., Schürr, A. (eds.) ICGT 2010. LNCS, vol. 6372, pp. 155–170. Springer, Heidelberg (2010)
14. Körtgen, A.T.: Modellierung und Realisierung von Konsistenzsicherungswerkzeugen für simultane Dokumentenentwicklung. Ph.D. thesis, RWTH Aachen University (2009)
15. Mens, T., Van Gorp, P.: A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science 152, 125–142 (2006)
16. Object Management Group (OMG): MOF Query/View/Transformation (QVT) 1.0 Specification (2008), http://www.omg.org/spec/QVT/1.0/
17. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)

Model-Based Automated and Guided Configuration of Embedded Software Systems

Razieh Behjati[1,2], Shiva Nejati[1], Tao Yue[1],
Arnaud Gotlieb[1], and Lionel Briand[1,3]

[1] Simula Research Laboratory, Lysaker, Norway
[2] University of Oslo, Oslo, Norway
[3] University of Luxembourg
{raziehb,shiva,tao,arnaud,briand}@simula.no

Abstract. Configuring Integrated Control Systems (ICSs) is largely manual, time-consuming and error-prone. In this paper, we propose a model-based configuration approach that interactively guides engineers to configure software embedded in ICSs. Our approach verifies engineers' decisions at each configuration iteration, and further, automates some of the decisions. We use a constraint solver, SICStus Prolog, to automatically infer configuration decisions and to ensure the consistency of configuration data. We evaluated our approach by applying it to a real subsea oil production system. Specifically, we rebuilt a number of existing verified product configurations of our industry partner. 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.

Keywords: Product configuration, Model-based software engineering, Constraint satisfaction, UML/OCL.

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 follow a product-line engineering approach to develop the software embedded in their systems. They typically build a generic software that needs to be configured for each product according to the product's hardware architecture [5]. For example, in the oil and gas domain, embedded software needs to be configured for various field layouts (e.g., from single satellite wells to large multiple sites), for individual devices' properties (e.g., specific sensor resolution and scale levels), and for communication protocols with hardware devices.

Software configuration in ICSs is complicated by a number of factors. Embedded software systems in ICSs have typically very large configuration spaces, and their configuration requires precise knowledge about hardware design and specification. The engineers have to manually assign values to tens of thousands of configurable parameters, while accounting for constraints and dependencies

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 226–243, 2012.

between the parameters. This results in many configuration errors. Finally, the hardware and software configuration processes are often isolated from one another. Hence, many configuration errors are detected very late and only after the integration of software and hardware.

Software configuration has been previously studied in the area of software product lines [20], where support for configuration largely concentrates on resolving high-level variabilities in feature models and their extensions [18,13,11], e.g., the variabilities specified for end-users at the requirements-level. Feature models, however, are not easily amenable to capturing all kinds of variabilities and hardware-software dependencies in embedded systems. Furthermore, existing configuration approaches either do not particularly focus on interactively guiding engineers or verifying partial configurations [19,6], or their notion of configuration and their underlying mechanism are different from ours, and hence, not directly applicable to our problem domain [16,14].

Contributions. We propose a model-based approach that helps engineers create consistent and error-free software configurations for ICSs. In our work, a large amount of the data characterizing a software configuration for a particular product is already implied by the hardware architecture of that product. Our goal is, then, to help engineers assign this data to appropriate configurable parameters while maintaining the consistency of the configuration, and reducing the potential for human errors. Specifically, our approach (1) interactively guides engineers to make configuration decisions and automates some of the decisions, and (2) iteratively verifies software and hardware configuration consistency. We evaluated our approach by applying it to a subsea oil production system. Our experiments show that our approach can provide certain types of user guidance in an efficient manner, and can automate up to 50% of configuration decisions for the subjects in our experiment, therefore helping save significant configuration effort and avoid configuration errors.

In Section 2 we motivate the work and formulate the problem by explaining the current practice in configuring ICSs. We give an overview of our model-based solution in Section 3. SimPL methodology [5] for modeling families of ICSs is briefly presented in Section 4. We present our model-based approach to the abovementioned configuration problems in Section 5. An implementation of our approach as a prototype tool is presented in Section 6. An evaluation of the approach using our prototype tool is given in Section 7. In Section 8, we analyze the related work. Finally we conclude the paper in Section 9.

2 Configuration of ICSs: 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 subsea

Xmas tree in a subsea oil production system provides mechanical, electrical, and software components for controlling and monitoring a subsea well.

Product configuration is an essential activity in ICS development. It involves configuration of both software and hardware components. Currently, software and hardware configuration is performed separately in two different departments within our industry partner. 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.

The software configuration is done 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 whose values differ from one product to another. The latter group, known as *configurable components*, may need to be, further, decomposed and configured. The configuration stops once the type and the number of all the components and the values of their configurable parameters are given.

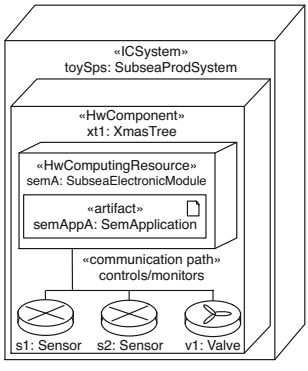


Fig. 1. A fragment of a simplified subsea production system

For example, software configuration for a family of subsea production systems starts by identifying the number and locations of SemApplication instances. Each instance is then configured according to the number, type, and other details of devices that it controls and monitors. To do this, the configuration engineer (the person who does the configuration) is typically provided with a hardware configuration plan. However, she has to manually check if the resulting software configuration conforms to the given hardware plan, and that it respects all the software consistency rules as well. In the presence of large numbers of interdependent configurable parameters this can become tedious and error-prone. In particular, due to lack of instant configuration checking, human errors such as incorrectly entered configuration data are usually discovered very late in the development life-cycle, making localizing and fixing such errors unnecessarily costly.

In short, the existing configuration support at our industry partner faces the following challenges (which seem to be generalizable to many other ICSs [5]): (1) Checking the consistency between hardware and software configurations is not automated. (2) Verification of partially-specified configurations to enable instant configuration checking is not supported. (3) Engineers are not provided with sufficient interactive guidance throughout the configuration process. In our previous work [5], we proposed a modeling methodology to properly capture and document, among other things, the software-hardware dependencies and consistency rules. In this paper, we build on our previous work to develop an automated guided configuration tool that addresses all the above-mentioned challenges.

3 Overview of Our Approach

Figure 2 shows an overview of our automated model-based configuration approach. In the first step, we build a configurable and generic model for an ICS family (the Product-line modeling step). In the second step, the Guided configuration step, we interactively guide users to generate the specification of particular products complying with the generic model built in the first step.

During the product-line modeling step, we provide domain experts with a UML/MARTE-based methodology, called SimPL [5], to manually create a product-line model describing an ICS family. The SimPL methodology enables engineers to create product line models from textual specifications and the scattered domain experts knowledge. These models can then be utilized to automate the configuration process. They include both software and hardware aspects as well as the dependencies among them. The dependencies are critical to effective configuration. Currently, most of these dependencies exist as tacit knowledge shared by a small number of domain experts, and only a fraction of them, mostly those related to software, have been implemented in the existing tool used by our industrial partner. Our domain analysis [5], however, showed that failure to capturing all the dependencies have led to critical configuration errors. We briefly describe and illustrate the SimPL methodology in Section 4.

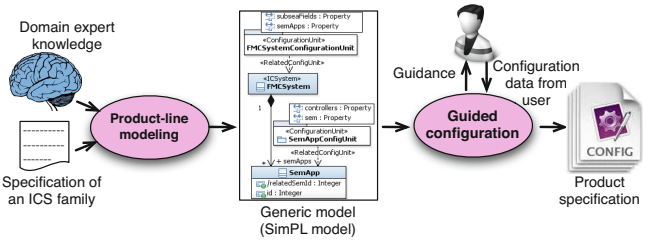


Fig. 2. An overview of our configuration approach

During the configuration step, engineers create full or partial product specifications by resolving variabilities in a product-line model. In our work, configuration is carried out iteratively, allowing engineers to create and validate partial product specifications, and interactively, guiding engineers to make decisions at each iteration. Therefore, our approach alleviates two shortcomings of the existing tool discussed in Section 2. Our configuration mechanism enables engineers to resolve variabilities in such a way that all the constraints and dependencies are preserved. At each iteration, the engineer resolves some of the variabilities by assigning values to selected configurable parameters. Our configuration engine, which is implemented using a constraint solver, automatically evaluates the engineer's decisions and informs her about the impacts of her decision on the yet-to-be-resolved variabilities, hence, guiding her to proceed with another round of

configuration. In Sections 5 and 6, we describe in details how the configuration step is designed and implemented, respectively.

4 Product-Line Modeling

In the first step of our approach in Figure 2, we use the SimPL modeling methodology [5] to create a generic model of an ICS family. The SimPL methodology enables engineers to create architecture models of ICS families that encompass, among other things, information about variability points in ICS families.

The SimPL methodology organizes a product-line model into two main views: the *system design view*, and the *variability view*. The system design view presents both hardware and software entities of the system and their relationships using the UML class diagram notation [1]. Classes, in this view, represent hardware or software entities distinguished by MARTE stereotypes [2]. The dependencies and constraints not expressible in class diagrams are captured by OCL constraints [3]. The variability view, on the other hand, captures the set of system variabilities using a collection of *template* packages. Each template package represents a *configuration unit* and is related to exactly one class in the system design view. Template parameters of each template package in the variability view are related to the configurable properties of the class related to that package. Template packages and template parameters are inherent features in UML and are intended to be used for the specification of generic structures. In the reminder of this section, we first describe a small fragment of a subsea product-line model, which is used as our running example. Then, using our running example, we provide a model-based view on the essential configuration activities mentioned in Section 2.

4.1 A Subsea Product-Line Model

Figure 3 shows a fragment of the SimPL model for a 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. The electronic boards are categorized into four different types based on their number of pins. 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 we discussed above.

[1] This is a sanitized fragment of a subsea production case study. For a complete model, see [5].

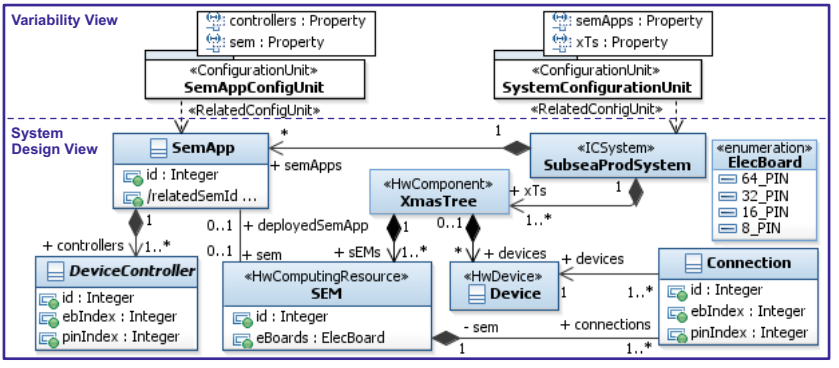


Fig. 3. A fragment of the SimPL model for the subsea production system

The variability view in the SimPL methodology is a collection of template packages. 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. Template parameters of this package specify the configuration parameters of the subsea production system, which are: the number of XmasTrees, and SEM applications (semApps). Some of the other configurable parameters in Figure 3 are: the number and type of device controllers in a SemAPP as shown in SemAppConfigUnit using the template parameter controllers, the number of SEMs and devices in a XmasTree, etc.

As mentioned earlier, the SimPL model may include OCL constraints as well. Two example OCL constraints related to the model in Figure 3 are given below.

```
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.

4.2 Configuration Activities in a Model-Based Context

As mentioned in Section 2, configuration involves a sequence of two basic activities: (1) specifying the type and the number of (sub)components, and (2)

determining the values for the configurable parameters of each component, while satisfying the constraints and dependencies between the parameters. We ground our configuration approach on the SimPL methodology and redefine the notion of configuration in modelling terms as follows: Given a SimPL model, configuration is creating an instance model (i.e., product specification in Figure 2) conforming to the classes, the dependencies between classes, and the OCL constraints specified in that SimPL model. Such instance model is built via two activities (1) creating instances for classes that correspond to configurable components, and (2) assigning values to the configurable parameters of those instances. For example, to configure the subsea system in Figure 3, we need to first create instances of XmasTree, SEM, Device, and SemApp, and then assign appropriate values to the configurable variables of these instances. Note that value assignment may imply instance creation as well. Specifically, a configurable parameter can represent the cardinality of an association. Assigning a value to such a parameter automatically implies creation of a number of instances to reach the specified cardinality.

5 Interactive Model-Based Guided Configuration

The outcome of the configuration step in Figure 2 is a (possibly partial) model of a product that is *consistent* with the SimPL model describing the product family to which that product belongs. In our approach, SimPL models are described using class-based models, while the product models are object-based. A product model is consistent with its related SimPL model when:

- Each object in the product model is an instance of a class in the SimPL model.
- Two objects of types C_1 and C_2 are connected only if there is an association between classes C_1 and C_2 in the SimPL model.
- The object model satisfies the OCL constraints of the SimPL model.

The above *consistency rules* are invariant throughout our configuration process, i.e., they hold at each configuration iteration even when the product model is defined partially. In this section, we first describe how our approach guides the user at each configuration iteration while ensuring that the above rules are not violated. We then demonstrate how a constraint solver can be used to maintain the consistency rules throughout the entire configuration process, and to automatically perform some of the configuration iterations.

5.1 Guided and Automated Configuration

The product configuration process is a sequence of *value-assignment* steps. At each step, a value is assigned to one *configurable parameter*. A configurable parameter can represent (1) a property in an instance of a class, (2) the size of a collection of objects in an instance of a class, or (3) the concrete type of an instance.

A *configuration* is a collection of value-assignments, from which a full or partial product model can be generated. A configuration is complete when all the configurable parameters are assigned a specific value, and is partial otherwise. Each configurable parameter has a *valid domain* that identifies the set of all values that can be assigned to that configurable parameter without violating any consistency rule. Below, we describe the guidance information that our tool provides to the user at each iteration of the configuration process.

Valid domains. At each iteration, the tool provides the user with the valid domains for all the configurable parameters. Such domains are dynamically recomputed given previous iterations. The values that the user provides should be within these valid domains, or otherwise, the user's decision is rejected and he receives an error message. For example, the valid domain for the configurable parameter pinIndex is initially 0..63. Therefore, if a user assigns to this parameter a value outside 0..63 his decision will be rejected.

Decision impacts. If the user's decision is correct, the decision is propagated through the configuration to identify its impacts on the valid domains of other configurable parameters. This may result in pruning some values from the valid domains of some configurable parameters. For example, the valid domain for the type of an eBoard in a SEM is initially {8_PIN, 16_PIN, 32_PIN, 64_PIN} (the set of all literals in the enumeration ElecBoard). If a user configures a Connection in a SEM by assigning 2 to ebIndex, and 13 to pinIndex, then according to the OCL invariant PinRange (defined above), the third eBoard in that SEM must at least have 14 pins. Therefore, such a value-assignment removes 8_PIN from the valid domain of the type of the third eBoard, resulting in the pruned valid domain {16_PIN, 32_PIN, 64_PIN}.

The impacts of the decisions are then reported to the user, in terms of reduced valid domains.

Value inference. After value-assignment propagation and pruning, the tool checks if the size of any valid domains is reduced to one. The configurable parameters with singleton valid domains are set to their only possible value. This enables automatic inferences of values for some configurable parameters, therefore, saving a number of value-assignment steps from the user. For example, in Figure 3 there is a one-to-one deployment relationship between SEM and SemApp. As a result, whenever the user creates a new instance of SEM the tool automatically creates a new instance of SemApp and correctly configures in it the cross-reference to the SEM. Inferring a value for a configurable parameter that represents the size of an object collection, is followed by automatically creating and adding to that collection the required number of objects.

5.2 Constraint Satisfaction to Provide Guidance and Automation

The main computation required for providing the aforementioned guidance and automation is the calculation of valid domains through pruning the domains of

all the yet-to-be-configured parameters after each configuration iteration using the user's configuration decision.

In our approach, we use a constraint solver over finite domains to calculate the valid domains. In this approach, the configuration space of a product family forms a *constraint system* composed of a set of variables, $x_1, ..., x_n$, and a set of constraints, $\mathcal{C}$, over those variables. Variables represent the configurable parameters, and get their values from the *finite domains* $\mathcal{D}_1, ..., \mathcal{D}_n$. A finite domain is a finite collection of tags, that can be mapped to unique integers. We extract the finite domains of variables from the types of the configurable parameters, enumerations, multiplicities, and OCL constraints in the SimPL model. The constraint set $\mathcal{C}$ includes both the OCL constraints and the information, e.g., multiplicities, extracted from the class diagrams in the SimPL model. A configuration in this scheme corresponds to a (possibly partial) evaluation of the variables $x_1, ..., x_n$. Using a constraint solver the consistency of a configuration w.r.t the constraint set $\mathcal{C}$ is checked, and the valid domains, $D^*{}_1, ..., D^*{}_n$, for all the variables are calculated.

At each value-assignment step during the configuration, a value v_i is assigned to a variable x_i. This value assignment forms a new constraint $c : x_i = v_i$, which is added to the constraint set $\mathcal{C}$. The added constraint is then propagated throughout the constraint system to identify the impacts of the assigned value on other variables, and to prune and update the valid domains of those variables. This process is realized through a simple and efficient Constraint Programming technique called *constraint propagation* [15]. Constraint propagation is a monotonic and iterative process. During constraint propagation, constraints are used to filter the domains of variables by removing inconsistent values. The algorithm iterates until no more pruning is possible.

Assigning a value to a variable representing the size of a collection relates to adding items to, or removing items from the collection. Adding an item to a collection implies introducing new variables to the constraint system. Similarly, removing items from a collection implies removing variables from the constraint system. As a result, to identify the impacts of changing the size of a collection, new variables have to be added or removed during constraint propagation. This is possible as constraint propagation does not require the set of initial variables to be known a priori. However, the process is no longer monotonic in that case and may iterate forever. In our application, the number of added variables is always bounded, avoiding any non-termination problems.

In our approach, we allow users to modify the previously assigned values as long as the modification does not give rise to any conflict. Since we always keep valid domains of all the configurable parameters up-to-date, conflicts can be detected simply by checking whether the new value is still within the valid domain of the modified configurable parameter. In the following section, we further elaborate on the design of a tool implementing the configuration process presented above.

6 Prototype Tool

Figure 4 shows the architecture of the configuration engine that provides the guidance and automation mentioned in Section 5. Inputs to the engine are the generic model of the product family, and the user-provided configuration data. The configuration process starts by loading the generic model. From the loaded model, the configuration engine extracts the first set of the configurable parameters. These configurable parameters are presented to the user via the interactive user interface for collecting configuration decisions. In addition, the configuration engine generates a constraint model from the input model of the product family. This constraint model is implemented in `clpfd`, a library of the *SICStus Prolog* environment [8,4]. In `clpfd`, each configurable parameter is represented by a logic variable, to which is associated a finite set of possible values, called a finite domain. After the generic model is loaded, the configuration engineer starts an interactive configuration session for entering configuration decisions.

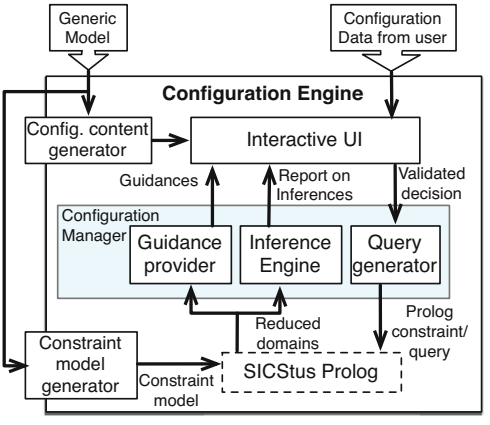


Fig. 4. Architecture of the configuration tool

The configuration engine iteratively and interactively collects configuration decisions from the user. At each iteration, the user enters the values for one or more configurable parameters. Using the domains of the configurable parameters, the consistency of the configuration decisions is checked. If the entered values are all consistent, the *Query generator* is invoked to create a new Prolog query representing a constraint system that contains all the constraints created from the collected configuration decisions. This Prolog query is then used to invoke constraint propagation in order to prune the domains. The new domains serve as inputs to the *Inference engine*, which implements the inference mechanism explained in Section 5.1 to infer values, and the *Guidance provider*, which reports the impacts of configuration choices (e.g., updated domains).

6.1 The `clpfd` Library of SICStus Prolog

Choosing Prolog as a host language for developing our configuration engine has several advantages. First, Prolog is a well-established declarative and high-level programming language, allowing fast prototyping for building a proof-of-concept tool, and containing all the necessary interfaces to widely-used programming languages such as Java or C++. In our tool development, we have used the `jasper` library that allows invoking the SICStus Prolog engine from a Java program. Second, as it embeds a finite domains constraint solver through the

clpfd library, this allows us to benefit from a very efficient implementation of constraint propagation [9], and all the available constructs (e.g., combinatorial constraints) that have been proposed for handling other applications.

6.2 Mapping to clpfd

To use the finite domains constraint engine of SICStus Prolog, we need to translate an ICS product specification into clpfd. This requires: (1) translating the SimPL model characterizing the ICS family, and (2) translating the instance model representing the product.

In the first translation, we create a Prolog/clpfd program capturing the UML classes, the relationships between the classes, and the OCL constraints of the SimPL model. Our approach for this translation is very similar to a generic UML/OCL to Prolog translation given by [7]. Briefly, we map UML classes and relationships to Prolog compound terms, and every OCL (sub)expression to a Prolog *rule* whose variables correspond to the variables of the given OCL (sub)expression.

In the second translation, given an instance model, we create a SICStus Prolog query to evaluate conformance of the instance model to its related SimPL model (consisting of classes, their relationships, and OCL constraints) captured as a Prolog program as discussed above. To build such query, we map each instance in the given instance model to a Prolog list, and map every configurable parameter of that instance to an element of that list. A configurable parameter that is not yet assigned to a value becomes a variable in the list. For example, a SICStus Prolog query related to an instance model looks like check_product(Als, Ids), where Als is the list representation of all instances, and Ids is the list of the identifiers of instances. The query generator in our tool is responsible for generating these two lists from the instances created and configured by the user. Given the query check_product(Als, Ids), the constraint engine checks whether the instance model specified by Als and Ids conforms to the input SimPL model, and if so, it provides the valid domains for all the variables in Als. Note that the calculation of the valid domains terminates because Als contains a finite number of variables (as the number of the instances in the product are finite), and all variables take their values from finite domains.

7 Evaluation

To empirically evaluate our approach, we performed several experiments which are reported in this section. The experiments are designed to answer the following three main research questions:

1. What percentage of the value-assignment steps can be saved using our automated configuration approach?
2. How much do the valid domains shrink at each iteration of configuration?
3. How long does it take to propagate a user's decision and provide guidance?

Saving a number of value-assignment steps is expected to reduce the configuration effort, and reduction of the domains decreases the complexity of decision making. Therefore, answers to the first two research questions provide insights on how much configuration effort can be saved. Answering the third research question provides insights into the applicability and scalability of our technique.

To answer these questions we designed an experiment in which we rebuilt three verified configurations from our industry partner using our configuration tool. One configuration belongs to the environmental stress screening (ESS) test of the SEM hardware, which we refer to in this section as the ESS Test. The other two are the verified configurations of two complete products, which we refer to in this section as Product_1 and Product_2. Table 1 summarizes the characteristics of these configurations. We performed our experiments using the simplified generic model of the subsea product family given in Section 4. Number of objects and variables in Table 1 are calculated w.r.t that simplified model.

Table 1. Characteristics of the rebuilt configurations

	# XmasTrees	# SEMs	# Devices	# Objects	# Variables
ESS Test	1	1	111	226	343
Product_1	9	18	453	1396	2830
Product_2	14	28	854	2619	5307

We report in Sections 7.1-7.3 the evaluation and analysis that we performed on the experiments to answer the above research questions. At the end of this section, we also discuss some limitations, directions for future work, and the generalizability of our approach.

7.1 Inference Percentage

The configuration effort required for creating the configuration of a product is expected to be proportional to the number of configuration iterations and the number of value-assignment steps. Automating the latter is therefore expected to save configuration effort and minimize chances for errors. To measure the effectiveness of our approach in reducing the number of value-assignment steps, we have defined an *inference rate* which is equal to the number of inferred decisions divided by the total number of decisions:

$$inference\ rate = \frac{inferences}{manual\_decisions + inferences} \quad (1)$$

Table 2 shows the inference rates in each case.

Table 2. Inference rates

	# Manual decisions	# Inferred decisions	Inference rate (%)
ESS Test	373	16	4.11
Product_1	1459	1426	49.42
Product_2	2802	2783	49.82

Note that the inference rate for Product_1 and Product_2 is very close to 50 %. This is because of the *structural symmetry* that exists in the architecture of the system. Structural symmetry is achieved in a product when two or more components of the system have identical or similar configurations. We have modeled the structural symmetries using two OCL constraints. One specifies that each XmasTree has two SEMs (*twin SEMs*) with identical configurations (i.e., identical number and types of electronic boards and devices connected to them). The other specifies that all the XmasTrees in the system have similar configurations (e.g., all have the same number and types of devices). The first OCL constraint applies to both Product_1 and Product_2, while the second applies to Product_2 only. As a result, the inference rate for Product_2 is slightly higher than that for Product_1. Neither of the OCL constraints applies to the ESS Test, which contains only one XmasTree and one SEM. Therefore, it shows a very low inference rate. In general, the architecture of the product family, and characteristics of the product itself (e.g., structural symmetry) can largely affect the inference rate.

Our experiment shows that our approach can automatically infer a large number of consistent configuration decisions specially for products with some degree of structural symmetry. Assuming automated value-assignments have similar complexity to manual ones, our approach can save about 50% of the configuration effort of Product_1 and Product_2.

7.2 Reduction of Valid Domains

Pruned domains are the output of constraint propagation. Pruning of the domains decreases the complexity of decisions to be made. As part of our experiment, we measured how the domains shrink after each constraint propagation step. Such reduction of the domains is measured by comparing the size of each pruned domain before and after constraint propagation. This is possible and meaningful because all the domains are finite. Table 3 shows the average reduction of domains for each case. *Reduction rate* in the table is defined as the proportion of the *reduction size* (i.e., number of distinct values removed from a domain) to the initial size of the domain (i.e., the number of distinct values in a domain). In the calculations in Table 3 we have not considered domain reductions that resulted in inferences. This result shows that the domains of variables can be considerably reduced when a value is assigned to a dependent variable. Specifically, it shows that, on average, after each value-assignment step 37.98% of the values of the dependent variables are invalidated. Without such a dynamic recomputation of valid domains, there would be a higher risk for the user to make inconsistent configuration decisions. Moreover, comparing the inference rate from Table 2 and the reduction rate from Table 3 over the three cases suggests that while structural symmetry can highly affect the inference rate, it does not have a large impact on the reduction rate.

Table 3. Average shrinking of the domains

	Count*	Avg. initial domain size	Avg. reduction size	Avg. reduction rate (%)
ESS Test	732	30.557	13.803	45.17
Product_1	2564	62.125	21.367	34.39
Product_2	7557	35.97	14.205	39.49
			Avg. over all cases:	**37.98**

* total number of domains that have been pruned or reduced.
Avg.: the average over all reduced domains in the whole configuration.

7.3 Constraint Propagation Efficiency

Providing automation and guidance as part of the interactive configuration process requires the underlying computation to be sufficiently efficient for our approach to be practical.

We define the efficiency of our approach as the amount of time needed for validating and propagating the user decision. For this purpose, we have measured at each constraint propagation step the execution time, and the number of variables in the constraint system. Figure 5 shows the average time required for propagating user decisions after each value-assignment step. As shown in this figure, for products with less than 1000 variables, it takes, on average, less than one second to validate and propagate the decision. However, this time grows polynomially with the number of variables, which itself is proportional to the number of instances.

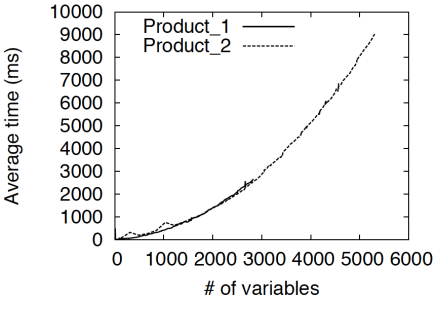


Fig. 5. Constraint propagation time grows quadratically with the number of variables (with a coefficient of determination of 0.9994)

Since in our experiment we have used a simplified model of the product family, we expect that for a complete model of the system the number of instances and the number of variables be much higher than that in this experiment. However, our experiment shows that not all of these variables are dependent on each other. To provide an insight into the level of dependency between variables, for each case, we can compute the average number of reduced domains. The average number of reduced domains is 1.8 (2564 from Table 3 divided by 1459 from Table 2) for Product_1 and, 2.7 for Product_2. In other words, on average, each variable in Product_1 (Product_2) is dependent to less than two (three) other variables. The polynomial ($O(n^2)$) growth of the execution time is, however, due to our current implementation, in which, we compute the valid domains of all variables (not only the dependent variables) by creating a new constraint propagation session after each value-assignment step. Therefore, we expect that by optimizing our implementation and incrementally adding new constraints to an existing constraint propagation session we can significantly improve the efficiency of our approach. Such an optimization requires an additional preprocessing step before

creating queries and invoking the constraint solver. This needs to be investigated in more depth and is left for future research.

7.4 Discussion

Limitations and directions for future work. The inference rate and the reduction rate, in addition to be affected by the architecture of the product family, are affected by the order in which the decisions are made. An *optimal order* of applying configuration decisions can be defined as the order which can result in the maximum inference rate and reduction rate. The optimal order can be reported to the user as additional guidance. Our current implementation does not provide such a guidance and therefore the results reported in this paper are probably, a lower bound for potential configuration effort savings. It is therefore important that in the future we support the optimization of the ordering to maximize inferred decisions and the reduction of domains. Devising criteria and heuristics for finding such optimal order is one direction of our future work.

Another research question is "How useful is the guidance provided by our approach?". Answering this question requires conducting an experiment involving human subjects. This experiment is also part of future work.

Generalizability of our approach. Like any other model-based engineering approach, the effectiveness of our approach depends on the quality of the input generic models. Our configuration approach can be used to configure only the variabilities that are captured in the generic model of the product family. Similarly, the approach can validate the decisions and automatically infer decisions only based on the dependencies that are captured in the model. Our evaluation in this paper shows that the SimPL methodology and notations that we proposed in [5] enables the creation of models of the required quality.

The use of a constraint solver over finite domains limits our approach to the constraints that capture restrictions on variables with finite domains. Constraint solvers over continuous domains are available to overcome this limitation but their integration with an efficient finite domains solver is still an open research problem [10]. Moreover, as we have not encountered this type of constraint with our industry partner, we don't expect this to be a restriction in our context.

8 Related Work

Most of the existing work on constraint-based automated configuration in product-line engineering focuses on resolving variabilities specified by feature models [17] and their extensions [12]. Basic feature models cannot express complex variabilities or dependencies required for configuring embedded systems [5]. However, extended feature models that allow attributes, cardinalities, references to other features, and cloning of features are, as mentioned in [11], as expressive as UML class diagrams and can be augmented by OCL or XPath queries to describe complicated feature relationships as well.

We compare our work with the existing automated configuration and verification tools proposed for extended feature models since these are the closest to our SimPL models. FMP [11] is an Eclipse plug-in that enables creation and configuration of extended feature models. FMP can verify full or partial configurations for a subset of extended feature models, specifically those with boolean variables and without clonable features. FAMA [6] drops this limitation and can verify extended feature models with variables over finite domains. However, FAMA is more targeted towards the verification and analysis of feature models. Therefore, it does not handle validating partial configurations or help build full configurations iteratively. Finally, Mazo et. al. [19] use constraint solvers over finite domains to analyze extended feature models. This approach is the closest to ours as it can handle all the advanced constructs in extended feature models, and further enables verification of full and partial configurations.

The main limitation of all of the above approaches is that none of them supports verification and analysis of complex constraints such as those in Section 4.1. These constraints express complex relationships between individual elements or collections of elements and are instrumental in describing software/hardware dependencies and consistency rules in embedded systems. Our tool, in addition to verifying these constraints, provides interactive guidance to help engineers effectively build configurations satisfying these constraints. Finally, to the best of our knowledge, none of the above approaches have been applied to nor evaluated on real case studies.

More recently, constraint satisfaction techniques have been used to automate configuration in the presence of design or resource constraints [16,14]. The main objective is to search through the configuration space in order to find optimized configurations satisfying certain constraints. Our work, however, focuses on interactively guiding engineers to build consistent product configurations, a problem that we have shown earlier in our paper to be important in practice. We do not intend to replace human decision making during configuration. Instead, we plan to support engineers when applying their decisions in order to reduce human errors and configuration effort.

In contrast to related work in [16,14], we enable users to interact with the constraint solver during the search. This is because supporting user guidance and interactive configuration are paramount to our approach. As a result, we require a technique that is fast enough for instant interaction with users and therefore cannot rely on dynamic constraint solving, which the authors in [16] have shown to be orders of magnitude slower than the SICStus CLP(FD) library. As for DesertFD in [14], it neither provides user guidance nor enables interactive configuration.

9 Conclusion

In this paper, we presented an automated model-based configuration approach for embedded software systems. Our approach builds on generic models created in our earlier work, i.e., the SimPL models, and uses constraint solvers to interactively guide engineers in building and verifying full or partial configurations.

We evaluated our approach by applying it to a real subsea production system where we rebuilt three verified configurations of this system to evaluate three important practical factors: (1) reducing configuration effort, (2) reducing possibility of human errors, and (3) scalability. Our evaluation showed that, in our three example configurations, our approach (1) can automatically infer up to 50% of the configuration decisions, (2) can reduce the size of the valid domains of the configurable parameters by 40%, and (3) can evaluate each configuration decision in less than 9 seconds.

While our preliminary evaluations demonstrate the effectiveness of our approach, the value of our tool is likely to depend on its scalability to very large and complex configurable systems. In particular, being an interactive tool, its usability and adoption will very much depend on how fast it can provide the guidance information at each iteration. Our current analysis shows that the propagation time grows polynomially with the size of the product. But we noticed in our work that after each iteration only a very small subset of variables are affected. Therefore, if we could reuse the analysis results from the previous iterations, we could possibly improve the time it takes to analyze each round significantly.

References

1. UML Superstructure Specification, v2.3 (May 2010)
2. Marte (2012), http://www.omgmarte.org/
3. Object Constraint Language (2012), http://www.omg.org/spec/OCL/2.2/
4. Sicstus Prolog Homepage (February 2012), http://www.sics.se/sicstus/
5. Behjati, R., Yue, T., Briand, L., Selic, B.: SimPL a product-line modeling methodology for families of integrated control systems, Tech. Repo 2011-14, SRL (2011), http://simula.no/publications/Simula.simula.746
6. Benavides, D., Segura, S., Trinidad, P., Ruiz Cortés, A.: Fama: Tooling a framework for the automated analysis of feature models. In: VaMoS (2007)
7. Cabot, J., Clarisó, R., Riera, D.: Verification of uml/ocl class diagrams using constraint programming, Washington, DC, USA, pp. 73–80 (2008)
8. Carlsson, M., Mildner, P.: Sicstus prolog – the first 25 years. CoRR, abs/1011.5640 (2010)
9. Carlsson, M., Ottosson, G., Carlson, B.: An Open-Ended Finite Domain Constraint Solver. In: Hartel, P.H., Kuchen, H. (eds.) PLILP 1997. LNCS, vol. 1292, pp. 191–206. Springer, Heidelberg (1997)
10. Collavizza, H., Rueher, M., Van Hentenryck, P.: A constraint-programming framework for bounded program verification. Constraints Journal (2010)
11. Czarnecki, K., Kim, P.: Cardinality-Based Feature Modeling and Constraints: A Progress Report. In: Proceedings of the International Workshop on Software Factories at OOPSLA (2005)
12. Czarnecki, K., Helsen, S., Eisenecker, U.: Formalizing cardinality-based feature models and their specialization. In: Software Process: Improvement and Practice (2005)
13. Czarnecki, K., Pietroszek, K.: Verifying feature-based model templates against well-formedness ocl constraints. In: GPCE 2006, pp. 211–220 (2006)

14. Eames, B.K., Neema, S., Saraswat, R.: Desertfd: a finite-domain constraint based tool for design space exploration. Design Autom. for Emb. Sys. 14(2), 43–74 (2010)
15. Van Hentenryck, P., Saraswat, V.A., Deville, Y.: Design, implementation, and evaluation of the constraint language cc(fd). Selected Papers from Constraint Programming: Basics and Trends (1995)
16. Horváth, Á., Varró, D.: Dynamic constraint satisfaction problems over models. Software and Systems Modeling (November 2010)
17. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Spencer Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21 (1990)
18. Lopez-Herrejon, R.E., Egyed, A.: Detecting Inconsistencies in Multi-View Models with Variability. In: Kühne, T., Selic, B., Gervais, M.-P., Terrier, F. (eds.) ECMFA 2010. LNCS, vol. 6138, pp. 217–232. Springer, Heidelberg (2010)
19. Mazo, R., Salinesi, C., Diaz, D., Lora-Michiels, A.: Transforming attribute and clone-enabled feature models into constraint programs over finite domains. In: ENASE (2011)
20. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc., Secaucus (2005)

Lightweight String Reasoning for OCL

Fabian Büttner* and Jordi Cabot

AtlanMod, École des Mines de Nantes - INRIA, Nantes, France
{fabian.buettner,jordi.cabot}@inria.fr

Abstract. Models play a key role in assuring software quality in the model-driven approach. Precise models usually require the definition of OCL expressions to specify model constraints that cannot be expressed graphically. Techniques that check the satisfiability of such models and find corresponding instances of them are important in various activities, such as model-based testing and validation. Several tools to check model satisfiability have been developed but to our knowledge, none of them yet supports the analysis of OCL expressions including operations on Strings in general terms. As, in contrast, many industrial models do contain such operations, there is evidently a gap.

There has been much research on formal reasoning on strings in general, but so far the results could not be included into model finding approaches. For model finding, string reasoning only contributes a sub-problem, therefore, a string reasoning approach for model finding should not add up front too much computational complexity to the global model finding problem. We present such a lightweight approach based on constraint satisfaction problems and constraint rewriting. Our approach efficiently solves several common kinds of string constraints and it is integrated into the EMFtoCSP model finder.

Keywords: OCL, String data type, Model Finding.

1 Introduction

Model-driven Engineering (MDE) is a popular approach to the development of software based on the use of models as primary artifacts. To precisely describe the conceptual structure of a model, the Object Constraint Language (OCL) [22] has been widely accepted as a de-facto standard. In a nutshell, OCL allows to express model constraints using a first-order logic like language for objects.

Naturally, the increased precision comes along with an increased complexity of the models. This raises the need for systematic approaches to model validation, model verification, and model-based testing. Model finding (also called model instantiation) is an important problem in this context. It considers the question if a given model (including constraints) is satisfiable, and if it is satisfiable, to identify one instance of the model. While in model verification, model finders are typically used to show unsatisfiability when reasoning about implications between different constraints, the focus in model-based testing is typically on finding satisfying instances, which can be used to test a system which is based on the model.

* This work has been partly funded by the European Project CESAR.

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 244–258, 2012.

The community has developed several approaches and tools for automated model finding for OCL-annotated models. To deal with the computational complexity of the problem (which is undecidable in general), most of them are based on some underlying formalism for which sophisticated decision procedures and tools exist, such as first-order logic and satisfiability modulo theory (SMT), relational logic, propositional logic, and constraint satisfaction problems (CSP).

While these approaches cover an extensive subset of OCL, to our knowledge none of them supports the String data type and its OCL operations. The primitive data types typically supported are Integer and Boolean. Given that, on the contrary, several 'real life' models actually do contain constraints over strings, there is evidently a gap that needs to be addressed. However, we need to be aware that, when compared to models that only contain Integer primitive values, adding strings to the subject of reasoning introduces another level of complexity.

There are several works that address string reasoning, some focused on checking grammar satisfiability as a stand-alone problem, others on path analysis for string-manipulating programs. However, in the context of model finding and, in particular, model-based testing, string reasoning only contributes a sub-problem to the overall search problem. Therefore, there is a trade-off between the completeness of the string reasoning procedures and the overall model finding performance.

In this paper we present a lightweight approach to integrate constraints over bounded strings into model finding using constraint rewriting and Boolean and integer constraints. It is a two-step approach that first reasons about the lengths of the strings, then infers constraints on the individual elements of the strings (their character variables). In general, our approach can be implemented in any off-the-shelf solver that supports reasoning about linear constraints. We included it in the OCL model finder EMFtoCSP [14], the successor of UMLtoCSP [7], which is based on the ECL^{i}PS^{e} constraint logic programming environment.

For many common constraint constellations, our approach is scalable and shows good performance, and we claim that it is suited for several practical applications that do not pose hard, non-tractable string constraints. We provide experimental results that show that models with more than a thousand strings can be found within seconds.

Paper Organization. In Sect. 2, we first discuss the state of the art for the topic. Section 3 then formally presents our approach. In Sect. 4 we discuss its limits and scalability, and present experimental results of our implementation of the approach in the tool EMFtoCSP. Section 5 concludes our contribution and identifies future work.

2 State of the Art

In this section, we describe the state of the art in model finding for OCL-annotated models in general and its translation into constraint satisfaction problems, and we put our work in the context of general formal reasoning techniques for strings.

2.1 Model Finding

The model finding (or model instantiation) problem for models with constraints can be defined as follows: Let $\mathcal{M}$ be a model defining structural elements such as classes,

associations, and attributes. Let $C_{\mathcal{M}}$ be a set of constraints over $\mathcal{M}$. Let $I(\mathcal{M})$ denote the (possibly infinite) set of models (instances) of $\mathcal{M}$. The pair $(\mathcal{M}, C_{\mathcal{M}})$ is called satisfiable iff there exists a least one instance $\sigma \in I(\mathcal{M})$ such that $\sigma \models \hat{c}_i$ holds for each $c_i \in C_{\mathcal{M}}$, where we assume $\hat{c}_i$ to be a logical representation of c_i that can be evaluated on σ. A model that is not satisfiable is called unsatisfiable. If a model is satisfiable, one is typically interested in a satisfying instance of it, too.

Model finding is important in several tasks within the model-driven approach. It is required in the validation and testing of systems based on the model (to systematically specify test cases), validation and testing of model transformations [3,23], as well in the validation and verification of the model itself. Model finding has also been applied to verify the correctness of model transformations as transformation models (e.g., [6,5]).

The community has developed several approaches and tools for automated model finding for models with OCL constraints. To deal with the computational complexity of the problem (which is undecidable for OCL in general), most of them are based on some underlying formalism for which sophisticated decision procedures and tools exists, such as, first-order logic and SMT [9], relational logic [2,19], propositional logic [25], genetic algorithms [1], graph grammars [27] and constraint satisfaction problems [7]. All of these approaches support a more or less extensive subset of OCL (e.g., including quantifiers and collections), but, to our knowledge, the support of the String data type is very limited. [12] supports strings, but requires the user to specify an explicit procedure for the construction of potential strings and is not a black box model finding approach. [1] considers strings, but the approach, which is based on genetic algorithms, is focused on test case generation only and is (intentionally) not exhaustive. A promising approach is our view is [18], where the authors propose an encoding of OCL strings for the relational logic solver Kodkod. However, to our knowledge, this encoding still requires a constant maximum number of boolean variables, equal for all string variables (even when this length is only required for a single string).

We want to emphasize that most of the underlying formalism and solvers employed in the aforementioned approaches do support bounded bitvectors or sequences. Thus, theoretically, strings can be translated straightforwardly into these formalisms. However, unless considerably small upper bounds are imposed on string lengths, this quickly leads to extreme search spaces, with considerable effects on the runtime of the solvers and the decidability of the search problem in practice. Furthermore, this also prevents them from taking into account the String semantics on the symbolic level to reduce the search space up front.

2.2 Model Finding as a CSP

A constraint satisfaction problem (CSP) can be defined by a tuple

$$(V, C)$$

where $V = \{v_1 \in D_1, \ldots, v_m \in D_m\}$ is a set of variables v_i and their domains D_i, and C is a set of constraints over V. Where clear from the context, we will omit variable

domains in the following. A constraint has the form $P(x_1, \ldots, x_n)$ where P denotes an n-ary predicate on $x_1, \ldots, x_n \in V$. An assignment β of values to variables *satisfies* a constraint $P(x_1, \ldots, x_n)$ if $P\big(\beta(x_1), \ldots, \beta(x_n)\big)$ is true. If β satisfies all constraints in C, it is a *solution* to P. We say a CSP is *consistent* (or feasible) if it has a solution, and *inconsistent* (or infeasible) if it does not.

Typical constraints employed in CSPs include a combination of arithmetic expressions, mathematical comparison operators and logical operators. Common techniques for the resolution of CSPs are based on backtracking and constraint propagation. CSPs can be represented in constraint logic programming, which embeds constraints into a logic program.

Eventually, the notion of the model finding problem in MDE is similar to the notion of a CSP, but it is based on a more complex structure (the variables are objects, links, etc.). In [7] a translation of the model finding problem for OCL-annotated models into a CSP is described. Given a $\mathcal{M}$ and a set of OCL constraints $C_{\mathcal{M}}$, the approach defines how to infer a CSP P that is consistent iff. $\langle \mathcal{M}, C_{\mathcal{M}} \rangle$ is satisfiable. The solutions to P correspond to instances of $\mathcal{M}$. Technically, the derived CSP P consists of two sub-problems

$$P_{\text{structure}} = (\ Cardinalities, CardinalityConstraints\ )$$

and

$$P_{\text{global}} = (\ Instance, InstanceConstraints\ )$$

where the solutions of the first sub-problem are (potentially) valid sizes for the sets of objects and links. The variables of the second sub-problems include lists of objects and links. Iterating the solutions to the first sub-problem (by backtracking), these lists are instantiated (i.e., a length is assigned to them) in the second sub-problem.

In a nutshell, two kinds of constraints are employed in the second sub-problem. The first kind includes constraints over Boolean and integer arithmetic, for which constraint propagation is available in most constraint programming languages. The second kind includes specific constraints to represent, for example, navigation operations. The derivation of these constraints can be implemented, for example, using *suspended goals*.

We have implemented our String reasoning approach in EMFtoCSP [10], which is the successor of UMLtoCSP [7] and supports EMF metamodels as well as UML models. Other approaches that also express model finding in terms of constraint programming include [21, 20, 8].

2.3 Formal String Reasoning

Reasoning on strings has been performed in various formalisms, both for bounded and unbounded strings. Solvers for Satisfiability Modulo Theories (SMT) commonly support theories that can be used to represent strings, such as arrays and bit-vectors. Also, as a string is only a specific case of a list, any theory of lists can be applied, too. Recently, a working group for the development of a theory for strings and regular languages has been formed [4].

In addition to the theory-based approaches, several approaches reason about string using finite automata and regular expression, e.g., [17,16,13,26]. [15] incorporates regular languages into a CSP. In [28] the authors perform path analysis for String-manipulating programs using SMT, employing a two step approach similar to ours, determining an approximation of string lengths in the first step first.

In comparison to these approaches, our approach is less complex in the sense that it performs symbolic reasoning only over the lengths of strings. Due to its two-step nature, it is not suited to solve real hard (NP-hard), non-tractable string problems in reasonable time. However, it efficiently solves several less hard string problems with minimal overhead, which makes it suitable for embedding into a more general model finding procedure.

3 Lightweight String Reasoning for OCL

We now present our approach for solving OCL constraints that use the OCL data type String and its operations. Our approach can be easily integrated into existing approaches to satisfiability checking (model finding) of models and OCL constraints such as the ones presented in Sect. 2.

At its core, we formulate the problem as a CSP. We provide five constraint predicates for strings that can be used to translate OCL constraints containing string expressions. These five predicates correspond directly to five core operations of the OCL data type String, namely equality (=), size, substring, concat, and indexOf. Our five constraints predicates are resolved into constraints on integers and Booleans in two rewriting steps (described as constraint handling rules, which we introduce below). Fig. 1 depicts this process and will be explained in the following.

We assume an OCL model and its constraints have been translated into a CSP $P = (V, C)$ (e.g., as described in [7], c.f. Sect. 2.2). V includes variables of type String and C includes constraints over these variables. In the first step, we infer additional constraints on the lengths of the individual string variables. The second step then translates the string constraints into constraints on the elements of the strings (i.e., their characters). The result of this two-step process is a CSP that can be solved using off-the-shelf solvers for linear constraints.

More formally, we first employ a rewriting operation $C \rightsquigarrow C_{\text{length}}$ (described in Sect. 3.4) that extends C by additional constraints over the lengths of all string variables in V. Then, we require that a solution to the *length sub-problem* is chosen, which introduces a potential choice point in a backtracking search process. We refer to the set of constraints in which the length variables are bound to fixed lengths as $C_{\text{length},i} \rightsquigarrow C_{\text{inst},i}$, where i shall number the different solutions of the length sub-problem.

The second rewriting operation $C_{\text{length},i} \rightsquigarrow C_{\text{inst},i}$ then (a) unifies all symbolic string variables s_i with lists of individual element variables $\langle s_{i,1}, s_{i,2}, \ldots \rangle$, and (b) rewrites the semantics of the string constraints into into Boolean and integer constraints over the element variables. The solutions to $(V, C_{\text{inst},i})$ are solutions to P. If $C_{\text{inst},i}$ has no solution, the next valid solution $i + 1$ to the length sub-problem must be selected. If there is not further solution to the length subproblem, P is *inconsistent*.

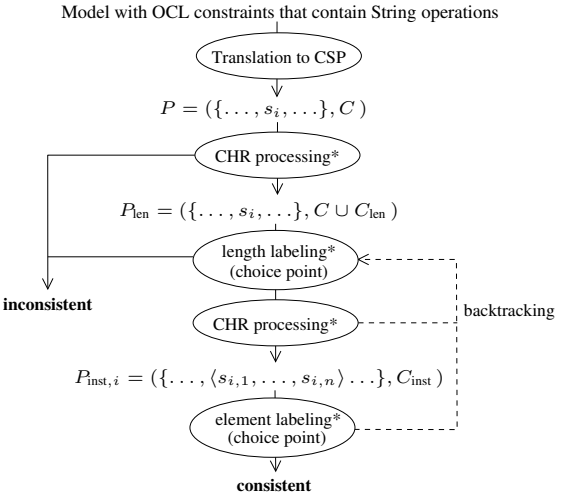


Fig. 1. The decision procedure for CSP with String constraints, including CHR processing. All processing steps (indicated with a star) are supposed to perform constraint propagation on linear and Boolean constraints.

The remaining section first describes the five OCL operations supported more precisely in Sect. 3.1. We provide our five String constraint predicates in Sect. 3.2. Sect. 3.3 gives a short primer on the constraint handling rules formalism, which we employ to define the two rewriting steps in Sections 3.4 and 3.5. We provide a complete derivation example in Sect. 3.6.

3.1 Considered OCL String Operations

The OCL specification [22] defines several operations for the data type String. In this work we consider a subset of five important core operations: the length of a string (s_1.concat(s_2)), the concatenation of two strings (s_1.concat(s_2) resp. $s_1 + s_2$), the indexed substring of a string (s_1.substring(i, j)), and the containment of a string in another one (s_1.indexOf(s_2)).

In accordance with the OCL specification, we assume the semantics of the core operations as follows: Let $\mathcal{A}$ be the set of characters, and $\mathcal{A}^*$ be the set of all strings (sequences of characters). The semantics of this core can be described by a set of interpretation functions $I(\text{size}) : \mathcal{A}^* \to \mathbb{N}$, $I(\text{concat}) : \mathcal{A}^* \times \mathcal{A}^* \to \mathcal{A}^*$, $I(=) : \mathcal{A}^* \times \mathcal{A}^* \to \mathbb{B}$, $I(\text{substring}) : \mathcal{A}^* \times \mathbb{N} \times \mathbb{N} \to \mathcal{A}^*$, and $I(\text{indexOf}) : \mathcal{A}^* \times \mathcal{A}^* \to \mathbb{N}$, as follows. $I(\text{size})(s)$ is defined to be the length of the sequence s (and can be 0). $I(=)(s_1, s_2)$ is true iff s_1 and s_2 have the same lengths and are equal element-wise, and false otherwise. $I(\text{concat})(s_1, s_2)$ is $s_1 \circ s_2$ (concatenation of sequences). $I(\text{substring})(s, i, j)$ is the subsequence from i to j, and is only defined for $1 \leq i \leq j \leq |s|$ according to the OCL specification. $I(\text{indexOf})(s_1, s_2)$ is the index of the first occurrence of s_2 in s_1 when s_1 is non-empty and 0 otherwise. In OCL, no string is a substring of the empty string (not even the empty string).

3.2 String Constraints

We define five string constraint predicates that are sufficient to express OCL constraints that use the core operations on strings as a CSP. They correspond directly to the operations (cf. Sect. 3.1). Examples for the encoding follow after this section.

Let s, $s_1, \ldots s_3$ denote string variables, and l, i ,and j denote integers. The five constraints are $\mathrm{len}(s, l)$ – the length of s is l, $\mathrm{eq}(s_1, s_2, b)$ – s_1 and s_2 are equal iff. b is true, $\mathrm{con}(s_1, s_2, s_3)$ – s_3 is the concatenation of s_1 and s_2, $\mathrm{sub}(s_1, i, j, s_2)$ – s_2 is the substring of s_1 from i to j, and $\mathrm{idx}(s_1, s_2, i)$ – the number i is either the first position at which s_2 occurs in s_1, or 0 if s_2 does not occur in s_1.

Notice that the string equality constraint has a *reified* form, which is necessary to deal with string equality in the linear and propositional reasoning. For example, the OCL constraint $s_1 = s_2 \,\mathrm{implies}\, s_1 = s_3$ would be expressed as

$$P = \big(\{s_1, s_2, s_3\}, \{(b_1 \rightarrow b_2), \mathrm{eq}(s_1, s_2, b_1), \mathrm{eq}(s_1, s_3, b_2)\}\big)$$

where $\rightarrow$ is assumed as a predefined constraint over two Booleans.

3.3 Constraint Handling Rules

To describe the rewriting operations on constraints, we employ Constraint Handling Rules (CHR), which is a well-known formalism and has several implementations available. However, our rewriting rules can also be implemented in other ways easily. As we make only very limited use of the formalism, we only introduce a simplified form here. For a thorough presentation of the formalism we refer to [11] and [24]. In our restricted context, a constraint handling rule has one of the three syntactic forms.

$$\begin{aligned} \mathit{rulename} \;@\; c_1, \ldots, c_m &\iff c'_1, \ldots c'_n \\ \mathit{rulename} \;@\; c_1, \ldots, c_m &\implies c'_1, \ldots c'_n \\ \mathit{rulename} \;@\; c_1, \ldots, c_k \setminus c_{k+1}, \ldots, c_m &\implies c'_1, \ldots c'_n \end{aligned}$$

where c_i and c'_i are constraints that typically share some variables. The common semantics of these rules is that they match a pattern of constraints $c_1, \ldots, c_m$ in the constraint store (which, in our case, is the set of constraints in the CSP). The constraints in the pattern are related by their common variables, for example, as in the pattern $c_1(s, i), c_2(s, j)$. The first kind of rules above is called a *simplification* rule. It removes the matched constraints $c_1, \ldots, c_m$ from the constraint store and replaces them by new constraints $c'_1, \ldots c'_n$. The second kind is called a *propagation* rule, which also adds $c'_1, \ldots c'_n$ to the imposed constraints, but also keeps $c_1, \ldots, c_m$ in the store. The third kind is called a *simpagation* rule and is a mixture of the former two. It keeps $c_1, \ldots, c_k$, but replaces $c_{k+1}, \ldots, c_m$ by $c'_1, \ldots c'_n$. The execution of a set of CHR rules is terminated when no more rules can be applied. For propagation rules, the CHR environment ensures that such rules are applied only once per match of their pattern.

3.4 First Rewriting Step: The Length Sub-problem

We now define the rules that infer linear additional constraints over the lengths of the string variables. This constitutes the first rewriting step in our approach (cf. Fig. 1).

Definition 1. *Length Inference Rules*

$$\textit{len-dom} \mathbin{@} \mathrm{len}(s,l) \implies 0 \le l \le \mathit{MaxLen}$$

$$\textit{len-one} \mathbin{@} \mathrm{len}(s,l_1) \setminus \mathrm{len}(s,l_2) \iff l_1 = l_2$$

$$\textit{eq-len} \mathbin{@} \mathrm{eq}(s_1,s_2,r), \mathrm{len}(s_1,l_1), \mathrm{len}(s_2,l_2) \implies r \rightarrow l_1 = l_2$$

$$\textit{con-len} \mathbin{@} \mathrm{con}(s_1,s_2,s_3), \mathrm{len}(s_1,l_1), \mathrm{len}(s_2,l_2), \mathrm{len}(s_3,l_3) \implies l_3 = l_1 + l_2$$

$$\textit{sub-len} \mathbin{@} \mathrm{sub}(s_1,i,j,s_2), \mathrm{len}(s_1,l_1), \mathrm{len}(s_2,l_2) \implies \begin{array}{l} l_2 = (j-i+1), \\ 1 \le i \le j \le l_1 \end{array}$$

$$\textit{idx-len} \mathbin{@} \mathrm{idx}(s_1,s_2,i), \mathrm{len}(s_1,l_1), \mathrm{len}(s_2,l_2) \implies \begin{array}{l} i \neq 0 \rightarrow l_1 \ge l_2, \\ l_1 = 0 \rightarrow i = 0 \end{array}$$

$$\textit{eq-refl} \mathbin{@} \mathrm{eq}(s,s,b) \iff b \leftrightarrow \mathrm{true}$$

$$\textit{eq-one} \mathbin{@} \mathrm{eq}(s_1,s_2,b_1) \setminus \mathrm{eq}(s_1,s_2,b_2) \implies b_1 \leftrightarrow b_2.$$

The propagation rule len-dom constrains all strings to be finite, using a maximum length *MaxLen* that can be any positive number. For example, given $\mathit{MaxLen} = 100$, the CSP $(\{s\}, \mathrm{len}(s,l)\})$ would be rewritten to $(\{s\}, \mathrm{len}(s,l), 0 \le l \le 100\})$. The simpagation rule len-one removes multiple lengths constraints for the same string and replaces them by linear constraints over the length variables. For example, $P = \big(\{s\}, \mathrm{len}(s,l_1), \mathrm{len}(s,4), l_1 \le 3\}\big)$ would be rewritten to $\big(\{s\}, \mathrm{len}(s,l_1), l_1 \le 3, l_1 = 4\big)$ – which a linear constraint solver would detect as unsatisfiable for any l_1. The propagation rule eq-len poses conditional equality. The rules con-len, sub-len, and idx-len generate the expected constraints in the same manner. Finally, the simplification rules eq-refl and eq-one include reflexivity and *tertium non datur* into the length inference.

Please note that we included the last two rules, eq-refl and eq-one for practical reasons only, as a performance optimization (these rules add only little overhead in terms of constraint processing). They are theoretically not required, because equality of strings will be translated into pair-wise equality of their elements (see below). On the contrary, we do not include transitivity (and neither symmetry) here, in order to keep the approach lightweight, as transitivity can lead to an exponential number of equality constraints. Theoretically, $\mathrm{eq}(s_1,s_2,\mathrm{true})$ is of course transitive (and $\mathrm{eq}(s_1,s_2,b)$ is symmetric), because the equality on the element variables has exactly these properties. We found, however, that turning these properties into rules produces too much overhead for the kind of (lightweight) string problems we consider.

When no more of the rules in Def. 1 can be applied (i.e., the execution of these rules has terminated), a ground assignment of all length variables has to be selected. In general, this introduces a choice point. Consider, for example, if we derived $P = \big(\{s_1,s_2\}, \{\mathrm{len}(s_1,l_1), \mathrm{len}(s_2,l_2), 1 \le l_1 \le 4, 1 \le l_2 \le 4, s_1 = s_2 - 1\}\big)$, the choices for l_1, l_2 would be $1,2$ and $2,3$. Assuming we select $1,2$ first, we would proceed with the CSP $P = \big(\{s_1,s_2\}, \mathrm{len}(s_1,1), \mathrm{len}(s_2,2)\big)$

3.5 Second Rewriting Step: Resolve String Constraints to Element Constraints

Given that a solution to the length sub-problem has been selected (i.e., all length variables have ground values), the following rule unifies all string variables with lists of

element variables, where each element variable is constrained to be in the range of the alphabet $\mathcal{A}$, for which charnums is assumed to assign a corresponding set of integers from 1 to $|\mathcal{A}|$.

Definition 2. *Structural Instantiation Rule*

$$\textit{len-inst} \ @ \ \mathrm{len}(s, n) \iff s = \langle x_1, \ldots, x_n \rangle, x_1 \in \textit{charnums}(\mathcal{A}), \ldots, x_n \in \textit{charnums}(\mathcal{A})$$

After all string variables have been instantiated using rule len-inst, all string constraints in C are finally replaced ($\Leftrightarrow$) by linear and Boolean constraints on the individual element variables using the following rewrite rules.

Definition 3. *Linear Representation Rules*

$$\textit{eq-inst} \ @ \ \mathrm{eq}(\langle x_1, \ldots, x_n \rangle, \langle y_1, \ldots, y_n \rangle, r) \iff r \leftrightarrow \left(\bigwedge_{1 \leq i \leq n} x_i = y_i \right)$$

$$\textit{con-inst} \ @ \ \mathrm{con}(\langle x_1, \ldots, x_m \rangle, \langle y_1, \ldots, y_n \rangle, \langle z_1, \ldots, z_{m+n} \rangle) \iff \left(\left(\forall_{1 \leq i \leq m} \ x_i = z_i \right), \left(\forall_{1 \leq j \leq n} \ y_j = z_{m+j} \right) \right)$$

$$\textit{sub-inst} \ @ \ \mathrm{sub}(\langle x_1, \ldots, x_m \rangle, i, j, \langle y_1, \ldots, y_n \rangle) \iff \forall_{0 \leq l \leq (n-m)} \left(i = l + 1 \rightarrow \left(\bigwedge_{1 \leq k \leq n} x_{k+l} = y_k \right) \right)$$

$$\begin{aligned} \textit{idx-inst} \ @ \ & \mathrm{idx}(\langle x_1, \ldots, x_m \rangle, \langle y_1, \ldots, y_n \rangle, i) \iff \\ & \forall_{0 \leq l \leq (n-m)} \Big(r' \leftrightarrow \left(\bigwedge_{1 \leq k \leq n} \ x_{k+l} = y_k \right), \\ & \qquad r' \rightarrow p_l = l + 1, \neg r' \rightarrow p_l = 0 \Big), \\ & i = \textit{min}^*(p_0, \ldots, p_{(n-m)}) \end{aligned}$$

where $\forall$ is used to express a set constraints and $b \leftrightarrow \bigwedge(\ldots)$ is used to represent that b is constrained to be the result of the conjunction of a set of Boolean values. r' and p_l are fresh variables and min* is the usual minimum function on natural numbers with the exception that 0 is regarded as the largest number.

The rule eq-inst poses one Boolean constraint that r is equal to the Boolean value of the conjunction of the element-wise equality of both strings. The rule con-inst poses one equality constraint for each element of the concatenated string. The rule sub-inst poses one constraint for each possible offset l of y in x. Please notice that we can safely assume m and n to be ground values, whereas i and j can be variables. The rule does not pose further constraints on j, because j has already been expressed dependent on i by rule sub-len before (see Def. 1). Rule idx-inst is of similar nature, only that it introduces one integer variable p_l per possible offset of y in x, which is constrained to be either the position of y in x or 0. The result of i is then expressed by the (modified) minimum of these values. This minimum can be rewritten syntactically into basic linear constraints.

After the rules of Def. 3 have been applied, the resulting CSP contains only relational and arithmetic constraints on Boolean and integer values. It can be solved using off-the-shelf solvers.

3.6 Derivation Example

To illustrate the definitions so far, we now provide a complete derivation example that shows (i) how an OCL constraint is expressed in terms of our string constraints, (ii) how the length inference takes place, and (iii) how the constraints are finally resolved for a given length assignment. We consider the following OCL constraint:

$$s_1 = (s_2 + s_3).\,\mathrm{substring}(2,3)\,\mathrm{and}(s_2 + s_3).\,\mathrm{size}() < 10$$

It can be represented as a CSP in a straight-forward manner using the constraints previously introduced. Because constraints predicates cannot be nested, the CSP requires two additional string variables s_4, s_5 to express the results of the subexpressions $s_2 + s_3$ and $(s_2 + s_3).\,\mathrm{substring}(2,3)$. This means that, while we are eventually interested in three strings, s_1,s_2, and s_3, we have to regard five strings in the CSP. Note that a len constraint with a free length variable is included for each string. Let $MaxLen = 1000$. We assume the built-in reified Boolean resp. linear constraints $\wedge(x, y, b)$ and $<(x, y, b)$, for which the third argument is the truth value of $x \wedge y$ resp. $x < y$. The resulting CSP is

$$\begin{aligned}\big(\{s_1,\ldots,s_5\},\{\ &\mathrm{len}(s_1,l_1),\mathrm{len}(s_2,l_2),\mathrm{len}(s_3,l_3),\mathrm{len}(s_4,l_4),\\ &\mathrm{len}(s_5,l_5),\wedge(b_1,b_2,\mathrm{true}),\mathrm{eq}(s_1,s_5,b_1),\\ &\mathrm{con}(s_2,s_3,s_4),\mathrm{sub}(s_4,2,3,s_5),<(l_4,10,b_2)\}\big)\end{aligned} \tag{1}$$

We assume that propagation on Boolean predicates will unify $b_1 = \mathrm{true}$ and $b_2 = \mathrm{true}$ from $\wedge(b_1, b_2, \mathrm{true})$. The rewriting rules for lengths apply as follows: len-dom infers a range constraint for each string length, eq-len infers $l_1 = l_5$, con-len infers a linear constraint $l_4 = l_2 + l_3$, sub-len infers the linear constraints $l_5 = 2$ and $1 \le i \le j \le l_4$. Assuming that the linear constraints propagate their bounds, the simplified resulting CSP is

$$\begin{aligned}\big(\{s_1,\ldots,s_5\},\{\ &\mathrm{len}(s_1,2),\mathrm{len}(s_2,l_2),\mathrm{len}(s_3,l_3),\mathrm{len}(s_4,l_4),\\ &\mathrm{len}(s_5,2),\mathrm{eq}(s_1,s_5,\mathrm{true}),\mathrm{con}(s_2,s_3,s_4),\\ &\mathrm{sub}(s_4,2,3,s_5),0 \le l_2 < 10,0 \le l_3 < 10,\\ &l_4 = l_2 + l_3,3 \le l_4 < 10\}\big).\end{aligned}$$

with l_1 and l_5 being removed from the variables (as $l_1 = l_5 = 2$), At this point, no more rules are applicable until we select a solution to the length sub-problem

$$\big(\{l_2,\ldots l_4\},\le l_2 \le 10,0 \le l_3 < 10,l_4 = l_2 + l_3,3 \le l_4 < 10\}\big) \tag{2}$$

We assume the assignment $\{l_1 \mapsto 2, l_2 \mapsto 2, l_3 \mapsto 3, l_4 \mapsto 3, l_5 \mapsto 2\}$ is chosen and apply rule len-inst. The CSP to solve is now

$$\begin{aligned}\big(\{\ &s_{1,1},s_{1,2},s_{2,1},s_{2,2},s_{3,1},s_{4,1},s_{4,2},s_{4,3},s_{5,1},s_{5,2}\},\\ \{\ &\mathrm{eq}(s_1,s_5,\mathrm{true}),\mathrm{con}(s_2,s_3,s_4),\mathrm{sub}(s_4,1,3,s_5),\\ &\mathrm{sub}(\langle s_{4,1},s_{4,2}\rangle,1,3,\langle s_{5,1},s_{5,2}\rangle),\\ &s_{1,1},s_{1,2},s_{2,1},s_{2,2},s_{3,1},s_{4,1},s_{4,2},s_{4,3},s_{5,1},s_{5,2} \in \mathrm{charnums}|\mathcal{A}|\}\big)\end{aligned}$$

with $s_1 = \langle s_{1,1}, s_{1,2}\rangle$, $s_2 = \langle s_{2,1}, s_{2,2}\rangle$, $s_3 = \langle s_{3,1}\rangle$, $\langle s_{4,1}, s_{4,2}, s_{4,3}\rangle$, and $\langle s_{5,1}, s_{5,2}\rangle$.

For this example, the resolving of the rules eq-inst, con-inst, and sub-inst finally unifies several variables, leaving a CSP that is *solved* (i.e., a CSP for which every assignment of character numbers to the element variables is a solution).

$$\left(\{s_{1,1}, s_{1,2}, s_{2,1}, s_{2,2}, s_{3,1}\}, \{s_{1,1}, s_{1,2}, s_{2,1}, s_{2,2}, s_{3,1} \in \text{charnums}|\mathcal{A}|\}\right)$$

All $|\mathcal{A}|^5$ solutions to this CSP are solutions to the original CSP (1) under the assignment of $\{l_2 \mapsto 2, l_3 \mapsto 1, l_4 \mapsto 3)\}$ to the length subproblem (2) with $(s_1, \langle s_{1,1}, s_{1,2}\rangle)$, $(s_2 = \langle s_{2,1}, s_{2,2}\rangle$, $s_3 = \langle s_{3,1}\rangle$, $s_4 = \langle s_{2,1}, s_{1,1}, s_{1,2}\rangle$, $s_5 = \langle s_{1,1}, s_{1,2}\rangle$. Naturally, in other cases not all assignments to the final CSP are solutions to the problems. In general, *search* must be performed for a solution. The impact of this search on the scalability of our approach is discussed in the next section.

4 Limits and Scalability

The rewriting rules presented in the previous section do not constitute a self-contained decision procedure. They have to be combined with a solver that is capable to reason about propositional logic and bounded integer constraints, for which decision procedures are available in various solvers. The presented constraint handling rules are terminating and confluent. This means that, in theory, every CSP on strings with bounded lengths can be solved using our approach if it can be expressed using the provided string constraint predicates and other decidable constraints. In this section, we first provide some experimental results that we gathered using our implementation of the approach, then we discuss scalability aspects in more generality.

4.1 Experimental Results

As mentioned before, we have implemented the described reasoning approach in our model finder EMFtoCSP, using the constraint handling rules support of the ECL^{i}PS^{e}solver. We now employ the model depicted class diagram in Fig. 2 and the following OCL constraints to illustrate performance and scalability aspects of our approach and implementation.

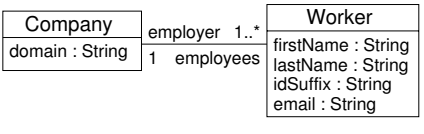


Fig. 2. Example Class Diagram annotated with OCL Constraints for its Strings

The example contains (conditional) string equality, concatenation, and and containment constraints. Due to the existential and universal quantifiers in EmailsUnique (all workers within a company must have the distinct email addresses) and OneSame (at least two workers must share first and last name), a quadratic number of of conditional equality constraints is posed. Both invariants are considerably hard in terms of computational complexity, which is why we have added them to our experiment. The constraint EmailStructured determines the structure of the email attribute in terms of concatenation.

```
context Company inv EmailsUnique:
   worker->forAll(w1,w2 | w1.email = w2.email implies w1 = w2 )
context Company inv OneSame:
  worker->exists(e1,e2 | e1 <> e2 and
                 e1.firstName = e2.firstName and
                 e1.lastName = e2.lastName )
context Worker inv EmailStructured:
  email = firstName + '.' +  lastName + '.' +
          idSuffix + '@' + employer.domain
context Worker inv NoAt:
  firstName.indexOf('@') = 0 and
  lastName.indexOf('@') = 0
```

EMFtoCSP translates the model and its constraints into a CSP as described in Sect. 2.2. In our extended version, the OCL constraints are translated using the string constraints introduced in the previous section. Recall that the original approach consists of two sub-problems *Structure* and *Global*. In the version with string support, the length inference is included into the global sub-problem (i.e., it adds further constraints to this sub-problem). The element labeling then constitutes a new, third sub-problem *Strings*.

Table 1 show the runtimes and linear constraint propagations performed by ECL^{i}PS^e for different instance sizes of the above example. The tests were conducted on a typical office 2.2Ghz laptop running ECL^{i}PS^{e}6.0 on Windows 7. All tests used a maximum string lengths of 1,000.

The table differentiates the runtimes and propagation counts for the CHR processing and the final labeling (i.e., the assignment of ground character values to the elements of all strings). For each test cases we constrained the workers to be evenly distributed among the companies. The table illustrates several aspects. First, we can see that the actual character labeling does not consume a significant amount of time once the constraint handling rules have been processed and the linear constraints have been posted.

Furthermore, we can observe that the runtimes for the cases where few companies have many workers consume the most processing time. This is due to the quadratic number of equality constraints that results from the OCL invariants EmailsUnique and OneSame. For the second row in Table 1 more than 30,000 conditional equality constraints are posed in the *Global* sub-problem. If the number of workers per company is less high, EMFtoCSP scales better, for example, when the ratio is 1:10, as in the third row.

Table 1. Experimental Results

Instance			CHR Processing		Character Labeling	
companies	workers	strings	cpu time	propagations	cpu time	propagations
1	10	41	0.1s	3,112	$\leq$ 0.1s	3,360
1	100	401	10.1s	220,192	1.1s	963,600
10	100	410	0.7s	31,120	$\leq$ 0.1s	33,600
50	100	450	0.4s	14,000	$\leq$ 0.1s	3,200
150	300	1,350	2.3s	42,000	$\leq$ 0.1s	9,600
10	300	1,210	11.7s	219,520	0.6s	440,800

4.2 General Discussion

In general, we can distinguish three categories for the solving of the CSP, where the distinction between the second and third case depends on the capabilities of he solver employed. In a nutshell, our approach works very efficiently in the first category, and less efficient in the others. We have conducted several experiments using our implementation of the approach in EMFtoCSP and the ECL^{i}PS^e solver, and found that several typical patterns of constraints encountered in models fall into the first case (and so does the previously presented example).

1. The *optimal case*: No backtracking is required, every valid length assignment yields a *solved* CSP on the string elements after applying our rewriting rules and performing constraint propagation. In this case, even very high numbers for the maximum string length (e.g., 1000) can be chosen.
2. In the *length search case*, not every valid length assignment yields a consistent CSP on the string elements, but the inconsistency of a chosen length assignment is detected by the solver without actually labeling the elements. A simple example for this case is given by the OCL constraint `s.indexOf('@') = 0 and s.indexOf('@') <> 0`, whose unsatisfiability is not recognized before instantiating the s to its element variables[1]. The ECL^{i}PS^e solver, for example, however detects that the resulting CSP (the third one in Fig. 1) is consistent without backtracking through the possible assignments of values to the element variables, leaving $MaxLen$ choices for the solver before reporting inconsistency. For constraints that fall into this case, the maximum string length must be set to a reasonable small value for practical applications.
3. The *labeling trap*: In this case, the inconsistency is not detected before actually labeling the element values. To ease the labeling trap in practice, the search procedure can be split to perform a two-pass run, where the first pass tries at most one assignment to the element variables for each solution to the length problem. In the second pass, the element labeling is repeated without restrictions. This tweak to the search space traversal helps to circumnavigate the labeling trap for insufficient string lengths.

However, for most constraint patterns typically encountered in 'real live' situations, the labeling trap is not a problem. In fact, as stated before, several common constraint patterns fall even into the first category. Summarizing, we state that our approach is perfectly suited to efficiently handle lightweight string problems that have many solutions, while it is not suited to solve non-tractable string problems (which, in general, are NP-hard), that only have few solutions, and for which the employed solver runs into the labeling trap.

5 Conclusion

In the previous sections, we have presented an approach that translates OCL String constraints into a constraint satisfaction problem that can be solved using off-the-shelf solvers, and which can be integrated easily into existing model finding approaches and

[1] We assume the constraint is translated straightforwardly using two separate idx constraints.

tools for OCL without adding too much overhead to the underlying decision procedures. Our approach is lightweight in the sense that it efficiently finds solutions for many common OCL constraint constellations and it is suited to handle models with thousands of strings. Due to its two-step nature, we can efficiently handle strings of potentially long lengths, which otherwise would lead to search space explosion when directly encoding all strings as bitvectors of the maximum length. We therefore claim that our approach is suited for various practical ('real world') applications. It is, however, not suited to tackle hard, non-tractable string constraints that only have few solutions. These must be addressed either by one-step bit-blasting approaches or by formal regular language reasoning and theorem provers that can handle an appropriate theory.

So far we considered an important subset of OCL string operations. We expect that the remaining operations (e.g., toLowerCase, at, characters, $<$) can be encoded in the same manner. To our knowledge, we are the first ones that integrate reasoning on String constraints into model finding for OCL-annotated models. We have implemented our results into the EMFtoCSP (formerly UMLtoCSP) tool and provided some experimental results. While we used constraint handling rules as a formalism to define our constraint rewriting rules, our approach can be easily implemented in other ways, for example, using suspended goals in constraint logic programming.

As the next step, we will evaluate the effect of using different solvers, for our final CSPs on the element variables and compare their suitability with the constraint logic programming approach. In particular, we hope to apply our approach to the Kodkod encoding of OCL in [18], in order to directly compare the bit-blasting approach and our two-step approach. Furthermore, we will evaluate the applicability and performance of our approach using further, more extensive case studies.

References

1. Ali, S., Iqbal, M.Z.Z., Arcuri, A., Briand, L.C.: A Search-Based OCL Constraint Solver for Model-Based Test Data Generation. In: QSIC, pp. 41–50 (2011)
2. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: UML2Alloy: A Challenging Model Transformation. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 436–450. Springer, Heidelberg (2007)
3. 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)
4. Bjorner, N., Nieuwenhuis, R., Veith, H., Voronkov, A. (eds.): Decision Procedures in Soft, Hard and Bio-ware - Follow Up, vol. 1(7). Dagstuhl Reports (2011)
5. Büttner, F., Cabot, J., Gogolla, M.: On Validation of ATL Transformation Rules By Transformation Models. In: Weißleder, S., Lúcio, L., Cichos, H., Fondement, F. (eds.) Proceedings of MoDeVVa 2011. ACM Digital Library (2012), doi:10.1145/2095654.2095666
6. Cabot, J., Clarisó, R., Guerra, E., de Lara, J.: Verification and validation of declarative model-to-model transformations through invariants. Journal of Systems and Software 83(2), 283–302 (2010)
7. Cabot, J., Clarisó, R., Riera, D.: UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In: Stirewalt, R.E.K., Egyed, A., Fischer, B. (eds.) Proceedings of Automated Software Engineering, ASE 2007. ACM (2007)
8. Cadoli, M., Calvanese, D., De Giacomo, G., Mancini, T.: Finite Satisfiability of UML Class Diagrams by Constraint Programming. In: Proc. of the CP 2004 Workshop on CSP Techniques with Immediate Application (2004)
9. Clavel, M., Egea, M., de Dios, M.A.G.: Checking Unsatisfiability for OCL Constraints. Electronic Communications of the EASST 24, 1–13 (2009)

10. The EMFtoCSP tool. Website, http://code.google.com/a/eclipselabs.org/p/emftocsp/
11. Frühwirth, T.W.: Constraint Handling Rules. In: Podelski, A. (ed.) Constraint Programming: Basics and Trends. LNCS, vol. 910, pp. 90–107. Springer, Heidelberg (1995)
12. 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)
13. Golden, K., Pang, W.: Constraint Reasoning over Strings. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 377–391. Springer, Heidelberg (2003)
14. González, C.A., Büttner, F., Clarisó, R., Cabot, J.: EMFtoCSP: A Tool for the Lightweight Verification of EMF Models. In: Proc. of Formal Methods in Software Engineering: Rigorous and Agile Approaches (FormSERA), Workshop at ICSE (to appear, 2012), http://www.formsera.org/FormSERA
15. Grahne, G., Nykänen, M., Ukkonen, E.: Reasoning about Strings in Databases. J. Comput. Syst. Sci. 59(1), 116–162 (1999)
16. Hooimeijer, P., Veanes, M.: An Evaluation of Automata Algorithms for String Analysis. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 248–262. Springer, Heidelberg (2011)
17. Kiezun, A., Ganesh, V., Guo, P.J., Ernst, M.D., Hooimeijer, P., Ganesh, V., Guo, P.J., Ernst, M.D.: HAMPI: A solver for string constraints. In: International Symposium on Software Testing and Analysis (2009)
18. Kuhlmann, M., Gogolla, M.: Strengthening SAT-Based Validation of UML/OCL Models by Representing Collections as Relations. In: Kolovos, D. (ed.) ECMFA 2012. LNCS, vol. 7349, pp. 32–48. Springer, Heidelberg (2012)
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. Malgouyres, H., Motet, G.: A UML model consistency verification approach based on metamodeling formalization. In: Proceedings of the 2006 ACM Symposium on Applied Computing, SAC 2006, pp. 1804–1809. ACM, New York (2006), http://doi.acm.org/10.1145/1141277.1141703
21. Maraee, A., Balaban, M.: Efficient Reasoning About Finite Satisfiability of UML Class Diagrams with Constrained Generalization Sets. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA. LNCS, vol. 4530, pp. 17–31. Springer, Heidelberg (2007), http://dl.acm.org/citation.cfm?id=1768765.1768767
22. OMG: Object Constraint Language Specification, version 2.3.1 (Document formal/2012-01-01) (2012)
23. 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)
24. Sneyers, J., Weert, P.V., Schrijvers, T., Koninck, L.D.: As time goes by: Constraint Handling Rules. TPLP 10(1), 1–47 (2010)
25. Soeken, M., Wille, R., Drechsler, R.: Encoding OCL Data Types for SAT-Based Verification of UML/OCL Models. In: Gogolla, M., Wolff, B. (eds.) TAP 2011. LNCS, vol. 6706, pp. 152–170. Springer, Heidelberg (2011)
26. Veanes, M., de Halleux, P., Tillmann, N.: Rex: Symbolic Regular Expression Explorer. In: ICST, pp. 498–507. IEEE Computer Society (2010)
27. 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)
28. Bjørner, N., Tillmann, N., Voronkov, A.: Path Feasibility Analysis for String-Manipulating Programs. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 307–321. Springer, Heidelberg (2009), http://dblp.uni-trier.de, doi: http://dx.doi.org/10.1007/978-3-642-00768-2_27

Domain-Specific Textual Meta-Modelling Languages for Model Driven Engineering

Juan de Lara and Esther Guerra

Universidad Autónoma de Madrid (Spain)

Abstract. Domain-specific modelling languages are normally defined through general-purpose meta-modelling languages like the MOF. While this is satisfactory for many Model-Driven Engineering (MDE) projects, several researchers have identified the need for *domain-specific meta-modelling* (DSMM) languages providing customised meta-modelling primitives aimed at the definition of modelling languages in a specific domain, as well as the construction of meta-model families.

In this paper, we discuss the potential of *multi-level meta-modelling* for the systematic engineering of DSMM architectures. For this purpose, we present: (i) several primitives and techniques to control the meta-modelling facilities offered to the users of the DSMM languages, (ii) a flexible approach to define textual concrete syntaxes for DSMM languages, (iii) extensions to model management languages enabling the practical use of DSMM in MDE, and (iv) an implementation of these ideas in the METADEPTH tool.

Keywords: Model-Driven Engineering, Deep Languages, Domain-Specific Meta-Modelling, Textual Concrete Syntax, Multi-Level Transformations.

1 Introduction

Model-Driven Engineering (MDE) promotes an active use of models throughout the software development. These models are sometimes defined using general-purpose languages like the UML, but for restricted, well-known domains, it is also frequent the use of Domain-Specific Modelling Languages (DSMLs).

In current MDE practice, DSMLs are built by the language designer using a meta-model defined with a general-purpose meta-modelling language, like the MOF. This meta-model describes the instances that the users of the language can build in the immediate meta-level below. Thus, DSMLs usually comprise two meta-levels: the definition of the DSML and its usage. More recently, several researchers [9,16] have pointed out the utility of using domain-specific meta-modelling (DSMM) languages as a means to provide domain-specific meta-modelling primitives to customize families of similar DSMLs, e.g., for expressing traceability [16], variability [16] or to define domain-specific process modelling notations [9] and DSML profiles [13]. In this case, the language spans three meta-levels: definition of the DSMM language for a specific domain, definition of the

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 259–274, 2012.

DSML by using the constructs provided by the DSMM language, and usage of the DSML. Unfortunately, existing approaches to DSMM are generally based on a two meta-level setting and the definition of ad-hoc "promotion" transformations between models and meta-models, which makes the adoption of DSMM cumbersome in practice. Moreover, there is no general framework for defining DSMM languages with integrated MDE support.

In this paper, we propose multi-level meta-modelling [5] as an underlying framework for DSMM, and discuss mechanisms to facilitate its adoption in MDE projects. Multi-level meta-modelling allows the definition of *deep* languages, which can be instantiated in more than one meta-level. In this way, the users of the language perform DSMM as, in each meta-level, the constructed models are instances of the upper meta-level but also meta-models w.r.t. the meta-level below. In our context, this means that a DSML is naturally defined as an instance of a DSMM language, and at the same time, it acts as a meta-model for lower meta-levels (i.e., it defines a language). Moreover, we provide: (a) means to customize the meta-modelling features that will be offered to the users of the DSMM languages, (b) a flexible way to define textual concrete syntaxes at every meta-level, and (c) model management languages able to work in a multi-level setting, enabling the use of DSMM in MDE projects (for space constraints we just show the use of model transformations). The framework is supported by MetaDepth [6], a multi-level meta-modelling tool supporting deep characterization through potency [5], and dual ontological/linguistic typing.

Paper Organization. Sec. 2 discusses related research, exposing motivations and needs in the area. Sec. 3 applies multi-level meta-modelling to DSMM and identifies some challenges: how to customise the DSMLs in a DSMM framework (Sec. 4), how to define a concrete syntax for the DSMLs (Sec. 5), and how to manipulate models in a multi-level setting (Sec. 6). Finally, Sec. 7 concludes.

2 Related Work

Several researchers have pointed out the benefits of using DSMM languages. For example, the traceability modelling language [16] (TML) is a DSMM language used to express the allowed traces and constraints between several meta-models. Its rationale is that TML users do not need the full power of EMF or MOF to construct trace meta-models, but they benefit from specific meta-modelling primitives like `Trace` and `TraceLink`. Other DSMM languages are described in [16] to express variability over DSMLs, and to extend DSMLs with interfaces for model reuse. However, no general framework for defining such DSMM languages is proposed. Instead, they use two meta-levels and define ad-hoc "promotion" transformations between models (e.g., a TML model) and meta-models (the resulting trace meta-model). These transformations are a way to emulate three meta-levels within two, hindering the construction of DSMM languages.

In [8], the authors present a language to declare component types with ports, which can be instantiated choosing a number of port instances. This DSMM language is defined in a two meta-level framework extended with capabilities to

instantiate the components, emulating the existence of two meta-levels within one. The price to pay is that one has to manually encode support for the definition of class/features/data types and their instantiation, the definition and evaluation of constraints, and the emulation of inheritance within a single meta-level.

In [13], the UML profiling mechanism is adapted for EMF-based DSMLs. This is another example of DSMM as users need a language to define new profiles and apply them at the meta-level below. Again, a two meta-level setting forces the use of workarounds. In this case, they emulate the existence of attribute instances at the lowest meta-level by the run-time adaptation of the meta-model, injecting new attribute types and classes.

Instead of emulating several meta-levels within two [8] or using artificial workarounds [13,16], we claim that a more natural way to define DSMM languages is the native use of multi-level meta-modelling, also known as deep meta-modelling [6]. Previously, Jablonski et al. [9] used multiple levels to build domain-specific process modelling notations. However, their approach is restricted to meta-modelling, and does not consider the language concrete syntax or its manipulation through model management languages, hindering its use in MDE.

Here, we propose some mechanisms to handle these deficiencies based on some multi-level meta-modelling techniques developed originally by Atkinson and Kühne [3]. There are several multi-level meta-modelling frameworks [1,2,12]. For example, DeepJava [12] extends Java to allow multiple instantiations, whereas the main concern in [1] is the efficient navigation between meta-levels. In [2], the authors discuss the visualization of multi-level languages but do not consider linguistic extensions or the integration with model management languages. All these frameworks either do not consider concrete syntaxes [1,12] or do not integrate model manipulation languages enabling their use for MDE [2].

Although there are many approaches to define textual concrete syntaxes for DSMLs [7,10], their definition for DSMM languages poses new challenges. For instance, there is the need to define the concrete syntax for models several meta-levels below, for which the concrete types that will be available in the models are unknown in advance. Sometimes, it is also necessary to extend the predefined concrete syntax for a particular DSML built using a DSMM language. This enables a progressive refinement of concrete syntaxes at different meta-levels.

Altogether, our contribution is a comprehensive framework to define DSMM language environments based on multi-level meta-modelling. Our approach covers the definition of textual concrete syntaxes, a fine grained customization of the meta-modelling facilities offered to the DSMM language users, and model management languages tailored to a multi-level setting.

3 Deep Meta-Modelling for Domain-Specific Meta-Modelling

In DSMM, users are not given the full power of a general-purpose meta-modelling language, but a more suitable meta-modelling language that contains primitives of the domain, and that is restricted for a particular meta-modelling task.

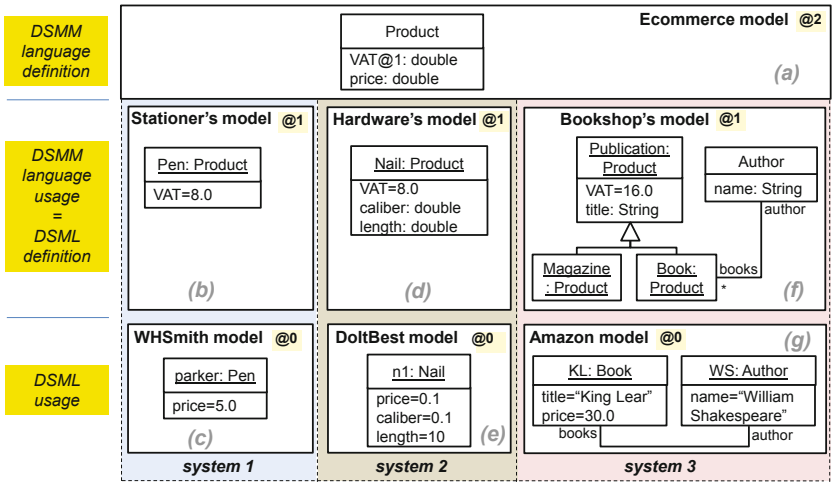


Fig. 1. Definition of a DSMM language for e-commerce (a). Using the language with increasing degrees of extension: no extension (b,c), property extensions (d,e), concept extensions (f,g).

For instance, assume we need to model information systems for e-commerce in various domains. For this purpose, we can build a specialized meta-modelling language that facilitates the construction of DSMLs for each one of these domains. An over-simplified definition of such a meta-modelling language and some of its uses for different scenarios are shown in Fig. 1. Model (a) defines the DSMM language, which is made of a single class `Product`. This language can be used to define a DSML for a stationer system (model (b)), for which we just use the primitives of the domain (e.g., we create an instance of `Product` called `Pen`). Finally, this DSML can be used to define the items in a particular stationery store (i.e., we can create instances of `Pen`, as done in model (c)).

In this way, the definition of a DSML spans three meta-levels: the `Stationer` model is an instance of `Ecommerce`, and `WHSmith` is an instance of `Stationer`. Therefore, it is natural to use a multi-level framework to support the definition and usage of our DSMM language, as these frameworks natively support instantiation across several meta-levels without recurring to artificial workarounds. In a multi-level framework, elements retain both a *type facet* which allows their instantiation in the next meta-level, and an *instance facet* as they are instances of an element at the meta-level above. Thus, model elements become *clabjects* (from the union of "class" and "object") enabling a more uniform way of modelling [3].

DSMM languages normally comprise three meta-levels. To enforce this depth in a multi-level framework, we can use *deep characterization* through the concept of *potency* [3]. The potency is an integer number that can be attached to models, clabjects, attributes and references. If an element is not explicitly given a potency, it receives the one of its immediate container. The potency of an element gets decremented at each meta-level, and when it reaches zero, the element

cannot be instantiated further. Thus, the definition of our DSMM language has potency 2 (see model (a) in Fig. 1, its potency is indicated by '@2'), it gets instantiated into models with potency 1 (middle models), and the instances of these have potency 0 and therefore cannot be instantiated in subsequent meta-levels. In this way, the DSMM language user is effectively performing DSMM because he builds models with potency 1, which are instantiated as models of potency 0.

The potency is also a way for the deep characterization of properties, in order to control the meta-level in which they can be assigned a value. For example, in our DSMM language, all products will receive a price. Hence, `Product` declares an attribute `price` with potency 2, so that it will receive a value two meta-levels below (i.e., each pen has its own price). The potency of the attribute is not explicit, but it is received from the enclosing model. In contrast, the VAT is the same for all products of the same type, hence it has potency 1.

DSMM languages are used to build meta-models for related but different domains. Hence, a particular domain may need to extend the meta-modelling concepts offered by the DSMM language with new domain-specific properties. For example, model (d) in Fig. 1 shows that, in the hardware domain, we need to increase the attributes offered by `Product`. In particular, `Nails` need to define their `caliber` and `length`. These two attributes are specific for nails and therefore cannot be included in the definition of `Product` as they are not general for every domain. Similarly, we may also need to declare domain-specific constraints, e.g., stating that the caliber should be larger than 0.1. Finally, some domains may need to make available new primitives to the users of the DSMLs. For instance, in the bookshop domain, the manipulated products are `Book`s, which have exactly one `Author` (see model (f) in Fig. 1). The concept of `Author` is not included in the DSMM language, and hence we need to include it in the meta-model for bookshops. This is only possible if the DSMM language provides facilities to define new clabjects, references and multiplicities. Moreover, one may wish to group several products in an inheritance hierarchy. For example, both `Magazine`s and `Book`s have a `title` and share the same `VAT` value.

The previous *linguistic extensions* can be supported by a multi-level framework if we use a dual ontological/linguistic typing for the model elements. The ontological typing is a relation within the domain, and refers to the type of which an element is instance. For example, the ontological type of `Pen` is `Product`, and the ontological type of `parker` is `Pen`. Hence, *ontological meta-modelling* is concerned with describing the concepts in a certain domain and their properties [4]. All elements in the top-most model (model (a) in Fig. 1) and some elements in the domain-specific meta-models (e.g., `Author`) may not have ontological type. In contrast, all elements have a linguistic type, which refers to the meta-modelling primitive used to create the element. For example, the linguistic type of `Product`, `Pen` and `parker` is clabject, while the linguistic type of `books` is reference.

One can interpret the union of the three models in each column of Fig. 1 as being conformant to a linguistic meta-model, as shown in Fig. 2(a) (the linguistic meta-model is only partially shown). In our approach, a *linguistic extension* is

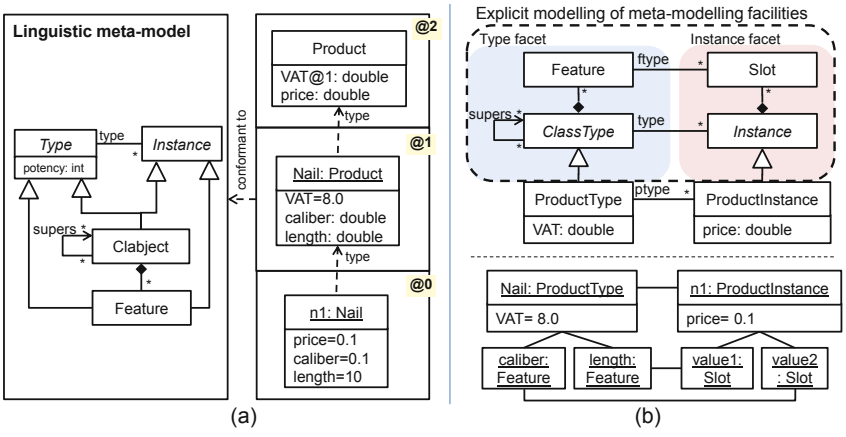


Fig. 2. Defining a DSMM language using: (a) 3 levels and dual typing, (b) 2 levels

an element without ontological typing, like `Author` or the `caliber` attribute in `Nail`. The dual ontological/linguistic typing is very convenient for DSMM as it makes available standard meta-modelling facilities at each meta-level.

Alternatively, Fig. 2(b) shows the definition of our DSMM language using only two meta-levels. This solution makes necessary to explicitly model the desired meta-modelling facilities, and to manually encode the machinery to emulate built-in support for instantiation and constraint checking. Thus, one should build mechanisms taking care of type conformance, data types, definition and evaluation of constraints, and so on.

Altogether, deep meta-modelling facilitates the construction of DSMM languages. However, the following challenges remain:

- We need mechanisms to control the linguistic extensions offered by the DSMM languages, as not any extension may be valid in any domain.
- To be usable in practice, we need to provide a suitable concrete syntax for the DSMM languages and for the DSMLs defined with them. Ideally, both syntaxes should be defined once together with the DSMM language definition, and it should be possible to refine or extend them to take into account the particularities of specific domains.
- To enable the integration of DSMM in MDE projects, we need appropriate model management languages able to work in this multi-level setting.

These three challenges are tackled in the next three sections.

4 Customising the Meta-Modelling Facilities

Designers need to control the way in which the designed DSMM languages will be used and extended. For this purpose, we propose the use of tags to identify the non-extendable language elements, and the use of constraints to ensure a

certain extensibility degree. We will illustrate both control mechanisms using the textual syntax of the METADEPTH [6] tool. For example, the listing shown in Fig. 3 defines our DSMM language for e-commerce systems (lines 1–7), its usage to define a stationer's model (lines 8–10), and an instance of this (lines 12–14), corresponding to models (a, b, c) in Fig. 1.

```
1 strict Model Ecommerce@2 {
2   strict Node Product {
3     VAT@1 : double;
4     price : double;
5     minPrice: $self.price > 0$
6   }
7 }

8 Ecommerce Stationer {
9   Product Pen { VAT = 8; }
10 }
11
12 Stationer WHSmith {
13   Pen parker { price = 5.0; }
14 }
```

Fig. 3. Definition and use of the DSMM language for e-commerce in METADEPTH

The top-model `Ecommerce` lacks ontological type and hence is declared using the keyword `Model` (line 1). This model defines clabject `Product` using the keyword `Node` (line 2). Potencies are specified using the "@" symbol. Constraints can be defined using Java or the Epsilon Object Language (EOL), a variant of OCL that permits side effects [11]. For example, the constraint `minPrice` in line 5 demands a positive price for the products. It receives potency 2 from the model, therefore it will be evaluated two meta-levels below. The model instantiated in lines 8–10 has `Ecommerce` as ontological type, which is used instead of the keyword `Model`.

By default, the meta-models built with a DSMM language can be extended with new primitives (i.e., new clabjects), and any element in the meta-models can be extended with new features. To fine tune the extensibility of a DSMM language, our first proposal is a tagging control mechanism to identify the non-extensible elements. In this way, if the model with the DSMM language definition is tagged as `strict`, it will not be possible to add clabjects without an ontological typing in the next meta-level. If a clabject is tagged `strict`, their instances are forbidden to define new attributes, references or constraints. In the previous listing, both the `Ecommerce` model and the `Product` node are `strict`. Thus, we can use the DSMM language to build the stationer's model in Fig. 1, but not the hardware model (as `Product` instances cannot be extended) or the bookshop model (as `Author` has no ontological type).

If an element is not tagged `strict`, then we may need to control its allowed linguistic extensions. For example, we may like each `Product` instance at potency 1 to declare an attribute acting as identifier, which will receive a value at potency 0. Even though we could declare such a field at meta-level 2 with potency 2, here we may wish to let the decision of the attribute name and type (e.g., `String` or `int`) to the meta-level 1. For this purpose, we propose defining constraints that can make use of facilities of the linguistic meta-model. Fig. 4 shows a constraint, with potency 1, demanding the linguistic extension of all `Product` instances with some attribute tagged as identifier. The method `newFields` belongs to the API of METADEPTH's linguistic meta-model, and returns a collection with the new attributes declared in a meta-level. The method `isId` checks if a field is

an identifier. As this constraint has potency 1, it will be evaluated at the next meta-level, where the DSMM language is used.

```
1  Node Product {
2    ...
3    extid@1: $self.newFields(). exists(f | f.isId())$
4  }
```

Fig. 4. Constraint demanding a linguistic extension

Finally, as the next section shows, we can also control the allowed linguistic extensions syntactically through the design of an appropriate concrete syntax.

5 Designing the Concrete Textual Syntax

Even though deep meta-modelling enables DSMM, our goal is building DSMM languages, and therefore we need to design a concrete syntax for them (in addition to their abstract syntax). In the previous section, we used the default textual concrete syntax that METADEPTH makes available to model uniformly at every meta-level. However, this syntax usually leads to verbose model specifications, while we may prefer a more compact, domain-specific representation. For example, instead of creating instances of `Product` using `Product Pen{VAT=8;}`, we may like a more concise syntax for product instantiation like `Pen(8%)`.

Should the designer only had to define the syntax of the DSMM language, he may use existing tools for describing textual syntaxes like xText [15], TCS [10] or ANTLR [14]. However, as Fig. 5 illustrates, a multi-level architecture poses some challenges that these tools are not able to deal with, since the designer has to provide a syntax for the languages built with the DSMM language as well. In this way, when defining a DSMM language, the designer has to provide both the syntax of the models at meta-level 1 (i.e., of the domain-specific meta-models) and the syntax of the models at level 0 (i.e., of the meta-model instances). For this purpose, we assign to each concrete syntax definition a potency governing the meta-level at which it is to be used. Thus, the syntax with potency 1 will be used in the next meta-level, and the one with potency 2 will be used two meta-levels below. Moreover, it should be possible to refine the syntax initially defined for the models at meta-level 0, in order to introduce domain-specific constructs and describe the syntax of any linguistic extension.

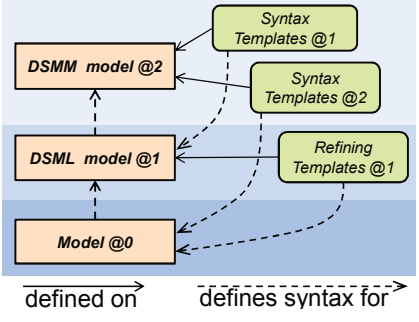


Fig. 5. Defining the concrete syntax

Following this idea, we have created a template-based language to define textual concrete syntaxes in METADEPTH. Using this language, the syntax of each

clabject is defined through a template, which has a potency attached, controlling the meta-level at which the template will be applied. Another template declares the syntax of the model itself. As an example, Fig. 6 shows to the left the definition of the concrete syntax for our example DSMM language, whereas the right corresponds to the syntax for models at meta-level 0.

```
1 Syntax for Ecommerce [".ecommerce_mm"] {
2  template@1 TEcommerce for Model Ecommerce:
3     "id '{' &TProduct* '}'"
4  template@1 TProduct for Node Product:
5     "id '(' #VAT '%' ')'"
6 }
```

```
7 Syntax for Ecommerce [".ecommerce"] {
8  template@2 DeepProds for Model Ecommerce:
9     "typename id '{' &DeepProd* '}'"
10 template@2 DeepProd for Node Product:
11    "typename id '(' #price '€' ')'"
12 }
```

Fig. 6. Defining the concrete syntax for models at levels 1 (left) and 0 (right)

The first line in the definition to the left declares the language to which the syntax applies (the `Ecommerce` model) as well as the associated file extension (*ecommerce_mm*). Its two templates have potency 1, therefore they correspond to the syntax of the DSMM language (i.e., the templates will be used in the next meta-level). In particular, lines 2–3 define the syntax of the instances of the `Ecommerce` model, whereas lines 4–5 define the syntax of the clabject `Product`. The keyword "id" stands for the identifier of an element (see lines 3 and 5), whereas the attributes of a clabject can be referenced by using the prefix "#" (like `VAT` in line 5). Templates can refer to other templates, as in line 3, where it is indicated that the instances of `Ecommerce` can contain zero or more `Product` instances ("&TProduct*"). Using this textual syntax, we can specify the `Stationer` model as `Stationer{ Pen(8%) }`.

The right of the same listing declares the syntax for the models at meta-level 0, which will be stored in a separate file with extension *ecommerce*. In this case, all templates have potency 2. At this point, we do not know the model type for which the syntax is defined, but we only know that it will be an indirect instance of `Ecommerce` (line 8). To access the name of the concrete type we use the keyword "typename", which is interpreted by the parser to check that it is an indirect instance of `Ecommerce` (line 9). The same applies to the definition of templates for clabjects (lines 10–11). With this syntax we can write `Stationer WHSmith { Pen parker(5.0€) }` to instantiate model `Stationer`.

Templates can include "semantic" actions (e.g., to initialize fields of the created clabjects) and may have several syntactic expressions. For instance, we can insert in line 12 of the previous listing `‘‘typename id’’ with ‘‘#price = 0’’` to permit defining products without a price, which gets initialized to 0.

Finally, similar to [7], we can use a single template to define the syntax of several clabjects in the same inheritance hierarchy. Thus, if a clabject does not have an associated template, it uses the template of its closest ancestor in the inheritance hierarchy. A useful keyword in this case is "type", which gets substituted by the name of the clabject. For example, given a clabject "B" inheriting from "A", and a template attached to "A" with body `‘‘type id’’`, then writing "`A a`" creates an `A` instance, while typing "`B b`" creates a `B` instance. As

a difference with "typename", "type" is used for direct types (i.e., at adjacent meta-levels) expecting exactly `A` or `B`. In contrast, "typename" is used for indirect types (i.e., at non adjacent meta-levels) and induces a checking that the name typed in place of "typename" is an indirect instance of the clabject the template is attached to.

5.1 Customising the Meta-Modelling Facilities at the Syntax Level

In Section 4, we showed how to customise the extensibility of a DSMM language at the abstract syntax by identifying strict (i.e., non-extensible) elements and constraining the kind of allowed extensions. These design decisions should be reflected in the concrete syntax of the DSMM language as well, to discard forbidden extensions syntactically even before than semantically.

Our template language provides the following keywords to customise the allowed extensions for a DSMM language at the concrete syntax: `flingext` to allow declaring new fields with no ontological type, `lingext` to allow the addition of new clabjects with no ontological type, `constraint` for declaring constraints, and `super` to define new inheritance relations for clabjects. Moreover, two additional keywords allow defining how these extensions should be instantiated at level 0: `flinginst` for field instances and `linginst` for clabject instances.

For example, the listing shown to the left of Fig. 7 provides a concrete syntax enabling the definition of new fields and constraints in the instances of `Product`, due to the expression in line 6. Moreover, line 10 enables the instantiation of those extra fields in indirect instances of `Product`. Note that, this time, the concrete syntaxes of models at levels 0 and 1 are defined together, and have associated the same file extension. The listing to the right of the figure shows the definition of two models using this concrete syntax. The model `Hardware` in lines 1–7 (in the column to the right of Fig. 7) is an instance of `Ecommerce`, while the model in lines 9–11 is an instance of `Hardware`.

```
1  Syntax for Ecommerce [".ecommerce"] {
2   template@1 TEcommerce for Model Ecommerce:
3     "id '{' &TProduct* '}'"
4   template@1 TProduct for Node Product:
5     "id '(' #VAT '%' ')' '{'
6           (flingext ';' | constraint)* '}' "
7   template@2 DeepProds for Model Ecommerce:
8     "typename id { &DeepProd* }"
9   template@2 DeepProd for Node Product:
10    "typename id '(' #price '€' flinginst* ')' "
11 }
```

```
1  Hardware {
2    Nail (8.0%){
3      caliber : double;
4      length : double;
5      bigger : $self.caliber>=0.1$
6    }
7  }
8
9  Hardware DoItBest {
10   Nail n1(0.1€ caliber=0.1 length=10)
11 }
```

Fig. 7. Extensible textual syntax (left), and its use (right)

The listing to the left of Fig. 8 illustrates the use of `lingext` to allow the definition of new clabjects at meta-level 1 (line 4), and the use of `supers` to allow inheritance between instances of `Product` (line 7). The right of the same

figure uses this syntax to define model (f) in Fig. 1. In the listing, `Magazine` (line 18) and `Book` (line 19) inherit from `Publication`, `Book` defines a new field `author` (line 20), and there is a new clabject `Author` with no ontological type (lines 22–24).

```
1  Syntax for Ecommerce [".ecommerce"] {
2
3   template@1 TEcommerce for Model Ecommerce
          :
4     "id '{' (&TProduct | lingext )* '}' "
5
6   template@1 TProduct for Node Product:
7     "id ('(' #VAT '%' ')')? ('extends' supers)?
8        (';' | '{' (flingext ';' | constraint)* '}') "
9
10  ...
11 }
12
```

```
13
14 Bookshop {
15   Publication (16%){
16     title : String;
17   }
18   Magazine extends Publication;
19   Book extends Publication {
20     author : Author;
21   }
22   Node Author {
23      name : String;
24   }
25 }
```

Fig. 8. Syntax template allowing inheritance and new clabjects (left), and its use (right)

5.2 Refining the Syntax of Domain-Specific Modelling Languages

Even if the DSMM language defines a syntax for the instances of the created domain-specific meta-models, the builder of a particular domain-specific meta-model may wish to design a special concrete textual syntax for some of the instantiated clabjects or for the linguistic extensions (see Fig. 5).

For example, we can design a template especially for `Nails`, as Fig. 9 shows. This template would be defined by the builder of the stationer's domain-specific meta-model at meta-level 1, and attached to it. The template refines the default one defined for products, so that the instances of `Nail` can be defined using this more specialised syntax (in addition to the default one). Hence, we can write `n1(0.1€, 0.1, 0.1)`, in addition to `Nail n1(0.1€ caliber=0.1 length=0.1)`. In order to disable the instantiation of nails using the latter, more general syntax, we should add the modifier `overwrite` to template `TNail`.

```
1 template@1 TNail for Node Nail:
2   "id '(' #price '€', #caliber, #length ')' "
```

Fig. 9. Refining the syntax template for nails at level 1

To conclude, as an implementation note, our template language for specifying concrete syntaxes has been implemented using a meta-model, whereas its concrete syntax has been specified with itself through bootstrapping. The parser generation relies on ANTLR [14]. Moreover, METADEPTH includes a registry of parsers and automatically selects the appropriate parser according to the file extension of the model to be loaded.

6 Model Management for DSMM Languages

To integrate DSMM languages in MDE, we need to provide suitable model management languages able to deal with multiple meta-levels. In METADEPTH we have adapted the Epsilon family of model management languages[1] to work in a multi-level setting. Hence, we can define model manipulation operations and constraints for DSMM languages using the Epsilon Object Language (EOL), code generators working at several meta-levels using the Epsilon Generation Language (EGL), and model-to-model transformations spanning several meta-levels using the Epsilon Transformation Language (ETL). As the working scheme and challenges are similar in all cases, we will illustrate our solution only in the context of model-to-model transformations.

As an example, assume we want to generate a graphical user interface that allows the customers of an e-commerce system to select the products (at level 0) they want to buy. For this purpose, we need to transform the products to a model representation of the graphical user interface, from which we can generate code for different platforms like Java Swing or HTML. Based on this example, in the following we illustrate the four typical transformation scenarios in a multi-level setting: *deep transformations*, *co-transformations*, *refining transformations* and *reflective and linguistic transformations*.

Deep Transformations. Oftentimes, a transformation needs to be defined using the meta-model of the DSMM language, and applied to the instances of the DSMLs built with it (i.e., at the bottom level). This scenario is depicted to the right of Fig. 10. In this case, the transformation definition needs to use indirect types because the direct types at level 1 are unknown when the transformation is defined. For example, if we want to generate a graphical user interface for any model of products at level 0, we would like to define the transformation only once at meta-level 2 together with the DSMM language definition. The left of Fig. 10 shows the ETL deep transformation to achieve this goal, which will be executed on indirect instances of the `Ecommerce` model. Rule `Product2CheckButton` creates a `CheckButton` for each indirect instance of `Product` (lines 3–8). The rule is annotated with the top-level meta-model needed by the transformation (line 1), and the level at which the transformation is to be executed (line 2). The *post* block (lines 10–15), which is executed when the transformation finishes, creates the *GroupBox* for the checkbuttons and the container *Window*.

Co-transformations. In this kind of transformations, a model and its meta-model need to be transformed at the same time, as the right of Fig. 11 illustrates. Here, the same transformation has to deal with direct and indirect instances of the clabjects in the meta-model of the DSMM language; therefore, a mechanism is needed to select the level at which the rules will be applied.

As an example, we may wish to generate a menu for each product type defined at level 1, and checkbuttons for each product instance at level 0. For this purpose, we can use the transformation in Fig. 11. Line 1 imports the previous

[1] See `http://www.eclipse.org/epsilon/`

```
1  @metamodel(name=Ecommerce,file=Ecommerce)
2  @model(potency=0)
3  rule Product2CheckButton
4     transform pr : Source!Product
5     to cb : Target!CheckButton {
6        cb.name := pr.name()+'_cbutton';
7        cb.text := pr.name()+'('+pr.price+')';
8     }
9
10 post {
11    var wd: Target!Window := new Target!Window();
12    var gb: Target!GroupBox := new Target!GroupBox();
13    wd.children := gb;
14    gb.children.addAll(Target!CheckButton.all());
15 }
```

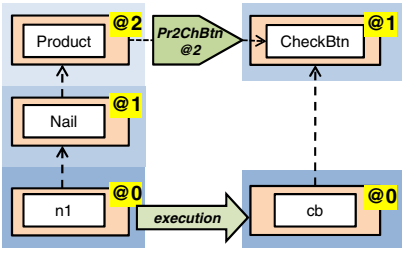


Fig. 10. Deep transformation example (left) and scheme (right)

transformation which transforms the products at level 0. Then, rule `ProductType2Menu` is executed for each `Product` at level 1. The level at which the rule is executed is specified by the model alias, before the '!' symbol (see line 4). Hence, we use `Level0` for a model with potency 0 and `Level1` for a model with potency 1. We can also use the alias `Source` to refer to the source model regardless its potency. This is the alias used in the listing of Fig. 10, where the annotation in line 2 forces the execution of the transformation on models with potency 0. Hence, our framework implicitly makes available all (meta-)*-models of the context model for the transformation.

Refining Transformations. Sometimes, a deep transformation needs to be refined for particular instances defined at level 1. This situation is depicted to the right of Fig. 12. For example, if we decide to transform the instances of `Nail` in a different way to consider the specific attributes that we added to it (`caliber` and `length`), we need to refine the transformation rule defined for `Products` in Fig. 10. The refined rule is shown in Fig. 12. The rule extends `Product2CheckButton`, but it is refined for type `Nail`. To support this kind of transformations, we adapted ETL to allow extending a rule if the child rule transforms a direct or indirect instance of the clabject type transformed by the parent rule. The child rule will be applied whenever is possible, executing the body of the rules of both parent and child. In our example, rule `Nail2CheckButton` will

```
1  import 'file:///Prod2GUI.etl';
2
3  rule ProductType2Menu
4     transform pr : Level1!Product
5     to mn : Target!Menu {
6        mn.name := pr.name()+'_menu';
7        mn.text := pr.name()+'('+pr.VAT+')';
8     }
```

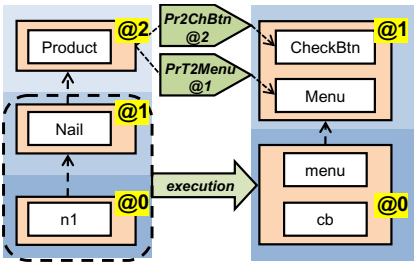


Fig. 11. Co-transformation example (left) and scheme (right)

```
1  import 'file:///Prod2GUIDeep.etl';
2
3  @metamodel(name=Ecommerce,file=Ecommerce.mdepth
       )
4  @model(potency=0)
5  rule Nail2CheckButton
6    transform pr : Source!Nail
7    to mn : Target!CheckButton
8    extends Product2CheckButton {
9      mn.name := pr.name()+'_check_nail';
10     mn.text := pr.name()+'('+pr.price+',
11       caliber='+pr.caliber+', length='+pr.length+')';
12   }
```

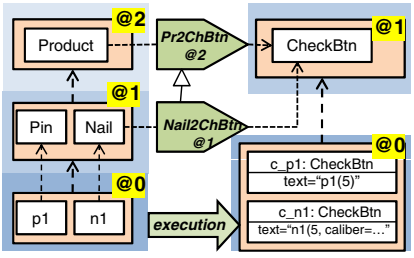


Fig. 12. Refining transformation example (left) and scheme (right)

be executed for instances of `Nail`, whereas rule `Product2CheckButton` will be executed for indirect instances of `Product` that are not instances of `Nail`.

Reflective and Linguistic Transformations. When defining a deep transformation, we may want to account for the linguistic extensions that can be performed at level 1. For this purpose, the transformation language needs reflective capabilities to access any new declared field, and it has to be possible to perform queries using linguistic types (i.e., `Node`, `Edge` and `Model`). The combination of these two capabilities enables the construction of generic transformations, applicable at any meta-level, and to elements of any ontological type. The working scheme of this kind of transformations is shown to the right of Fig. 13.

```
1  rule Product2CheckButton
2    transform pr : Level0!Product
3    to cb : Target!CheckButton {
4      cb.name := pr.name()+'_cbutton';
5      cb.text := pr.name()+'( price='+pr.price+' ';
6      for (f in pr.newFields())
7          cb.text := cb.text+f.name()+'= '+
8                     f.getValue()+' ';
9      cb.text := cb.text+')';
10   }
11
12 rule Node2Label
13     transform pr : Level0!Node
14     to cb : Target!Label {
15     guard: not pr.isKindOf(Level0!Product)
16     cb.name := pr.name()+'_label';
17     cb.text := pr.name()+'(';
18     for (f in pr.fields())
19         cb.text := cb.text+f.name()+'= '+
20                    f.getValue()+' ';
21     cb.text := cb.text+')';
22   }
```

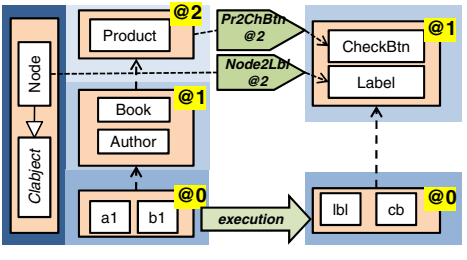


Fig. 13. Linguistic transformation example (left) and scheme (right)

The listing in Fig. 13 shows a transformation with one reflective rule and another one defined on a linguistic type. Rule `Product2CheckButton` is reflective. It gets executed for each indirect instance of `Product` at level 0, creating a `CheckButton`. The rule takes into account that `Product` instances at level 1

may have been extended with new attributes. Thus, the rule iterates on the new attributes in line 6 (returned by `newFields`), concatenating their name and value. Technically, this reflection is possible because ETL is also reflective, being able to call transparently methods of the metaDepth API.

In its turn, rule `Node2Label` uses linguistic typing, being applicable to all `Node` instances (all elements) of potency 0 which are not indirect instances of `Product` (forbidden by the guard in line 15). In this way, if we apply this transformation to the `Amazon` model in Fig. 1, we obtain one `CheckButton` (the transformation of the `KL` book by rule `Product2CheckButton`) and one `Label` (the transformation of the `WS` author by `Node2Label`).

In the presented transformation examples, the target language has two meta-levels. We also allow DSMM languages as target, and currently we only support rules specifying the creation of direct instances of clabjects. One may consider *abstract* rules specifying the creation of indirect instances, which would need to be refined at level 1 stating which clabject to instantiate. This is left for future work.

7 Discussion and Future Work

In this paper, we have presented our approach to define DSMM languages supporting the flexible definition of a textual concrete syntax, a fine control of the exposed meta-modelling facilities, and integration in MDE projects by making available multi-level model management languages.

We also discussed the typical transformation scenarios in a multi-level setting (deep transformations, co-transformations, refining transformations and linguistic/reflective transformations) and illustrated their support using ETL. These scenarios apply to other model management languages and tasks as well. In particular, they apply to the definition of textual syntaxes: at the top-level, we can define syntactic templates for level 0 models (similar to deep transformations), or for both level 0 and level 1 models (similar to co-transformations); we can add refining templates at level 1 (like in refining transformations); and we can define templates dealing with linguistic extensions (as in linguistic transformations). Each model management language needs to provide appropriate constructs to deal with each scenario, namely: the ability to select the meta-level at which a certain operation is to be applied (e.g., potencies for rules and templates), the ability to select clabjects of specific meta-levels (e.g., aliases `Level0` and `Level1` in rules), the possibility to obtain indirect instances of clabjects (transparently in our case), to access clabjects by their linguistic type (e.g., `Node`) and to reflectively access linguistic extensions (e.g., method `newFields`).

We are currently using metaDepth to define DSMM languages in different domains: component-based systems, web engineering and mobile devices. We are also exploring the definition of visual syntaxes for DSMM languages, and extending the integration of the tool with multi-level meta-modelling languages.

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. Aschauer, T., Dauenhauer, G., Pree, W.: Representation and Traversal of Large Clabject Models. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 17–31. Springer, Heidelberg (2009)
2. Atkinson, C., Gutheil, M., Kennel, B.: A flexible infrastructure for multilevel language engineering. IEEE Trans. Soft. Eng. 35(6), 742–755 (2009)
3. Atkinson, C., Kühne, T.: Rearchitecting the UML infrastructure. ACM Trans. Model. Comput. Simul. 12(4), 290–321 (2002)
4. Atkinson, C., Kühne, T.: Model-driven development: A metamodeling foundation. IEEE Software 20(5), 36–41 (2003)
5. Atkinson, C., Kühne, T.: Reducing accidental complexity in domain models. Software and System Modeling 7(3), 345–359 (2008)
6. 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)
7. Espinazo-Pagán, J., Menárguez, M., García-Molina, J.: Metamodel Syntactic Sheets: An Approach for Defining Textual Concrete Syntaxes. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 185–199. Springer, Heidelberg (2008)
8. Herrmannsdörfer, M., Hummel, B.: Library concepts for model reuse. Electron. Notes Theor. Comput. Sci. 253, 121–134 (2010)
9. Jablonski, S., Volz, B., Dornstauder, S.: A meta modeling framework for domain specific process management. In: COMPSAC 2008, pp. 1011–1016 (2008)
10. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: GPCE. ACM (2006)
11. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon Object Language (EOL). In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 128–142. Springer, Heidelberg (2006)
12. Kühne, T., Schreiber, D.: Can programming be liberated from the two-level style? – Multi-level programming with DeepJava. In: OOPSLA 2007, pp. 229–244 (2007)
13. Langer, P., Wieland, K., Wimmer, M., Cabot, J.: From UML Profiles to EMF Profiles and Beyond. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 52–67. Springer, Heidelberg (2011)
14. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. Pragmatic Bookshelf (2007), http://www.antlr.org/
15. xText, http://xtext.org
16. Zschaler, S., Kolovos, D.S., Drivalos, N., Paige, R.F., Rashid, A.: Domain-Specific Metamodelling Languages for Software Language Engineering. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 334–353. Springer, Heidelberg (2010)

Metamodel Based Methodology for Dynamic Component Systems

Gabor Batori[1], Zoltan Theisz[2], and Domonkos Asztalos[1]

[1] Software Engineering Group, Ericsson Hungary Ltd.
{gabor.batori,domonkos.asztalos}@ericsson.com
[2] evopro Informatics and Automation Ltd.
zoltan.theisz@evopro.hu

Abstract. MBE solutions, including their corresponding MDA frameworks, cover many parts of industrial application development processes. Although model based development methodologies are in abundance, fully integrated, domain specific methodologies still find their niche in specialized application scenarios. In this paper, such an alternative methodology will be presented that targets reconfigurable networked systems executing on top of interconnected heterogeneous hardware nodes. The methodology covers the whole development cycle; it even utilizes a configuration model for component reconfigurability, and also involves a first-order logic based structural modeling language, Alloy, in the analysis of component deployment and reconfiguration. The methodology is supported by both a metamodel based tooling environment within GME and a robust distributed middleware platform over Erlang/OTP. Due to its special applicability, the methodology is limited in scope and scaling, though core parts have been successfully showcased in a sensor network demonstrator of the IST project RUNES.

1 Introduction

Dynamic component systems provide a versatile platform for creating autonomic distributed peer-to-peer applications in various industrial domains. A particularly challenging case of these domains is the area of intelligent sensor networks, that combine sensory and effectory facilities within their control loops. Due to the frequent reconfiguration of software components, this field of applicability is characterized by the inherent complexity of such environments; therefore creating an effective, high-quality software development methodology that seriously unburdens the day-to-day tasks of application developers is a rather ambitious endeavor. However, the Reconfigurable Ubiquitous Network Embedded Systems (RUNES) IST project [1] successfully addressed this challenge by providing a common distributed component-based platform architecture on top of heterogeneous networks of computational nodes. The RUNES middleware platform [2] is accompanied by a corresponding model-based software development methodology and tooling support. In effect, the approach and the implemented framework are based on well–known concepts of Model-Integrated Computing [3] and support a rapid application development environment in GME [4]. Nevertheless, the validation and eventual verification of the produced dynamic networked components turn out to be a rather ambitious endeavor, which requires detailed software engineering know-how. Hence, the original RUNES MBE methodology [5] had to be extended

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 275–286, 2012.

by some practical, formal logic based techniques in Alloy [6] in order to establish an overarching metamodel based methodology. This covers the whole development cycle, including formal scenario validation and better quality insurance for some MBE modeling tasks by eliminating non-trivial dynamic errors or failure situations that may frequently reoccur in the application design of reconfigurable component systems.

The paper is structured as follows: Section 2 provides a background on the technical domain of networked reconfigurable component systems, establishing this way the conceptual frame for the rest of the paper. In Section 3, the metamodel based development process is presented covering the whole life-cycle of component applications. Next, Section 4 describes, in relative details, the modeling assets used in the various stages of the development process, from scenario analysis up to its validation. Then, Section 5 mentions some of the case studies and Section 6 gives some insight onto the practical side of the methodology. Finally, in Section 7, the conclusions are provided.

2 Networked Reconfigurable Dynamic Component System

The architecture of the targeted networked reconfigurable dynamic component system consists of a reflective, reconfigurable middleware model of the component system and a corresponding Component Run-Time Kernel (CRTK) that provides the management APIs. The reflective components are linked together by their interfaces, they communicate via message sending and store their meta-data in a distributed database within the middleware. Each computational node incorporates an instance of the CRTK, which provides the basic middleware APIs of component management. These architectural concepts are mirrored by an effective reference implementation, called ErlCOM [7], which runs on Ericsson's Erlang/OTP distributed infrastructure [8] utilizing Mnesia [9] for the distributed database.

The component is the basic unit of the system that corresponds to an active actor-like process, it owns snippets of executable code and has a uniquely registered name in the middleware's global registry. The components are organized into caplet hierarchies the root of which is occupied by a capsule, which is the main process entity of the node. The caplets' main purpose is to provide supervisory facilities for the maintenance of robustness and longevity of the whole component system. The supervisory decisions are taken according to a set of predefined constraints stored within a particular component framework. The interaction between components is carried out via pure message passing that is managed by the bindings representing the behavioral policy of the communication channels. The bindings are also components with special communication properties. Message passing is synchronous; messages can be intercepted both before they enter the interfaces of the recipients and after the replies have left those same interfaces. The pre- and post-actions of the bindings constitute a list of additional transformations on individual messages. Bindings are created when a receptacle, that is, a required interface, of a particular component is to be bound to a provided interface of another component, provided that their compatibility has been checked and validated. Finally, both components and bindings possess explicit state information which is stored as metadata in the global repository within the distributed middleware deployed over the networked nodes.

3 Development Process

Professional software developments are always accompanied by corresponding development processes that safeguard industrial scale applicability of the chosen technology. Although there is a plethora of such MBE approaches, e.g. Rational Unified Process, the ambition level of our process design was influenced by the aim of being able to cover all stages of component based application development, including generative metamodeling technologies and scenario validation and verification. The overview of the process stages are depicted in Figure 1. The process is layered into five stages; namely, Scenario, Application Model, Platform Model, Code Repository and Running System. The arrows of the non-iterative part of the process, connecting together the artifacts of the various stages, are labeled by sequence numbers in accordance to their timing. Here, the stages are only briefly introduced, the assets these stages are operating on will be described in Section 4.

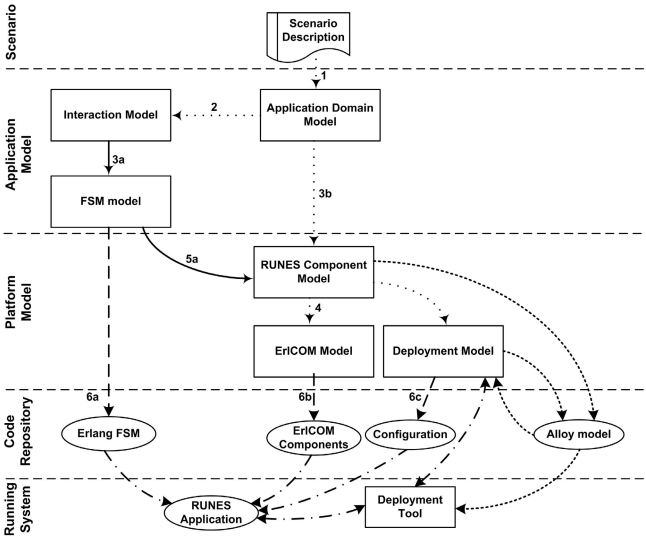


Fig. 1. MBE development process

The Scenario evaluates and finalizes a set of scenario descriptions by establishing the scope of the application domain. Based on our experience, real-life distributed applications usually involve intense interactions among application components, thus both structural and interaction modeling are of equally importance. Therefore, the Application Domain Model is created to cover the scenario in such a way that all use case details are taken adequately into account and all stakeholders' roles have been discovered.

The roles make up the basic elements of the interaction model, hence the dynamicity of the use cases must be translated into corresponding Message Sequence Charts (MSC). The Application Domain Model and the Interaction Model must be detailed enough so

that quality investigations could be carried out in order to check the feasibility of the design. Moreover, this stage involves many creative decisions, so both arrow 1 and 2 in Figure 1 are dotted, this way showing that the activity is mainly manual in nature.

The Interaction Model is transformed into a FSM Model and then a further translation maps it onto the RUNES Component Model. The solid arrow indicates that the translation is executed via non-trivial graph transformations as reported in [5]. The Application Domain Model usually requires creative refinements and only semi-automatically (dotted line) can be translated into a RUNES Component Model.

The Platform Model stage has been conceived to support total semantics elaboration, that is, the RUNES Component Model is extended by the semantics of the platform, the components and the FSMs. This step involves some manual coding in Erlang so that the total executable specification of the application can be fully established.

The final application model takes into consideration the distributed nature of the application; hence, a Deployment Model is also getting populated. It entirely specifies the total component allocation of the application over the available hardware nodes of the underlying network.

The Code Repository is the stage which copes with source code management. The code production is fully automated, which is indicated by dashed lines, and the resulted code snippets of the components are stored within the global repository of the middleware as modules ready to be loaded. The Deployment Model is translated into an initial run-time configuration which is deployed over the participating ErlCOM nodes. Any changes of the component configuration at run-time are managed by the Deployment Tool, which continuously updates the Deployment Model.

The Deployment Tool implements a kind of metamodel-driven component management, which generalizes policy-based network management in such a way that the information model used by the network management infrastructure mirrors the software assets of the component based system produced during the generative model translators. In effect, it establishes a soft real-time synchronization loop between the GME model repository and the running component application. In other words, the Deployment Tool, which is a protocol independent abstraction of GANA's Decision Making Element [10], first deploys the initial component configuration of the application then it constantly readapts the component configuration by listening to both application and middleware notifications and by observing all changes within the Deployment Model stored inside the model repository. Therefore, with the control logic properly established, the Deployment Tool is able to manage both re-active and pro-active component reconfigurations as reported in [5,11].

The Alloy based model verification step extends this standard operation of the Deployment Tool. It contains two additional model transformations; one that originates from the RUNES Component Model and another that takes a compatible RUNES Deployment Model and it turns them into a configuration scenario that can be verified within Alloy Analyzer [12]. The model transformations produce configuration scenarios, which include both the structural and the behavioral specifications of the application. However, only those parts of the FSM action semantics are kept from the total dynamic behavior that either directly relate to important control logic elements of the scenario or which belong to the operations provided by the underlying ErlCOM

middleware. These steps simplify, though, more precisely specify when and with which parameters the application invokes the operations of the CRTK. Hence, the validation of a particular scenario investigates mainly the evolution of the application from the point of view of its component reconfigurations that are allowed by the semantics of the ErlCOM middleware. The results of this verification step provide useful hints to the run-time autonomic control mechanisms which will either be embedded into the application or be defined as explicit rules of the policy engine. The validation and verification step is rather iterative in nature, which is luckily well supported by Alloy Analyzer.

4 Modeling Assets

4.1 Interaction Model

Large-scale networked systems can be efficiently represented by a large number of interacting services. By combining all those services an entity is getting involved in the complete behavior specification for that particular entity can be established. Hence, our service concept is effectively based on the interaction patterns between cooperating entities. The notion of a role describes the contribution of an entity within a given interaction pattern. In our methodology we follow a particular service oriented approach [13], which maps service specifications onto a set of interconnected components, each of them having an internal Finite State Machine (FSM), and a corresponding pool of abstract communication channels. This techniques also incorporates an effective state machine synthesis algorithms so that scenarios can be easily turned into a corresponding set of FSMs, that fully specify the intended dynamic behavior of the specified system (see Figure 2). The state machine generation is carried out automatically and relies on two types of MSCs, the basic MSCs and the high level MSCs (HMSC). The output of the transformation is one FSM per role within the domain model; that is, the FSM implementing the respective role's contribution to the services it is associated with. The FSMs are incorporated in stage Application Model.

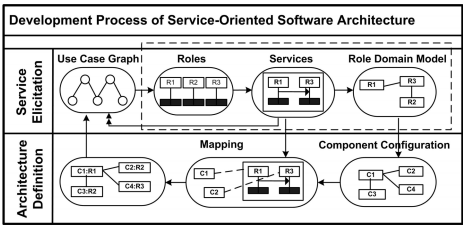


Fig. 2. Service-based interaction model

4.2 Structural Model

The core part the component metamodel is illustrated in Figure 3. The metamodel mirrors the constituents of the networked reconfigurable dynamic component systems (see

Section 2). Hence, Components, short for ErlCOMComponent, are encapsulated units of functionality and deployment, which interact with each other only via interfaces and receptacles. Interfaces are defined by a list of related operation signatures and associated data types. Components can also provide multiple interfaces, this way embodying a clear separation of concerns (e.g. between base functionality and component management). Capsules and Caplets are platform containers providing access to the run-time APIs. Bindings ensure consistent connection setup between a compatible Interface and a Receptacle. The compatibility checks take into account the list of Operations describing the service specification of a particular Interface or Receptacle. The Operations are further specified by their signatures containing the list of incoming parameters and the outgoing ReturnValue if it exists. The component model itself is complemented by two other architecture elements: component frameworks and reflective extensions. ComponentFrameworks (CF) are groupings of Components with embedded constraints that guarantee that only "meaningful" component configurations can be built. All entities of the metamodel (Component, Capsule, Interface, Receptacle, Binding, ComponentFramework) can store arbitrary <key,value> attributes, which describe the reflective behavior of the ErlCOM middleware. Component interactions can be intercepted at the Bindings by pre- and post-actions to enable additional processing on the level of individual messages. These last two features of the middleware are specified in a different part of the component metamodel. The structural model of a component application is designed in stage Platform Model.

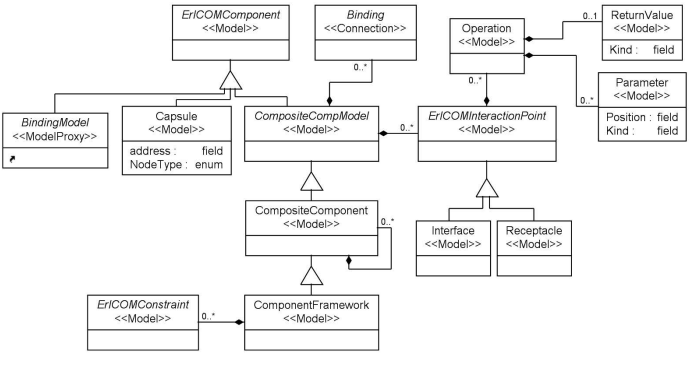


Fig. 3. Structural metamodel

4.3 Behavior Model

The component behavior description is formalized in an abstract model of action semantics (see Figure 4). This Behavior Model is rather generic, though it provides a selection attribute for specifying the modeled behavior within the selected implementation language(s). The entities that can contain behavior descriptions are the Interface and the Component. A fully specified component model is later translated into the chosen target implementation language(s) by language optimized model interpreter(s). By

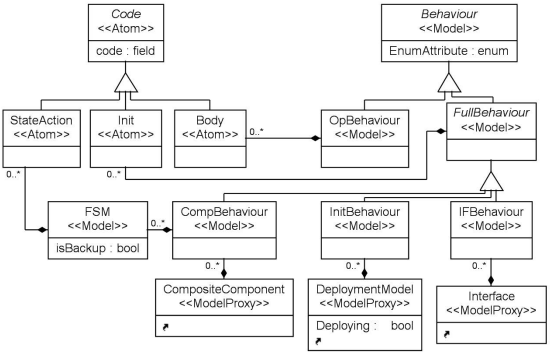


Fig. 4. Behavior metamodel

this means, Components can be operationally described in various programming languages; however, they must rely on the same modeling framework. A language specific model interpreter processes only those parts of a component model which contain relevant information for the desired target language environment. Therefore, the metamodel embodies various code snippets; the snippets are later woven together into executable component implementations by the corresponding model interpreters. Although this facility is rather versatile, over the ErlCOM CRTK only Erlang has been used as the language of executable specification of dynamic behavior. The core building blocks of such a specification are as follows:

- Init - Initialization code for a component, an interface or the system.
- Body - Executable specification of the operation of an interface. The signature of the operation is defined in the model and automatically generated by the interpreter.
- StateAction - Specifies the action semantics inside an FSM state which is automatically injected into the corresponding connection point within the generated FSM.

4.4 Deployment Model

The complete synthesized platform specific application model contains both the structural configuration and the behavioral semantics of all the constituent components, including their interconnecting bindings and component framework constraints. That model represents the functional view of the application; however, it neither specifies how the application is deployed on the available networked nodes nor how it has to start. Therefore, the deployment configuration (see Figure 5) must be modeled, too. The deployed component configuration, which contains the complete synthesized platform specific application model and the initial configuration of the components, is called in our methodology as the total synthesized platform specific distributed application model.

From the point of view of out model based framework, one of the most important elements of the deployment infrastructure is the Deployment Tool. The schematics of the Deployment Tool based reconfiguration is shown in Figure 6. The Deployment Tool

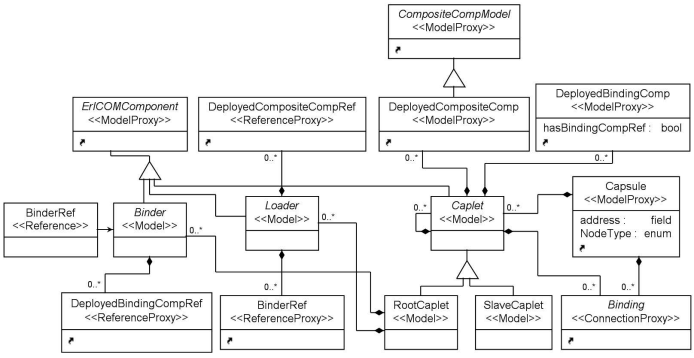


Fig. 5. Deployment metamodel

analyzes the initial component configuration of the total synthesized platform specific application model and creates the needed ErlCOM run-time elements by invoking the corresponding operations of the ErlCOM API. (The complete ErlCOM API semantics, including dynamic behavior of the middleware, has been published in [7]) After the initial deployment has been completed the application starts running and the ErlCOM CRTK continuously monitors all component reconfiguration and in case of observable component changes events are sent containing descriptive notifications to the Deployment Tool. The Deployment Tool keeps track of the actual component configuration of the running system by updating the total synthesized platform specific RUNES application model. Deployment Tool plug-ins can also execute policy based rules either re-actively or pro-actively. Any corrective changes on the modeled component configuration of the component application will be reflected by the run-time deployment.

4.5 Validation and Verification Model

Distributed reconfigurable component systems are complex by nature, hence, the importance of formal description techniques in system design is well known. In our methodology, we have based the formal model analysis on the usage of Alloy [6], a first order logic based description language, that is powered by SAT solvers. This formal description technique has been successfully used for modelling various complex systems in a wide range of application domains. It has been applied in [14] for the analysis of some critical correctness properties that should be satisfied by any secure multicast protocols. The idea of applying Alloy for component based system analysis was also suggested by Warren et al. [15], where OpenRec's Alloy model is investigated. This Alloy model served as a conceptual basis for our own Alloy component model, which specifies all the core items of the ErlCOM metamodel and middleware semantics, including structural elements, the precise dynamics of finite state machines and the major concepts of the deployment metamodel, in their first order logic based semantics. The precise definition of this Alloy model,that enables the detailed analysis of dynamic component behavior has been published in [16]. There are similar techniques that aim to identify various types of dynamic system reconfigurations [17]; however, our approach is a better fit

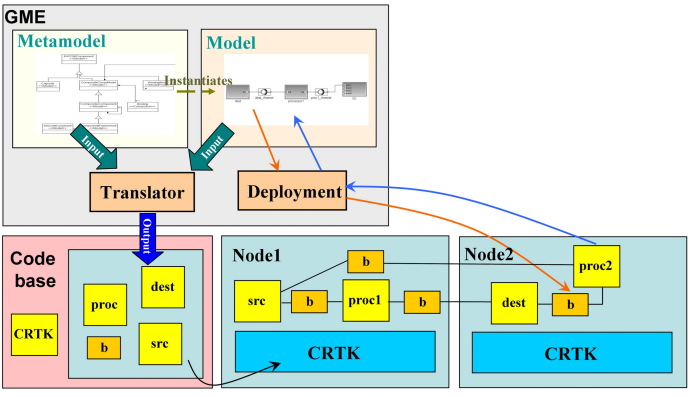


Fig. 6. Deployment Tool based component reconfiguration

for the networked reconfigurable dynamic component systems. Nevertheless, all these attempts provide a rather good categorization of various problems and corresponding solutions related to dynamic software evolution. Considering the tooling support, Aydal et al. [18] found Alloy Analyzer one of the best analysis tools for state-based modeling languages, which has been a serious concern in our selection of Alloy for scenario validation and verification purposes.

The graphical visualization of the structural model of a scenario example in Alloy Analyzer is depicted in Figure 7. It shows a model that represents a snapshot of a dynamically evolving component configuration of a sensor network scenario example taken from the RUNES project [1]. The components (black hexagons) have been deployed over a cross shaped capsule (gray pentagons) topology. The connections among the capsules of this topology are indicated by green arrows. The internal resources, here the maximum number of deployed components/bindings, of the capsules are limited in capacity. The concrete mapping of the components and bindings (white rhombuses) onto the capsules, at a particular instance of time, is visualized by the brown and red arrows, respectively.

Regarding the dynamic behavior of the FSMs, Figure 8 shows the state machine of the NetworkDriver component as an example from the same scenario. The initial status of the FSM is given by the start state (black ellipse) and the initial transition (white rectangle). The other states are represented by gray colored ellipses, while the transitions are shown via red rectangles.

Due to page limitation, this paper cannot detail on a full analysis example (simplified example is reported in [16]), so the validation/verification session is only summarized. Basically, such a session is carried out within Alloy Analyzer and it is driven by the Alloy model of the scenario under investigation. Validation only generates a set of potential runs of the scenario, while verification also injects logical properties into the Alloy specification of the component application before it looks for counter-examples, and locates them if found. In general, the approach helps to analyze configuration sequences so that they both comply with some application constraints and avoid non-trivial pitfalls. The result of these analyses is fed back to the control logic of the Deployment Tool (see Section 4.4).

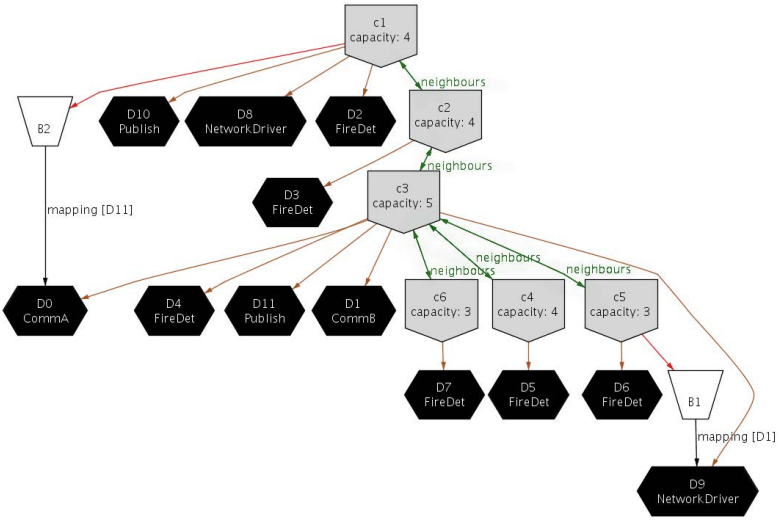


Fig. 7. Structural analysis model

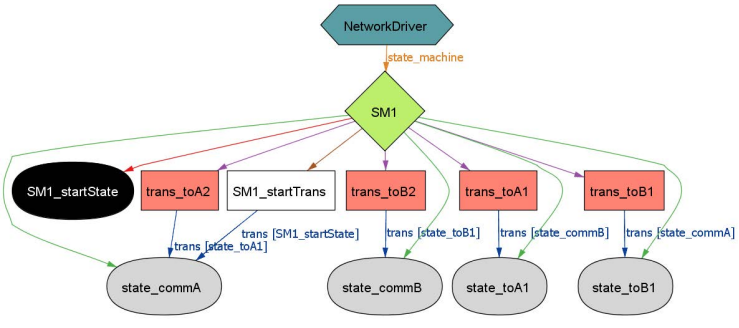


Fig. 8. Dynamic analysis model

5 Case Studies

There had been many prototypical case studies during the frame of the RUNES IST [1] project. The aim of the studies had been to successively evaluate the methodology and its applied modeling and platform technology. The efforts of this continuous evaluation resulted in the process described in Section 3. The final RUNES demonstrator [19] based around a scaled down model of the road tunnel scenario consisting of ten TMote Sky motes and an embedded Gateway hardware, which connected the motes to a local visualization infrastructure, had been mostly developed following this process and relying on the available modeling assets. The Alloy based verification has been an internal research activity aimed to exploit the results of the RUNES project and directly targeted scenario verification and validation in Alloy. One of such a scenario validation use case

has been reported publicly in [16]. Finally, this validation step has been integrated into our methodology.

6 Evaluation

Every methodology has its limits, our approach not being an exception. The research challenges of the RUNES project focused on the effective component based software generation in the domains of embedded systems and sensor networks. Therefore, the resulting methodology and tooling had to satisfy these challenges. The final demonstrator had been developed following the described process, however, not all the code assets could be automatically generated for each hardware nodes. The limitation was due to the non-existence of ErlCOM CRTK on the TMote Sky motes. Although semantically correct FSMs had been generated for the ErlCOM simulation of the mote CRTK, these code snippets had to be manually ported to the target platform. Though, we regard it as a platform limitation that may change in the future. Concerning our verification and validation efforts, we have hit the same practical barriers that are all too known in the model checking community. Without applying adequate abstraction scenario use cases are impossible to be analyzed verbatim from the development model. Our practical approach relied on case-by-case scenario selection and expert validation via Alloy Analyzer, however much of the Alloy description can be produced from the scenario models.

7 Conclusion

This paper disseminated a multi-stage, metamodel based software development methodology for the domain of networked reconfigurable dynamic component systems. After the brief introduction of the underlying technology, the steps of the development process have been explained, then, the various model assets have been described one-by-one. Although the production part of the development methodology had been thoroughly tested within the RUNES IST project, the validation/verification part is still under further investigation. The results are promising, but we are fully aware of the limitations of first order logical based verification tools and the related scalability issues. Nevertheless, we firmly believe that this proposed MBE methodology well served the original challenges of our motivation.

References

1. Arzén, K.-E., Bicchi, A., Dini, G., Hailes, S., Johansson, K.H., Lygeros, J., Tzes, A.: A component-based approach to the design of networked control systems. European Journal of Control (2007)
2. Costa, P., Coulson, G., Mascolo, C., Picco, G.P., Zachariadis, S.: The RUNES Middleware: A reconfigurable component-based approach to networked embedded systems. In: Proc. of the 16th Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC 2005), Berlin, Germany (September 2005)

3. Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty, T.: Model-integrated development of embedded software. Proceedings of the IEEE 91, 145–164 (2003)
4. Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., Thomason, C., Nordstrom, G., Sprinkle, J., Volgyesi, P.: The generic modeling environment. In: Proceedings of WISP 2001, Budapest, Hungary, pp. 255–277 (May 2001)
5. Batori, G., Theisz, Z., Asztalos, D.: Domain Specific Modeling Methodology for Reconfigurable Networked Systems. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 316–330. Springer, Heidelberg (2007)
6. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, London (2006)
7. Batori, G., Theisz, Z., Asztalos, D.: Robust reconfigurable erlang component system. In: Erlang User Conference, Stockholm, Sweden (2005)
8. Armstrong, J.: Making reliable distributed systems in the presence of software errors. SICS Dissertation Series 34 (2003)
9. Mattsson, H., Nilsson, H., Wikström, C.: Mnesia – A Distributed Robust DBMS for Telecommunications Applications. In: Gupta, G. (ed.) PADL 1999. LNCS, vol. 1551, pp. 152–163. Springer, Heidelberg (1999)
10. Prakash, A., Theisz, Z., Chaparadza, R.: Formal Methods for Modeling, Refining and Verifying Autonomic Components of Computer Networks. In: Gavrilova, M.L., Tan, C.J.K., Phan, C.-V. (eds.) Transactions on Computational Science XV. LNCS, vol. 7050, pp. 1–48. Springer, Heidelberg (2012)
11. Batori, G., Theisz, Z., Asztalos, D.: Configuration aware distributed system design in erlang. In: Erlang User Conference, Stockholm, Sweden (2006)
12. Jackson, D.: Alloy analyzer (2008), http://alloy.mit.edu/
13. Krüger, I.H., Mathew, R.: Component Synthesis from Service Specifications. In: Leue, S., Systä, T.J. (eds.) Scenarios. LNCS, vol. 3466, pp. 255–277. Springer, Heidelberg (2005)
14. Taghdiri, M., Jackson, D.: A Lightweight Formal Analysis of a Multicast Key Management Scheme. In: König, H., Heiner, M., Wolisz, A. (eds.) FORTE 2003. LNCS, vol. 2767, pp. 240–256. Springer, Heidelberg (2003)
15. Warren, I., Sun, J., Krishnamohan, S., Weerasinghe, T.: An automated formal approach to managing dynamic reconfiguration. In: 21st IEEE International Conference on Automated Software Engineering (ASE 2006), Tokyo, Japan, pp. 37–46 (September 2006)
16. Theisz, Z., Batori, G., Asztalos, D.: Formal logic based configuration modeling and verification for dynamic component systems. In: MOPAS 2011 (2011)
17. Walsh, D., Bordeleau, F., Selic, B.: A Domain Model for Dynamic System Reconfiguration. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 553–567. Springer, Heidelberg (2005)
18. Aydal, E.G., Utting, M., Woodcock, J.: A comparison of state-based modelling tools for model validation. In: Tools 2008 (June 2008)
19. 5th RUNES Newsletter, p. 6 (2007), http://www.socrades.eu/Documents/objects/file1201161327.23

Bidirectional Model Transformation with Precedence Triple Graph Grammars

Marius Lauder*, Anthony Anjorin*, Gergely Varró**, and Andy Schürr

Technische Universität Darmstadt, Real-Time Systems Lab,
Merckstr. 25, 64283 Darmstadt, Germany
name.surname@es.tu-darmstadt.de

Abstract. Triple Graph Grammars (TGGs) are a rule-based technique with a formal background for specifying bidirectional model transformation. In practical scenarios, the unidirectional rules needed for the forward and backward transformations are automatically derived from the TGG rules in the specification, and the overall transformation process is governed by a control algorithm. Current implementations either have a worst case exponential runtime complexity, based on the number of elements to be processed, or pose such strong restrictions on the class of supported TGGs that practical real-world applications become infeasible. This paper, therefore, introduces a new class of TGGs together with a control algorithm that drops a number of practice-relevant restrictions on TGG rules and still has a polynomial runtime complexity.

Keywords: triple graph grammars, control algorithm of unidirectional transformations, node precedence analysis, rule dependency analysis.

1 Introduction

The paradigm of Model-Driven Engineering (MDE) has established itself as a promising means of coping with the increasing complexity of modern software systems and, in this context, *model transformation* plays a central role [3]. As industrial applications require reliability and efficiency, the need for formal frameworks that guarantee useful properties of model transformation arises. This is especially the case for *bidirectional* model transformation, where defining a precise semantics for the automatic manipulation and synchronization of models with a corresponding efficient tool support is quite challenging [4]. Amongst the numerous bidirectional model transformation approaches surveyed in [18], the concept of *Triple Graph Grammars (TGGs)* features not only solid formal foundations [5,12] but also various tool implementations [7,11,12].

TGGs [16] provide a declarative, rule-based means of specifying the consistency of source and target models in their respective domains, and tracking

* Supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at TU Darmstadt.
** Supported by the Postdoctoral Fellowship of the Alexander von Humboldt Foundation and associated with the Center for Advanced Security Research Darmstadt.

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 287–302, 2012.

inter-domain relationships between model elements explicitly by automatically maintaining a correspondence model. Although TGGs describe how *triples* consisting of source, correspondence, and target models are simultaneously derived, most practical software engineering scenarios require that source or target models already exist and that the models in the correspondence and the opposite domain be consistently constructed by a unidirectional forward or backward transformation. As a consequence, TGG tools that support bidirectional model transformation (i) rely on unidirectional forward and backward operational rules, automatically derived from a single TGG specification, as basic transformation steps, and (ii) use an algorithm that controls which rule is to be applied on which part of the input graph. As a TGG rule in the specification might require *context elements* created by another TGG rule, the control algorithm must consider these *precedences/dependencies* at runtime when (a) determining the order in which graph nodes can be processed, and (b) selecting the rule to be applied.

In this paper, we introduce a *node precedence analysis* to provide a global view on the dependencies in the source graph and to guide the transformation process. Additionally, we combine the node precedence analysis with *a rule dependency analysis* to support the control algorithm in determining the node processing order and selecting the next applicable rule. This approach can now exploit global dependency information, and perform an iterative, top-down resolution which is more expressive (can handle a larger class of TGGs) and fits better into future incremental scenarios. Finally, we prove that the improved control algorithm is still correct, complete, and polynomial.

Section 2 introduces fundamental definitions using our running example while Sect. 3 discusses existing TGG batch algorithms. Sect. 4 presents our rule dependency and node precedence analysis, used by the TGG batch algorithm presented in Sect. 5. Finally, Sect. 6 gives a broader overview of related *bidirectional* approaches and Sect. 7 concludes with a summary and future work.

2 Fundamentals and Running Example

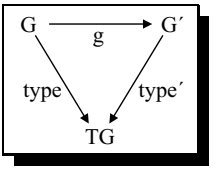
In this section, all concepts required to formalize and present our contribution are introduced and explained using our running example.

2.1 Type Graphs, Typed Graphs and Triples

We introduce the concept of a *graphs*, and formalize *models* as *typed graphs*.

Definition 1 (Graph and Graph Morphism). *A* graph $G = (V, E, s, t)$ *consists of finite sets* V *of nodes, and* E *of edges, and two functions* $s, t : E \to V$ *that assign each edge source and target nodes. A* graph morphism $h : G \to G'$, *with* $G' = (V', E', s', t')$, *is a pair of functions* $h := (h_V, h_E)$ *where* $h_V : V \to V'$, $h_E : E \to E'$ *and* $\forall e \in E : h_V(s(e)) = s'(h_E(e)) \ \wedge \ h_V(t(e)) = t'(h_E(e))$.

Definition 2 (Typed Graph and Typed Graph Morphisms).
A type graph *is a graph* $TG = (V_{TG}, E_{TG}, s_{TG}, t_{TG})$.
A typed graph $(G, type)$ *consists of a graph* G *together with a graph morphism* type*:* $G \to TG$.
Given typed graphs $(G, type)$ *and* $(G', type')$, $g : G \to G'$ *is a* typed graph morphism *iff the diagram commutes.*



These concepts can be lifted in a straightforward manner to *triples* of connected graphs denoted as $G = G_S \xleftarrow{h_S} G_C \xrightarrow{h_T} G_T$ as shown by [6,12]. In the following, we work with *typed graph triples* and corresponding morphisms.

Example. Our running example specifies the integration of *company structures* and corresponding *IT structures*. The *TGG schema* (Fig. 1) is the type graph triple for our running example. The *source domain* is described by a type graph for company structures: A `Company` consists of a `CEO`, `Employees` and `Admins`. In the *target domain*, an IT structure (`IT`) provides `PCs` and `Laptops` in `Networks` controlled by a `Router`. The *correspondence domain* specifies valid links between elements in the different domains.

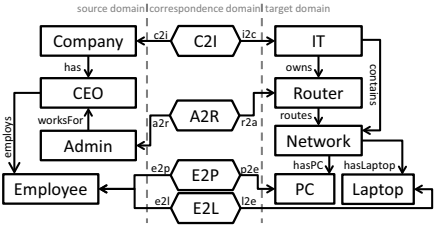


Fig. 1. TGG Schema for the integration of a company with its IT structure

A schema conform (typed graph) triple is depicted in Fig. 2. The company `ES` has a CEO named `Andy` for whom administrator `Ingo` works. Additionally, `Andy` employs `Tony` and `Marius`. The corresponding IT structure `ES-IT` consists of a router `WP53` for the network `ES-LAN` with a PC `PC65` and a laptop `X200`.

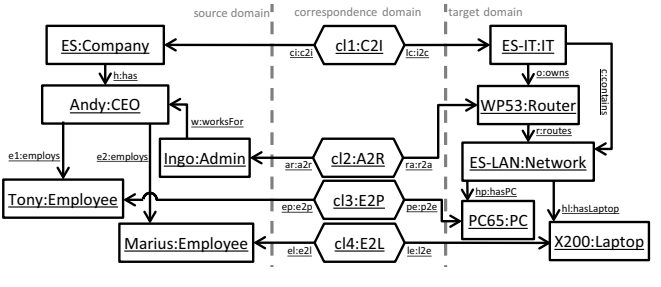
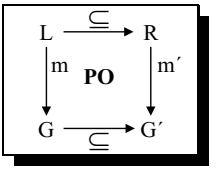


Fig. 2. A TGG schema conform triple

2.2 Triple Graph Grammars and Rules

The simultaneous evolution of typed graph triples such as our example triple (Fig. 2) can be described by a *triple graph grammar* consisting of *transformation rules*. This is formalized in the following definitions.

Definition 3 (Graph Triple Rewriting for Monotonic Creating Rules). *A* monotonic creating rule $r := (L, R)$, *is a pair of typed graph triples such that* $L \subseteq R$. *A rule* r rewrites *(via adding elements) a graph triple* G *into a graph triple* G' *via a match* $m : L \to G$, *denoted as* $G \overset{r@m}{\rightsquigarrow} G'$, *iff* $m' : R \to G'$ *is defined by building the* pushout G' *as denoted in the diagram.*



Elements in L denote the precondition of a rule and are referred to as *context elements*, while elements in $R \setminus L$ are referred to as *created elements*.

Definition 4 (Triple Graph Grammar). *A* triple graph grammar $TGG := (TG, \mathcal{R})$ *consists of a type graph triple* TG *and a finite set* $\mathcal{R}$ *of monotonic creating rules. The generated language (*$G_\emptyset$ *denotes the empty graph triple) is* $\mathcal{L}(TGG) := \{G \mid \exists\, r_1, r_2, \ldots, r_n \in \mathcal{R} :\ G_\emptyset \overset{r_1@m_1}{\rightsquigarrow} G_1 \overset{r_2@m_2}{\rightsquigarrow} \ldots \overset{r_n@m_n}{\rightsquigarrow} G_n = G\}$.

Example. The rules depicted in Fig. 3 build up an integrated company and IT structure simultaneously. Rule (a) creates the root elements of the models (a `Company` with a `CEO` and a corresponding `IT`), while Rule (b) appends additional elements (an `Admin` and a corresponding `Router` with the controlled `Network`). Rules (c) and (d) extend the models with an `Employee`, who can choose a `PC` or a `Laptop`. We use a concise notation by merging L and R of a rule, depicting context elements in black without any markup, and created elements in green with a "++" markup.

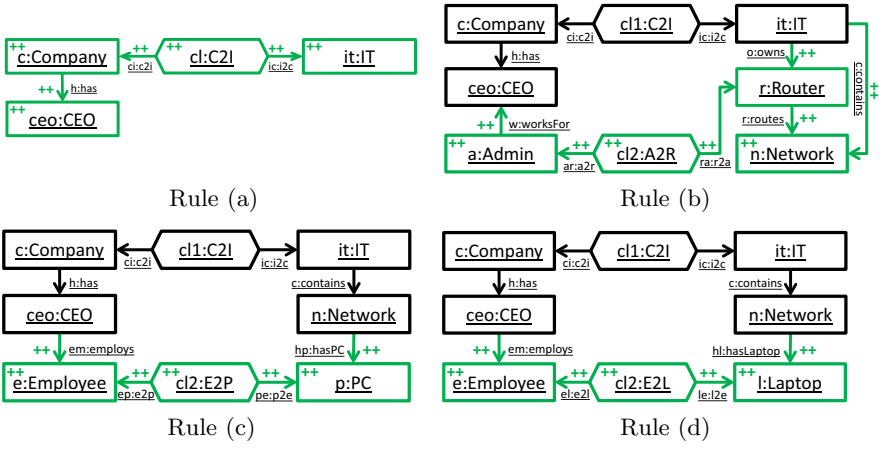


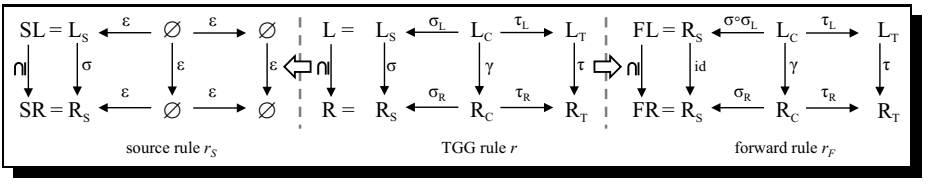
Fig. 3. Rules (a)–(d) for the integration

2.3 Derived Operational Rules

The real potential of TGGs as a bidirectional transformation language lies in the automatic derivation of *operational rules*. Such operational rules can be used to transform a given source domain model to produce a corresponding target domain model and vice versa. Although we focus in the following sections only on a forward transformation, all concepts and arguments are symmetric and can be applied analogously for the case of a backward transformation.

It has been proven by [5,16] that a sequence of TGG rules, which describes a simultaneous evolution, can be uniquely decomposed into (and conversely composed from) a sequence of *source rules* that only evolve the source model and *forward rules* that retain the source model and evolve the correspondence and target models. These operational rules serve as the building blocks used by a control algorithm for unidirectional forward and backward transformation.

Definition 5 (Derived Operational Rules). *Given a* $TGG = (TG, \mathcal{R})$ *and a rule* $r = (L, R) \in \mathcal{R}$*, a* source rule $r_S = (SL, SR)$ *and a* forward rule $r_F = (FL, FR)$ *can be derived according to the following diagram:*



Example. From Rule (c) of our running example (Fig. 3), the operational rules r_S and r_F depicted in Fig. 4 can be derived. The source rule extends the source graph by adding an `Employee` to an existing `CEO` in a `Company`, while the forward rule r_F *transforms* an existing `Employee` of a `CEO` by *creating* a new `E2P` link and a `PC` in the corresponding `Network`.

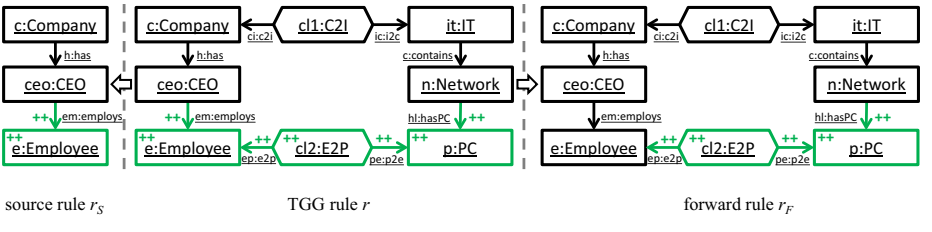


Fig. 4. Source and forward rules derived from Rule (c)

3 Related Work on TGG Control Algorithms

Constructing forward (and conversely backward) transformations from operational rules requires a *control algorithm* that is able to determine a sequence of forward rules to be applied to a given source graph. The challenge is to specify a control algorithm that is correct (only consistent graph triples are produced), complete (all consistent triples, which can be derived from a source or a target graph, can actually be produced), efficient (runtime complexity scales polynomially with the number of nodes to be processed), and still expressive enough for real-world applications. To better understand this challenge, we discuss how existing algorithms handle the source graph of our example triple (Fig. 2).

(I) Bottom-Up, Context-Driven and Recursive: An established strategy is to transform elements in a bottom-up context-driven manner, i.e., to start with a random node and check if all context nodes (dependencies) are already transformed *before* the selected initial node can be transformed. If a context node is not yet transformed, the algorithm transforms it, by recursively checking and transforming its context. Context-driven algorithms always start their transformation process with an arbitrarily selected node, without "knowing" if this was a good choice, i.e., if the node can be transformed immediately or if the input model as a whole is even valid. Such algorithms are correct, but, in general, have problems with completeness due to wrong *local* decisions.

(I.a) Backtracking: A simple backtracking strategy could be employed to cope with wrong local decisions. For our example, a first iteration over all nodes would determine that only `ES` together with `Andy` can be transformed by applying Rule (a). In a second iteration the algorithm would determine again in a trial and error manner that only `Ingo` can be transformed next with Rule (b), as neither `Tony` nor `Marius` can be transformed using Rule (c) or (d) (a `Network` is missing in the opposite domain). Finally, `Tony` and `Marius` can be transformed. This algorithm is correct and complete as shown in [5,16] but has exponential runtime and is, therefore, impractical for real-world applications.

It is, however, possible to guarantee polynomial runtime of the context-driven recursion strategy by restricting the class of supported TGGs appropriately as in case of the following approaches.

(I.b) Functional Behavior: Demanding *functional behavior* [7,9] guarantees that the algorithm can choose freely between applicable rules at every decision point and will always get the *same result* without backtracking. Although functional behavior might be suitable for fully automatic integrations, our experience with industrial partners [14,15] shows that user interaction or similar guidance (e.g., configuration files) of the integration process is required and leads naturally to non-functional sets of rules with certain degrees of freedom [13,14,15]. Please note that our running example is clearly non-functional due to Rules (c) and (d), which can be applied to the same elements on the source side, but create different elements on the target side. Therefore, depending on the choice of rule applications, *different* target graphs are possible with our running example. Demanding functional behavior is a strong restriction that reduces the expressiveness and suitability of TGGs for real-world applications [12,17].

Nevertheless, such a strategy has polynomial runtime and its applicability can be enforced statically via critical pair analysis [6].

(I.c) Local Completeness: Algorithms that allow a non-functional set of rules to handle a larger set of scenarios exploit the explicit traceability to cope with non-determinism and non-bijectivity [19], while still guaranteeing completeness for a certain class of TGGs. Hence, [12] demands *local completeness*, i.e., that a local decision between rules that can transform the current node *cannot* lead to a dead-end. This means that a local choice (which can be influenced by the user or some other means) might actually result in *different* output graphs, which are, however, always consistent, i.e., in the defined language of the TGG ($\mathcal{L}(TGG)$). For our running example, we could start with an arbitrary node, e.g., `Ingo`. According to Rule (b), a `CEO` and a `Company` are required as context and Rule (a) will thus be applied to `ES` and `Andy`. After processing `Ingo`, `Tony` and `Marius` can be transformed in an arbitrary order, each time making a local choice if a `PC` (Rule (c)) or `Laptop` (Rule (d)) is to be created. Furthermore, a *dangling edge check* is introduced in [12] to further enlarge the class of supported TGGs via a look-ahead to prevent wrong local decisions that would lead to "dangling" edges that can no longer be transformed. Note that our running example is *not* local complete, as it cannot be decided whether an `Admin` or an `Employee` should be transformed first (Rules (c) and (d) demand an element on the *target* side that can only be created by Rule (b)). For this reason, the algorithm might fail if it decides to start with one of the `Employees`. In this case, Rules (c) and (d) would state that `ES` and `Andy` are required as context and have to be transformed first. This is, however, insufficient as a `Network` must be present in the target domain as well. This context-driven approach fails here as transforming `ES` and `Andy` with Rule (a) *does not* guarantee that the employees `Marius` and `Tony` can be transformed. The problem here is that context-driven algorithms only regard the given input graph for controlling the rule application and do not consider *cross-domain context dependencies* such as `Network` in this case.

(II) Top-Down and Iterative: In contrast to context-driven recursive strategies, which lack a *global view* on the overall dependencies and seem to be unsuitable for an *incremental* synchronization scenario, algorithms can operate in a top-down iterative manner exploiting a certain global view on the whole input graph instead of arbitrarily choosing a node to be transformed.

(II.a) Correspondence-Driven: The algorithm presented by [11] requires that all TGG rules demand and create at least one correspondence link, i.e., a hierarchy of correspondence links must be built up during the transformation. The correspondence model can be used to store dependencies between links in this case and is interpreted as a directed acyclic graph, which is used to drive and control the transformation. This algorithm is both batch *and* incremental but it is unclear from [11] for which class of TGGs completeness can be ensured.

(II.b) Precedence-Driven: A precedence-driven strategy defines and uses a partial order of nodes in the source graph according to their *precedence*, i.e., the sorting guarantees that the nodes can only be transformed in a sequence that is compatible with the partial order.

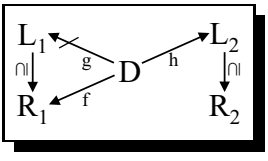
4 Rule Dependency and Precedence Analysis for TGGs

In this section, we present a node precedence analysis that provides a partial order required for a precedence-driven strategy, together with a rule dependency analysis that partially solves the problem of cross-domain context dependencies caused by context elements in the domain under construction.

4.1 Rule Dependency Analysis

To handle cross-domain context dependencies, we utilize the concept of *sequential independence* as introduced by [6], to statically determine which rules depend on other rules. The intuition is that a rule r_2 depends on another rule r_1, if r_1 creates elements that r_2 requires as context.

Definition 6 (Rule Dependency Relation $\lessdot_R$). *Given rules $r_1 = (L_1, R_1)$ and $r_2 = (L_2, R_2)$, r_2 is sequentially dependent on r_1 iff a graph D and morphisms f, h exist, such that there exists no morphism g as depicted to the right, i.e., at least one element required by r_2 (an element in L_2), is created by r_1 (this element is in R_1 but not in L_1).*



The precedence relation $\lessdot_R \subseteq \mathcal{R} \times \mathcal{R}$ is defined for a given TGG as follows:
$r_1 \lessdot_R r_2 \Leftrightarrow r_2$ *is sequentially dependent on* r_1.

In practice, $\lessdot_R$ can be calculated statically by determining all possible intersections of R_1 and L_2. If at least one element in an intersection is not in L_1 then r_2 is sequentially dependent on r_1 (i.e., $r_1 \lessdot_R r_2$).

Example. For the TGG rules of our running example (Fig. 3), the following pairs of rules constitute $\lessdot_R$: *Rule* $(a) \lessdot_R$ *Rule* (b), *Rule* $(a) \lessdot_R$ *Rule* (c), *Rule* $(a) \lessdot_R$ *Rule* (d), *Rule* $(b) \lessdot_R$ *Rule* (c), and *Rule* $(b) \lessdot_R$ *Rule* (d).

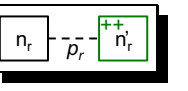
4.2 Precedence Analysis

The following definitions present our path-based node precedence analysis which is used to topologically sort the nodes in a source graph and thus control the iterative transformation process:

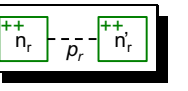
Definition 7 (Paths and Type Paths). *Let G be a typed graph with type graph TG. A* path *p between two nodes $n_1, n_k \in V_G$ is an alternating sequence of nodes and edges in V_G and E_G, respectively, denoted as $p := n_1 \cdot e_1^{\alpha_1} \cdot n_2 \cdot \ldots \cdot n_{k-1} \cdot e_{k-1}^{\alpha_{k-1}} \cdot n_k$, where $\alpha_i \in \{+, -\}$ specifies if an edge e_i is traversed from source $s(e_i) = n_i$ to target $t(e_i) = n_{i+1}$ (+), or in a reverse direction (–). A* type path *is a path between node types and edge types in V_{TG} and E_{TG}, respectively. Given a path p, its* type (path) *is defined as $type_p(p) := type_V(n_1) \cdot type_E(e_1)^{\alpha_1} \cdot type_V(n_2) \cdot type_E(e_2)^{\alpha_2} \cdot \ldots \cdot type_V(n_{k-1}) \cdot type_E(e_{k-1})^{\alpha_{k-1}} \cdot type_V(n_k)$.*

For our analysis we are only interested in paths that are induced by certain *certain patterns* present in the *TGG rules*.

Definition 8 (Relevant Node Creation Patterns). *For a* $TGG = (TG, \mathcal{R})$ *and all rules* $r \in \mathcal{R}$*, where* $r = (L, R) = (L_S \leftarrow L_C \rightarrow L_T, R_S \leftarrow R_C \rightarrow R_T)$*. The set* Paths_S *denotes all paths in* R_S *(note that* $L_S \subseteq R_S$*).*
The predicates $context_S : Paths_S \rightarrow \{true, false\}$ *and* $create_S : Paths_S \rightarrow \{true, false\}$ *in the source domain are defined as follows:*
context$_S(p_r) := \exists\ r \in \mathcal{R}$ *s.t.* p_r *is a path between two nodes* $n_r, n'_r \in R_S$ *:*
$(n_r \in L_S) \wedge (n'_r \in R_S \setminus L_S)$*, i.e., a rule* r *in* $\mathcal{R}$ *contains a path* p_r *which is isomorphic to the node creation pattern depicted in the diagram to the right.*



create$_S(p_r) := \exists\ r \in \mathcal{R}$ *s.t.* p_r *is a path between two nodes* $n_r, n'_r \in R_S$ *:*
$(n_r \in R_S \setminus L_S) \wedge (n'_r \in R_S \setminus L_S)$*, i.e., a rule* r *in* $\mathcal{R}$ *contains a path* p_r *which is isomorphic to the node creation pattern depicted in the diagram to the right.*



We can now define the set of interesting type paths, relevant for our analysis.

Definition 9 (Type Path Sets). *The set* TPaths_S *denotes all type paths of paths in* $Paths_S$ *(cf. Def. 8), i.e.,* $TPaths_S := \{tp \mid \exists\ p \in Paths_S \text{ s.t. } type_p(p) = tp\}$*. Thus, we define the restricted* create type path set *for the source domain as* $TP_S^{create} := \{tp \in TPaths_S \mid \exists\ p \in Paths_S \wedge type_p(p) = tp \wedge create_S(p)\}$*, and the restricted* context type path set *for the source domain as* $TP_S^{context} := \{tp \in TPaths_S \mid \exists\ p \in Paths_S \wedge type_p(p) = tp \wedge context_S(p)\}$*.*

In the following, we formalize the concept of *precedence between nodes*, indicating that one node could be used as context when transforming another node.

Definition 10 (Precedence Function $\mathcal{PF}_S$). *Let* $\mathcal{P} := \{\lessdot, \doteq, \cdot\lightning\cdot\}$ *be the set of precedence relation symbols. Given a* $TGG = (TG, \mathcal{R})$ *and the restricted type path sets for the source domain* TP_S^{create}*,* $TP_S^{context}$*. The precedence function for the source domain* $\mathcal{PF}_S : \{TP_S^{create} \cup TP_S^{context}\} \rightarrow \mathcal{P}$ *is computed as follows:*

$$\mathcal{PF}_S(tp) := \begin{array}{ll} \lessdot & \text{iff } tp \in \{TP_S^{context} \setminus TP_S^{create}\} \\ \doteq & \text{iff } tp \in \{TP_S^{create} \setminus TP_S^{context}\} \\ \cdot\lightning\cdot & \text{otherwise} \end{array}$$

Example. $\mathcal{PF}_S$ for our running example consists of the following entries:
Rule (a): $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO}) = \doteq$ and $\mathcal{PF}_S(\texttt{CEO} \cdot \texttt{has}^- \cdot \texttt{Company}) = \doteq$
Rule (b): $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO} \cdot \texttt{worksFor}^- \cdot \texttt{Admin}) = \lessdot$ and
$\mathcal{PF}_S(\texttt{CEO} \cdot \texttt{worksFor}^- \cdot \texttt{Admin}) = \lessdot$
Rules (c) and (d): $\mathcal{PF}_S(\texttt{Company} \cdot \texttt{has}^+ \cdot \texttt{CEO} \cdot \texttt{employs}^- \cdot \texttt{Employee}) = \lessdot$ and
$\mathcal{PF}_S(\texttt{CEO} \cdot \texttt{employs}^- \cdot \texttt{Employee}) = \lessdot$

Restriction. As our precedence analysis depends on paths in rules of a given TGG, the presented approach requires TGG rules that are (weakly) connected in each domain. Hence, considering the source domain, the following must hold: $\forall\ r \in \mathcal{R}, \forall\ n, n' \in R_S : \exists\ p \in \text{Paths}_S$ between n and n'.

Based on the precedence function $\mathcal{PF}_S$, relations $\lessdot_S$ and $\doteq_S^*$ can now be defined and used to topologically sort a given input graph and determine the sets of elements that can be transformed at each step in the algorithm.

Definition 11 (Source Path Set). *For a given typed source graph G_S, the* source path set *for the source domain is defined as follows:*
$P_S := \{p \mid p \text{ is a path between } n, n' \in V_{G_S} \wedge type_p(p) \in \{TP_S^{create} \cup TP_S^{context}\}\}$.

Definition 12 (Precedence Relation $\lessdot_S$). *Given $\mathcal{PF}_S$, the precedence function for a given TGG, and a typed source graph G_S. The precedence relation $\lessdot_S \subseteq V_{G_S} \times V_{G_S}$ for the source domain is defined as follows: $n \lessdot_S n'$ if there exists a path $p \in P_S$ between nodes n and n' such that $\mathcal{PF}_S(type_p(p)) = \lessdot$.*

Example. For our example triple (Fig. 2), the following pairs constitute $\lessdot_S$: (`ES` $\lessdot_S$ `Ingo`), (`ES` $\lessdot_S$ `Tony`), (`ES` $\lessdot_S$ `Marius`), (`Andy` $\lessdot_S$ `Ingo`), (`Andy` $\lessdot_S$ `Tony`), and (`Andy` $\lessdot_S$ `Marius`).

Definition 13 (Relation $\doteq_S$). *Given $\mathcal{PF}_S$, the precedence function for a given TGG, and a typed source graph G_S. The symmetric relation $\doteq_S \subseteq V_{G_S} \times V_{G_S}$ for the source domain is defined as follows: $n \doteq_S n'$ if there exists a path $p \in P_S$ between nodes n and n' such that $\mathcal{PF}_S(type_p(p)) = \doteq$.*

Definition 14 (Equivalence Relation $\doteq_S^*$). *The equivalence relation $\doteq_S^*$ is the transitive and reflexive closure of the symmetric relation $\doteq_S$.*

Example. For our example triple (Fig. 2), the following equivalence classes constitute $\doteq_S^*$: {`Andy`, `ES`}, {`Ingo`}, {`Tony`}, and {`Marius`}.

Definition 15 (Precedence Graph $\mathcal{PG}_S$). *The precedence graph for a given source graph G_S is a graph $\mathcal{PG}_S$ constructed as follows:*
(i) The equivalence relation $\doteq_S^*$ is used to partition V_{G_S} into equivalence classes $EQ_1, \ldots EQ_n$ which serve as the nodes of $\mathcal{PG}_S$, i.e., $V_{\mathcal{PG}_S} := \{EQ_1, \ldots, EQ_n\}$.
(ii) The edges in $\mathcal{PG}_S$ are defined as follows:
$E_{\mathcal{PG}_S} := \{e \mid s(e) = EQ_i,\ t(e) = EQ_j : \exists\ n_i \in EQ_i, n_j \in EQ_j \text{ with } n_i \lessdot_S n_j\}$.

Example. The corresponding $\mathcal{PG}_S$ constructed from our example triple is depicted in Fig. 5(a) in Sect. 5.

5 Precedence TGG Batch Algorithm

In this section, we present our batch algorithm (cf. Algorithm 1) and explain how the introduced rule dependency and node precedence analyses are used to efficiently transform a given source graph. For a forward transformation (a backward transformation works analogously), the input for the algorithm is a graph G_S, the statically derived rule dependency relation $\lessdot_R$, and the precedence function for the source domain $\mathcal{PF}_S$.

Procedure TRANSFORM determines a graph triple $G_S \leftarrow G_C \rightarrow G_T$ as output. The first step (line (2)) of the algorithm is to build the precedence graph

Algorithm 1. Precedence TGG Batch Algorithm

```
1: procedure TRANSFORM(G_S, ≺_R, PF_S)
2:     PG_S ← BUILDPRECEDENCEGRAPH(G_S, PF_S)
3:     while (PG_S contains equivalence classes) do
4:         readyNodes ← all nodes in equiv. classes in PG_S without incoming edges
5:         readyNodes ← sort readyNodes utilizing ≺_R
6:         for (node n in readyNodes) do
7:             transformedNodes ← CHOOSEANDAPPLYRULE(n)
8:             if transformedNodes ≠ ∅ then
9:                 PG_S ← remove all nodes in transformedNodes from PG_S
10:                break
11:            end if
12:        end for
13:        if transformedNodes = ∅ then
14:            terminate with error        ▷ Local Completeness Criterion violated
15:        end if
16:    end while
17:    return G_S ← G_C → G_T
18: end procedure
```

$\mathcal{PG}_S$ according to Def. 15. Note that the procedure BUILDPRECEDENCEGRAPH will terminate with an error if there is a cycle in the precedence graph and it is thus impossible to sort the elements of the source graph according to their dependencies. Starting on line (3), a while-loop iterates over equivalence classes in $\mathcal{PG}_S$ until there are none left. In the while-loop, the set *readyNodes* contains all nodes that can be transformed next, i.e., whose context elements have already been transformed (line (4)). This set is determined by taking all nodes in the equivalence classes of $\mathcal{PG}_S$, which do not have incoming edges (dependencies). On line (5), *readyNodes* is sorted according to the partially ordered relation $\prec_R$, i.e., the rules that can be used to transform nodes in *readyNodes* are determined, sorted with $\prec_R$ and reflected in *readyNodes*. This could be achieved by assigning an integer to each rule according to the partial order of $\prec_R$ and then selecting the largest number of all rules that translate $n \in$ *readyNodes* for n.[1] Next, a for-loop iterates over the sorted *readyNodes* (line (6)). On line (7) the procedure CHOOSEANDAPPLYRULE is used to determine and filter the rules as presented in [12], allowing for user input or choosing arbitrarily from the final applicable rules. If a rule could be successfully chosen and applied to transform n on line (7), a non-empty set of *transformedNodes* is returned that is used to update $\mathcal{PG}_S$ on line (9). In this case, the for-loop is terminated and the while-loop is repeated with the updated and thus "smaller" $\mathcal{PG}_S$. If *transformedNodes* is empty, the for-loop is repeated for the next node in *readyNodes*. If *transformedNodes*, however, *remains* empty on line (13), we know that no node in *readyNodes* has been transformed and that the algorithm has hit a dead-end. This can only

[1] If it is not possible to sort *readyNodes* due to cycles in $\prec_R$, this additional analysis supplies no further information and *readyNodes* remains unchanged.

happen for TGGs that violate the *Local Completeness Criterion* (cf. algorithm strategy I.c in Sect. 3) and are *not* in the class of supported TGGs.

Example. To demonstrate the presented algorithm, we apply a forward transformation for the source graph of our example triple depicted in Fig. 2. Given as input is G_S, the rule dependency relation $\lessdot_R$ (depicted as a graph in Fig. 5(b)), and the precedence function $\mathcal{PF}_S$ (cf. example for Def. 10). On line (2), the precedence graph $\mathcal{PG}_S$ for G_S, depicted in Fig. 5(a), is built. $\mathcal{PG}_S$ is acyclic, hence the transformation can continue.

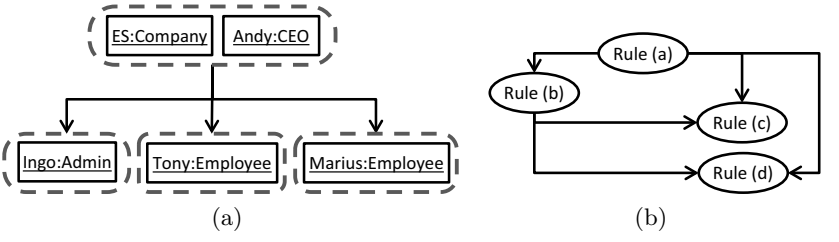


Fig. 5. $\mathcal{PG}_S$ for the input graph (left) and relation $\lessdot_R$ for all rules (a)–(d) (right)

On line (4), the set *readyNodes* is determined, consisting in this case of the nodes `ES` and `Andy` from a single equivalence class of $\mathcal{PG}_S$. On line (5), only one rule can be used to transform both nodes and, therefore, the sorting is trivial. On line (6) `ES` or `Andy` is chosen randomly, and in either case, the only candidate rule is Rule (a) (Fig. 3), which can be directly applied on line (7). Again in either case, *transformedNodes* contains both nodes as Rule (a) transforms `ES` and `Andy` simultaneously. $\mathcal{PG}_S$ is updated on line (9) to consist of three unconnected equivalence classes `Ingo`, `Tony`, and `Marius`, and the for-loop terminates. In the second iteration through the while-loop, *readyNodes* now contains all these three elements and will be sorted according to $\lessdot_R$ on line (5). This time, the sorting reveals that `Ingo` must be transformed before `Tony` and `Marius` as Rules (c) and (d) both require a `Network` as context in the target domain, which can only be created by applying Rule (b) first, i.e., *Rule* $(b) \lessdot_R$ *Rule* (c), *Rule* $(b) \lessdot_R$ *Rule* (d) (Fig. 5(b)). The for-loop in line (6), therefore, starts with `Ingo`. Applying Rule (b) (line (7)) puts `Ingo` in *transformedNodes*, $\mathcal{PG}_S$ is updated on line (9) to now contain only `Tony` and `Marius` and the for-loop is terminated with the break on line (10). In the third iteration, *readyNodes* contains `Tony` and `Marius`, and no sorting is needed as Rules (c) and (d) do not depend on each other. On line (6) `Tony` could be randomly selected first and (arbitrarily or via user input) Rule (c) could be chosen to be applied on line (7). After updating $\mathcal{PG}_S$ again and breaking out of the for-loop, only `Marius` remains untransformed. Similar to the penultimate iteration, Rule (d) could be selected and applied this time. Updating $\mathcal{PG}_S$ on line (9) empties the precedence graph, which terminates the while-loop on line (3). The created graph triple depicted in Fig. 2 is returned on line (17).

Formal Properties of the Precedence TGG Batch Algorithm

In the following we argue that the presented algorithm retains all formal properties stipulated in [17] and proved for the context-driven algorithm of [12].

Definition 16 (Correctness, Completeness and Efficiency).
Correctness: *Given a source graph G_S, the transformation algorithm either terminates with an error or produces a graph triple $G_S \leftarrow G_C \rightarrow G_T \in \mathcal{L}(TGG)$.*
Completeness: *For all triples $G_S \leftarrow G_C \rightarrow G_T \in \mathcal{L}(TGG)$, the transformation algorithm produces a consistent triple $G_S \leftarrow G'_C \rightarrow G'_T \in \mathcal{L}(TGG)$ for the input source graph G_S.*
Efficiency: *According to [17], a TGG batch transformation algorithm is efficient if its runtime complexity class is $O(n^k)$, where n is the number of nodes in the source graph to be transformed and k is the largest number of elements to be matched by any rule r of the given TGG.*

All properties are defined analogously for backward transformations.

Theorem. *Algorithm 1 is correct, complete and efficient for any source-local complete TGG [12].*

Proof.
Correctness: If the algorithm returns a graph triple, i.e., does not terminate with an error, it was able to determine a sequence of source rules $r_{1_S}, r_{2_S}, \ldots, r_{n_S}$ that would build the given source graph G_S and, thus, the corresponding sequence of forward rules $r_{1_F}, r_{2_F}, \ldots, r_{n_F}$ that transform the given source graph (Def. 5). The *Decomposition and Composition Theorem* of [5] guarantees that it is possible to compose the sequence $r_{1_S}, r_{2_S}, \ldots, r_{n_S}, r_{1_F}, r_{2_F}, \ldots, r_{n_F}$ to the sequence of TGG rules $r_1, r_2, \ldots, r_n$ which proves that the resulting graph triple is consistent, i.e., $G_S \leftarrow G_C \rightarrow G_T \in \mathcal{L}(TGG)$. □
Completeness: Showing completeness is done in two steps: First of all, we consider the algorithm without the additional concept of rule dependencies via the relation $\lessdot_R$.

The remaining algorithm transforms nodes with the same concepts (e.g., dangling edge check) as the previous algorithm in [12], but iteratively *in a fixed sequence*, for which we guarantee, by definition of the precedence graph (cf. 15), that the context of every node is always transformed first. As the context-driven strategy taken by the algorithm in [12] is able to transform a model by arbitrarily choosing an element and transforming its context elements in a bottom-up manner (cf. Sect. 3), the fixed sequence taken by our algorithm must be a possible sequence that could be chosen by the algorithm in [12]. Algorithm 1 can, therefore, be seen as forcing the context-driven algorithm to transform elements in one of the possible sequences, from which it can arbitrarily choose. This shows that all completeness arguments from [12] can be transferred to the new algorithm, i.e., Algorithm 1 is complete for the class of local complete TGGs.

In a second step, we now consider the algorithm with the additional relation $\lessdot_R$ and, therefore, the capability of handling specifications with cross-domain

context dependencies as in our running example. We have shown in Sect. 3 that the algorithm presented in [12] cannot cope with such specifications as they violate the local-completeness criterion. We can, hence, conclude that Algorithm 1 is more expressive than the previous context-driven algorithm as it can handle certain TGGs that are not local complete. We leave the precise categorization of this new class of TGGs to future work. □

Efficiency: Building the precedence graph $\mathcal{PG}_S$ on line (2), essentially a topological sorting, is realizable in $O(n^l)$, where l is the maximum length of relevant paths according to $\mathcal{PF}_S$. Note that l can be at most of size k (the largest number of elements to be matched by any rule r of the given TGG), thus we can estimate this with $O(n^k)$. The while-loop starting on line (3) iterates through $\mathcal{PG}_S$, which will be decreased every time by at least one node from an equivalence class. The while-loop is, thus, run in the worst-case (equivalence classes in $\mathcal{PG}_S$ all consisting of exactly one node) n times. In the while-loop, we select equivalence classes without incoming edges in line (4). This can be achieved in $O(n)$ by iterating through $\mathcal{PG}_S$. Building the topological order on line (5) requires inspecting all nodes in *readyNodes* and their appropriate rules in $O(n)$. The for-loop starting on line (6) iterates in the worst-case over all nodes in *readyNodes* where updating $\mathcal{PG}_S$ on line (9), requires traversing all successor nodes which is at most $n-1$ (i.e., $O(n)$). As argued in [12], transforming a node, i.e., checking all conditions and performing pattern matching (line (7)), is assumed to run in $O(n^k)$ (cf. Def. 16). Summarizing, we obtain: $n^k + n \cdot (n + n + n \cdot (n^k + n)) \in O(n^k)$. □

As TGGs are symmetric [8], all arguments can be transferred analogously to backward transformations.

6 Related Work on Alternative Bidirectional Languages

Complementing our related work on TGG batch algorithms (cf. Sect. 3), we now focus on *alternative bidirectional languages* that share and address similar challenges as TGGs but take fundamentally different strategies. As bidirectionality is a challenge in various application domains and communities, there exists a substantial number of different approaches, formalizations and tools [18]. The lenses framework is of particular interest when compared to TGGs, as [8] has shown that incremental TGGs can be viewed as an implementation of a *delta-based* framework for *symmetric lenses*. Although we have presented a batch algorithm for TGGs, our ultimate goal is to provide a solid basis for an efficient incremental TGG implementation. As compared to existing lenses implementations for string data or trees such as *Boomerang* [2], TGGs are better suited for MDE where model transformations operate on complex *graph-like* structures. Similar to TGGs, *GRoundTram*, a bidirectional framework based on graph transformations [10], aims to support model transformations in the context of MDE. There are, however, a number of interesting differences: (i) While GRoundTram demands a forward transformation from the user and automatically generates a consistent backward transformation, TGGs (in this respect similar to lenses) provide a language from which both forward and backward transformations

are automatically derived. Both approaches face a different set of non-trivial challenges. (ii) GRoundTram uses UnQL+, which is based on the graph query algebra UnCAL, with a strong emphasis on compositionality, while TGGs are rule-based algebraic graph transformations. (iii) GRoundTram maintains traceability in an implicit manner while TGGs create explicit typed traceability links between integrated models, which can be used to store extra information for incremental model synchronization or manual reviews. In contrast to both Boomerang and GRoundTram, TGGs adhere to the fundamental *unification* principle in MDE (everything is a model) and as such, a bidirectional model transformation specified as a TGG is a model which is conform to a well defined TGG metamodel. Unification has wide-reaching consequences including enabling a natural *bootstrap* and *higher order transformations*. Finally, TGGs served as an inspiration and basis for the standard OMG bidirectional transformation language QVT and can be regarded as a valid implementation thereof [18].

7 Conclusion and Future Work

In this paper, an improvement of our previous TGG batch algorithm was presented. We introduced a novel *node precedence analysis* of TGG specifications combined with a *rule dependency analysis* to further support the batch transformation control algorithm in determining the node processing order. The result is an iterative batch transformation strategy in a top-down manner with increased expressiveness. We have shown that this algorithm runs in polynomial runtime and complies to the formal properties for TGG implementations according to [17], and, therefore, is well-suited for real-world applications where efficiency is almost as important as the reliability of the expected result.

As a next step, we shall implement the presented algorithm as an extension of our current batch implementation in our metamodeling tool eMoflon[2] [1], and start working on an efficient incremental TGG algorithm based on our rule dependency and node precedence analyses. Finally, providing a *rule checker* that decides at compile time if a given TGG can be transformed by our algorithm is a crucial task to improve the usability of our tool.

References

1. Anjorin, A., Lauder, M., Patzina, S., Schürr, A.: eMoflon: Leveraging EMF and Professional CASE Tools. In: Heiß, H.U., Pepper, P., Schlingloff, H., Schneider, J. (eds.) Proc. of MEMWe 2011. LNI, vol. 192. GI (2011)
2. Bohannon, A., Foster, J., Pierce, B., Pilkiewicz, A., Schmitt, A.: Boomerang: Resourceful Lenses for String Data. ACM SIGPLAN Notices 43(1), 407–419 (2008)
3. Czarnecki, K., Helsen, S.: Feature-based Survey of Model Transformation Approaches. IBM Systems Journal 45(3), 621–645 (2006)
4. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional Transformations: A Cross-Discipline Perspective. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 260–283. Springer, Heidelberg (2009)

[2] http://www.moflon.org

5. Ehrig, H., Ehrig, K., Ermel, C., Hermann, F., Taentzer, G.: Information Preserving Bidirectional Model Transformations. In: Dwyer, M.B., Lopes, A. (eds.) FASE 2007. LNCS, vol. 4422, pp. 72–86. Springer, Heidelberg (2007)
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Monographs in Theoretical Computer Science. An EATCS Series. Springer, New York (2006)
7. Giese, H., Hildebrandt, S., Lambers, L.: Toward Bridging the Gap between Formal Semantics and Implementation of Triple Graph Grammars. In: Lúcio, L., Vieira, E., Weißleder, S. (eds.) Proc. of MoDeVVA 2010, pp. 19–24. IEEE (2010)
8. 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)
9. Hermann, F., Golas, U., Orejas, F.: Efficient Analysis and Execution of Correct and Complete Model Transformations Based on Triple Graph Grammars. In: Bézivin, J., Soley, M.R., Vallecillo, A. (eds.) Proc. of MDI 2010. ICPS, vol. 482, pp. 22–31. ACM (2010)
10. Hidaka, S., Hu, Z., Inaba, K., Kato, H., Nakano, K.: GRoundTram: An Integrated Framework for Developing Well-Behaved Bidirectional Model Transformations. In: Alexander, P., Pasareanu, C., Hosking, J. (eds.) Proc. of ASE 2011, pp. 480–483. IEEE (2011)
11. Kindler, E., Rubin, V., Wagner, R.: An Adaptable TGG Interpreter for In-Memory Model Transformations. In: Schürr, A., Zündorf, A. (eds.) Proc. of Fujaba Days 2004, pp. 35–38 (2004)
12. Klar, F., Lauder, M., Königs, A., Schürr, A.: Extended Triple Graph Grammars with Efficient and Compatible Graph Translators. In: Engels, G., Lewerentz, C., Schäfer, W., Schürr, A., Westfechtel, B. (eds.) Nagl Festschrift. LNCS, vol. 5765, pp. 141–174. Springer, Heidelberg (2010)
13. Königs, A.: Model Transformation with Triple Graph Grammars. In: Proc. of MTIP 2005 (2005)
14. Lauder, M., Schlereth, M., Rose, S., Schürr, A.: Model-Driven Systems Engineering: State-of-the-Art and Research Challenges. Bulletin of the Polish Academy of Sciences, Technical Sciences 58(3), 409–422 (2010)
15. Rose, S., Lauder, M., Schlereth, M., Schürr, A.: A Multidimensional Approach for Concurrent Model Driven Automation Engineering. In: Osis, J., Asnina, E. (eds.) Model-Driven Domain Analysis and Software Development, pp. 90–113. IGI (2011)
16. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
17. Schürr, 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)
18. Stevens, P.: A Landscape of Bidirectional Model Transformations. In: Lämmel, R., Visser, J., Saraiva, J. (eds.) GTTSE 2007. LNCS, vol. 5235, pp. 408–424. Springer, Heidelberg (2008)
19. Stevens, P.: Bidirectional Model Transformations in QVT: Semantic Issues and Open Questions. SoSym 9(1), 7–20 (2008)

A Timed Automata-Based Method to Analyze EAST-ADL Timing Constraint Specifications

Tahir Naseer Qureshi, De-Jiu Chen, and Martin Törngren

KTH – The Royal Institute of Technology, Stockholm, Sweden
{tnqu,chen,martin}@md.kth.se

Abstract. The increasing development complexity of automotive embedded systems has led to industrial needs of improved information management, early verification and validation of a system etc. EAST-ADL; an automotive-specific architectural description language provides a structured model-based approach for information management throughout the development process. A method to formally analyze consistency of EAST-ADL based timing constraint specifications using timed-automata is presented. A mapping scheme providing a basis for automated model-transformations between EAST-ADL and timed-automata is the main contribution. The method is demonstrated with a case study of a brake-by-wire system. Guidelines for extending the mapping framework are also provided.

Keywords: Model-based development, EAST-ADL, Timed-Automata, UPPAAL, Timing Constraints.

1 Introduction

Model-based development (MBD), i.e. the use of computerized models for different activities [1], is being applied in various engineering domains to manage complexities and increase development efficiency. In case of automotive embedded control systems, MBD is being used extensively in many different forms such as automatic code generation from design models using tools like Simulink. Traditionally, the development starts with algorithm (e.g. control algorithm for fuel control) formalization followed by steps like rapid prototyping, generation of production code, hardware-in-the-loop (HIL) testing and final calibrations [2]. The process is efficient for single ECUs (Electric Control Units) but several issues related to timing, interface and communication occur during ECU integration [2]. There is a need for efficient information management and integration of models, tools and languages related to different views and abstraction levels. The views can be product-related, such as hardware, software and infrastructure, or concern related, such as safety, dependability etc. Information traceability, reusability of solutions, early verification and validation, reduced time and cost are examples of the intended benefits from an efficient information management and integration.

One approach to deal with multiple views and structured information management is to base the development on a comprehensive system model to which the views can

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 303–318, 2012.

be related. There exist several generic as well as domain-specific solutions. SysML[1] , EAST-ADL (Electronics Architecture and Software Technology- Architecture Description Language) [3] and AUTOSAR (AUTomotive Open System ARchitecture) [4] are examples of such solutions. EAST-ADL is automotive specific having a broad coverage of system lifecycle and specification support with different views and concerns in a well-structured manner. It is also aligned with AUTOSAR and other automotive standards such as ISO26262 [5]. It relies on external tools for activities like analysis of specifications related to functionality, requirements or safety, verification and validation etc. Wide industrial acceptance of EAST-ADL is hindered by its limited tool support.

The presented work is motivated by the fact that ensuring consistency between the constraints specified for different parts of a system can lead to decreased integration issues. A method which paves a way for automated model-transformation between EAST-ADL and timed automata is presented in this paper. It is shown that EAST-ADL timing constraint specifications and the execution behavior of a component can be abstracted as a network of timed-automata. The main contribution is a mapping framework based on pre-defined timed-automata templates, its usage as well as extension guidelines for checking specification consistency. A case-study of a brake-by-wire system is used to demonstrate the usage of the framework with PapyrusUML[6] and UPPAAL[7] as the tools for modeling and analysis respectively.

2 EAST-ADL and Timing Extension – Concept and Notations

EAST-ADL evolved through several European projects during the last decade complementing the best industrial practices such as Hardware-In-Loop (HIL) and Software-In-Loop (SIL) simulations with the goal to provide an architectural description language to facilitate the development of automotive embedded systems. The core of the EAST-ADL language definition [3] consists of structural specifications at four different abstraction levels (namely vehicle, analysis, design and implementation). For example product line features (end-to-end functionality) and their variations are specified at the vehicle level whereas the detailed design of functional components, connections and allocations to various hardware components is carried out at the design level. The core supports hierarchical specifications of a system with the concepts of function types and prototypes together with various types of ports and connectors with automotive specific attributes. Specifications of requirements, dependability, variability, and behavioral and timing constraints are supported by corresponding language extensions which refer to the core functional artifacts of EAST-ADL. This modular approach not only separates the definition of functional and non-functional aspects but also enables the use of existing tools for various development activities.

The presented work focuses on a subset of EAST-ADL artifacts applicable to the design level of abstraction and described as follows:

[1] http://www.omgsysml.org/#Specification

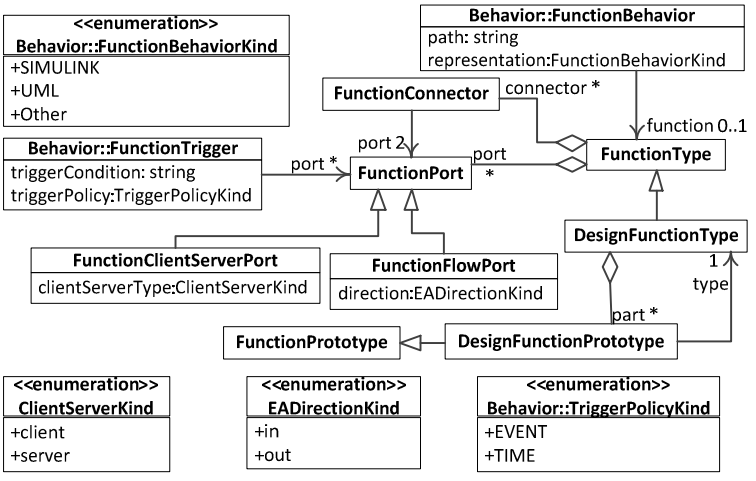


Fig. 1. EAST-ADL core structure and behavior extension

The core and behavior artifacts focused in this paper are shown in Fig. 1 where the artifacts prefixed with *Behavior::* are part of the behavior annex and the rest belongs to the core language definition.. The behavior of a function (FunctionType or DesignFunctionType in Fig. 1) can be classified as the execution behavior and the internal behavior. EAST-ADL relies on external representations like Simulink for internal behavior representation and specifies its execution behavior in the form of triggering information (determined by the triggerPolicy of the associated FuncitonTrigger) i.e. if a function is event triggered (e.g. arrival of data at its port) or time-triggered. The behavior of an EAST-ADL function has three main steps consisting of reading data at input ports, performing computations and writing data on the output port. All functions have run-to-completion semantics i.e. a function runs all the steps before it starts to execute again. All the functions run concurrently unless specified by the designer. Moreover, the ports of an EAST-ADL function have single-sized overwritable and non-consumable buffer semantics.

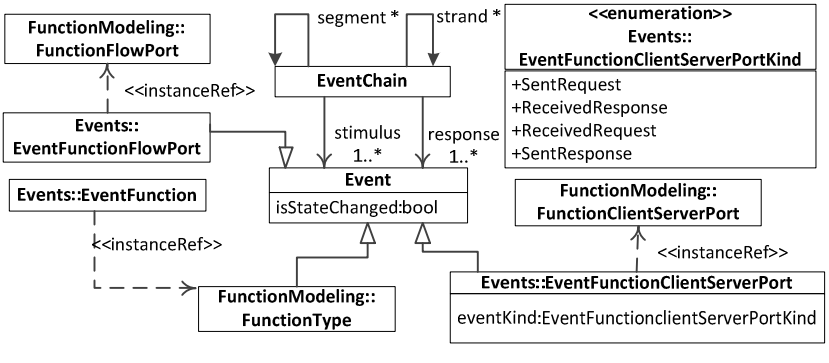


Fig. 2. Events and event chains in EAST-ADL

The timing extension of EAST-ADL is derived from TADL (Time Augmented Description Language) [12]. It can be used to specify the timing constraints on the execution behavior of a function and precedence between different functions. As shown in Fig. 2 the timing extension is based on the concepts of events and event chains. *EventFunction*, *EventFunctionFlowPort* and *EventFunctionClientServerPort* are the three event kinds referring to the triggering of a function by some sort of dispatcher, arrival of a data at a port and service requested (or received) by a client-server port respectively.

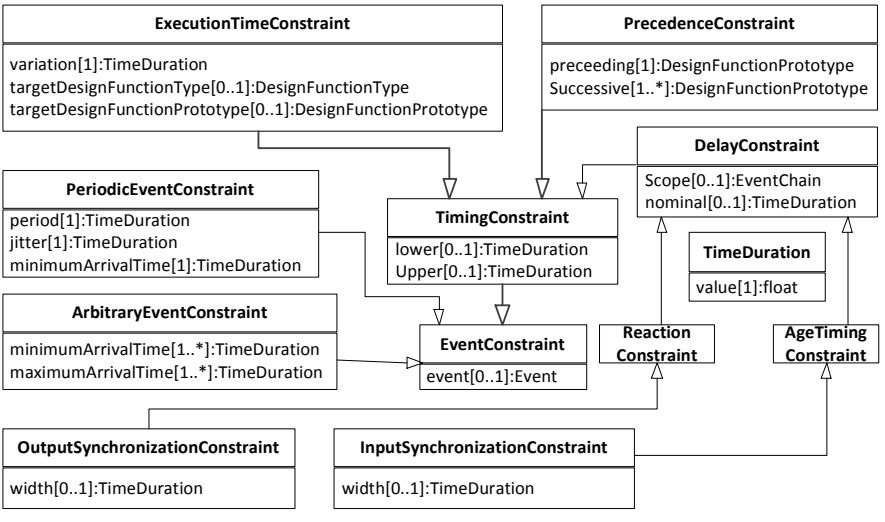


Fig. 3. EAST-ADL timing constraints

A function or a group of functions perform some kind of transformation of data present at their input ports and send the output through their output ports. *Event chains* and the constraints applied on them as well as individual events enable a designer to specify end-to-end timing constraints such as the minimum and maximum time allowed from the occurrence of one event called *stimulus* to the occurrence of another event called *response*. An event chain can further be refined into smaller event chains called *strands* (parallel chains) or *segments* (sequenced). The constraints addressed in this paper are shown in Fig. 3. The periodicity of an event occurrence is an example of possible constraints on the events shown as *PeriodicEventConstraint* in Fig. 3. In addition to above it is also possible to specify constraints on internal behavior (shown as FunctionBehavior in Fig. 1). For additional information, the readers are referred to [3, 9, 12].

3 Timed Automata and UPPAAL

Timed automata (TA) [8] is essentially an automata augmented with clock and time semantics to enable formal model-checking of real-time systems. It has been used for

modeling and verification of several systems and scenarios. Let C be a set of clocks, F(C) a set of clock constraints in the form $x \diamond y \; or \; x + y \diamond c$ where $x, y \in C, c \in N \; and \; \diamond \in \{<, \leq, =, \geq, >\}$. A timed-automata TA is a tuple $(L, Lo, C, \sum, I, E)$ where,

- L is a finite set of locations, nodes or states.
- Lo is the initial location.
- $\sum$is a set of actions
- $E \subseteq L \times \mathrm{F}(C) \times \sum \times 2^C \times L$ is a set of edges or transition.
- $I: L \rightarrow \mathrm{F}(C)$ assigns invariants to locations

The semantics of timed-automata is a transition system where state and pairs (*l*,*u*), and transition are defined by rules

$(l,u) \xrightarrow{d} (l, u+d) if \; u \in I(l) and \; (u+d) \in I(l)$ for a non-negative real $d \in \mathbb{R}_+$

$(l,u) \xrightarrow{a} (l', u') if \; l \xrightarrow{g,a,r} l', u \in g, u' = [r \mapsto 0]u \; and \; u' \in I(l')$

where

- u ,v denote functions known as clock assignments C mapping to $\mathbb{R}_+$. In addition $u \in g$ is used to denote that *u* satisfy the guard g.
- For $d \in \mathbb{R}_+$, $u+d$ is the clock assignment which maps all $x \in C$ to $u(x) + d$
- For $r \subseteq C$, $[r \mapsto 0]u$ denote the clock reset mapping all clocks in r to 0 with u for the other clocks in C.

Often a set of timed-automata are used in a networked form with a common set of clocks and actions. A special synchronization action denoted by an exclamation sign (!) or a question mark (?) is used for synchronization between different timed automata. A timed automata in a network is concurrent unless and until mechanisms like synchronization actions are applied. The readers are referred to [8] for a formal definition and semantics of a network of timed-automata. .

3.1 UPPAAL

UPPAAL is a timed-automata based model checking tool for modeling, validation and verification of real-time systems. The tool has three main parts: an editor, a simulator and a verifier, for modeling, early fault detection (by examination i.e. without exhaustive checking) and verification (covering exhaustive dynamic behavior) respectively. A system in UPPAAL is modeled as a network of timed automata. A subset of CTL (computation tree logic) is used as the query language in UPPAAL for verification. In addition to the generic timed-automata UPPAAL uses the concept of *broadcast* channels for synchronizing more than two automata. The concept of urgent and committed state is also introduced to force a transition as soon as it is enabled. The three kinds of properties which can be checked with UPPAAL are (i) *Reachability* i.e. some condition can possibly be satisfied, (ii) *Safety* i.e. some condition will never occur and (iii) *Liveness* i.e. some condition will eventually become true.

UPPAAL uses the concept of templates for reusability and prototyping of system components. Each template can be instantiated multiple times with varying parameters. The instantiation is called a process. The tool has been used in many industrial cases such as a gear box controller from Mecel AB and Philips Audio protocol and several more[2].

4 EAST-ADL and Timed-Automata Relationship

Both timed-automata and EAST-ADL are developed for real-time embedded systems. While the purpose of TA is model-checking of generic real-time system, EAST-ADL on the other focuses on describing structural and some behavioral aspects of embedded systems. There exist at least four different possibilities for mapping and relating EAST-ADL with timed-automata. (i) One possibility is to use timed-automata for defining the behavior of a system by exploiting EAST-ADL external behavior representation support (FunctionBehavior in Fig. 1) as done in [15]. (ii) Another possibility is to transform EAST-ADL behavior extension artifacts [9] to timed-automata for behavioral analysis. (iii) A third possibility is to model timing constraints with timed-automata with a suitable behavior abstraction. (iv) Finally, a combination of both timing constraints and EAST-ADL behavior constraints [9] can also be considered for formal analysis using timed-automata. As the internal functional behavior and hence the associated constraints are out of scope of the work, only the timing constraints and design level of abstraction is considered, corresponding to possibilities (iii), with the following assumptions and limitations:

- Only the *Functional Design Architecture* (FDA) is considered. The design level has two parts namely Functional design architecture (FDA) and *Hardware Design Architecture* (HDA). While HDA covers hardware topology, FDA is used to model software components, middleware functions, device drivers and hardware transfer-functions. Hence, FDA together with and constraints such as time budgets applied on its contained functions can provide a suitable abstraction for the target analysis.
- Only one function type i.e. the FDA is allowed to have prototypes of other functions in its composition for the sake of simplicity. An FDA is a DesignFunctionType (Fig. 1) which can contain several parts (DesignFunctionPrototype). Each prototype refers to a type (DesignFunctionType). This kind of modeling allows a hierarchical composition with infinite depth. Such concept of hierarchical decomposition is not possible with timed-automata; therefore, an additional mechanism for flattening the functional hierarchy (if allowed) of EAST-ADL models will be required if hierarchy is allowed.

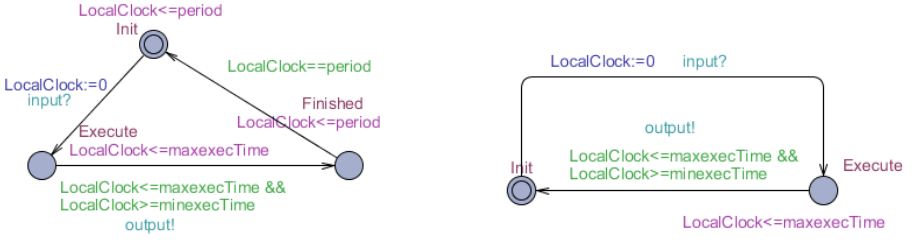
4.1 Mapping Scheme

This subsection presents a mapping scheme between EAST-ADL and timed-automata. The proposed scheme is based on the experiences from a previous work [10] where a timed-automata model of an existing emergency braking system (EBS)

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

was utilized to identify relevant EAST-ADL artifacts followed by the verification of the mapping by transformation of a representative industrial case-study of a brake-by-wire system in EAST-ADL to timed-automata. The same approach is used for the proposed mapping scheme and its validation. This mapping consists of templates for each function and timing constraint type. The templates for the timing constraints act as monitors indicating if a constraint is met or not. In addition to the description of the semantics of the mapped EAST-ADL artifacts, template implementations in UPPAAL are also presented for illustration.

Event. In terms of timed automata, an event can be modeled as a synchronization action. For example, the synchronization action *output!* for the transition to the final from the execute state shown in b can be considered as an event corresponding to an *EventFunctionFlowPort* referring to a port with direction *out* or *EventFunctionClientServerPort* with kind of either *sentRequest* or *sentResponse*.



(a) Periodic (b) Aperiodic

Fig. 4. Function templates

Function Execution Behavior. As shown in Fig. 4, a function can be modeled with three or two locations for time-triggered and event triggered systems respectively. The type is determined by the triggering policy of its function trigger shown in Fig. 1. In Fig. 4 , the *Init*, *Execute* and *Finished* states represent the initial, execution (related to internal behavior) and waiting for the execution period to finish. The parameters *maxexecTime* and *minexecTime* are obtained from *ExecutionTimeConstraint* (Fig. 3) where *max-* and *minexecTime* correspond to the upper and lower limits of timing constraint. On the other hand the period is obtained from *PeriodicEventConstraint* applied on the *EventFunction* referring to the *DesignFunctionType* under consideration. The *input?* and *output*! synchronization actions correspond to reading and writing on all the input and output ports respectively of an EAST-ADL function.

Timing and Event Constraints. A constraint is either satisfied or not satisfied; therefore, four locations corresponding to initial, intermediate, success, fail states are necessary to model a constraint. On occurrence of an event, the automaton proceeds to the intermediate state(s). Based on the applicable guard conditions the (*fail*) *success* state is reached if a particular constrained is (not) satisfied. Both the timing (related to event chain) and event constraints refer to one or more events. The transition(s) to reach a *fail* (*safe*) state is enabled by clock guards and synchronization actions representing the timing bounds and event occurrences respectively. Each automaton

has a local clock denoted by *LocalClock* in the following text. The event and timing constraint templates are as follows:

Periodic event constraint: A periodic event constraint is used to specify constraints on the periodicity of an event. An UPPAAL template for a periodic event constraint is shown in Fig. 5. The three applied parameters (also shown in Fig. 3) in this template *are period (P), jitter (J) and the minimum arrival time* of the event. The synchronization action "event?" can refer to any event whose periodicity is required to be constrained.

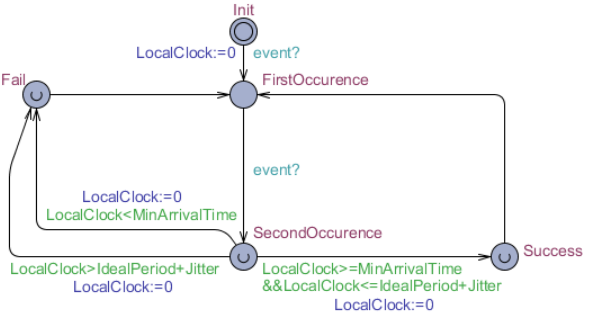
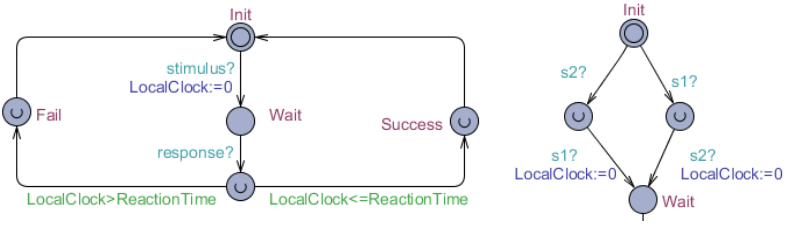


Fig. 5. Periodic event constraint template

***Reaction constraint*:** A reaction constraint specifies a bound between the occurrences of stimuli? and responses of an event chain. According to [3] there exist five possible specification combinations ({upper, lower}, {upper, lower, jitter}, {upper}, {lower}, {nominal, jitter}) for a delay constraint. The presented work considers only one combination i.e. {upper} which corresponds to the maximum time allowed.



(a) Reaction constraint (b) Extension illustration

Fig. 6. Reaction constraint template

In the reaction constraint template (Fig. 6a) the clock is reset when a *stimulus* event occurs. As soon as the response event occurs the automata transits to *Fail* or *Success* state depending on the elapsed time i.e. the *LocalClock* value. The template considers only one stimulus and one response. It can be extended for multiple stimuli and

responses by adding additional states and parallel transitions. For example, in case of two stimuli, two states between the *Init* and *Wait* states e.g. *s1* and *s2* can be added where the transition from *Init* to *s1* can correspond to the first stimulus occurrence, *s1* to *Wait* corresponding to the second stimulus and vice versa. This is illustrated in Fig. 6b.

***Precedence constraint*:** A precedence constraint specifies the constraint on the order of execution of events. A template for two events is shown in Fig. 7 where *input2* is constrained to occur after *input1*. In order to extend this template for more events additional states can be added similar to reaction constraint leading to the *Fail* and *Success* states.

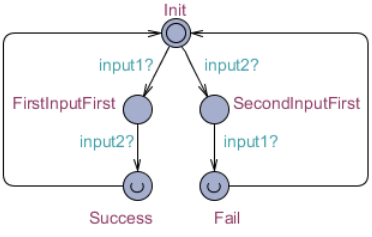


Fig. 7. Precedence event constraint template

***Arbitrary event constraint*:** An arbitrary event constraint specifies bounds on an aperiodic event. The event can occur singly, occasionally or irregularly. The bounds can be specified between two or more occurrences of an event by minimum and maximum arrival time attributes in the form of an array. The element of the array corresponds to the time between the first and later occurrences of an event. For example, the third element corresponds to the time constraint between the first and the fourth event occurrences.

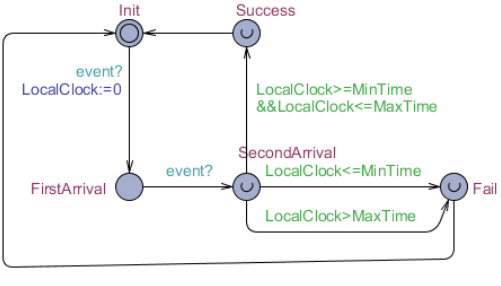


Fig. 8. Arbitrary event constraint template

The template shown in Fig. 8 models bounds on two consecutive occurrence of an event. If it is desired to constraint occurrence of the first and third event then a new state and transition can be added before the clock value is checked. The template can be extended for more than three occurrences in a similar fashion.

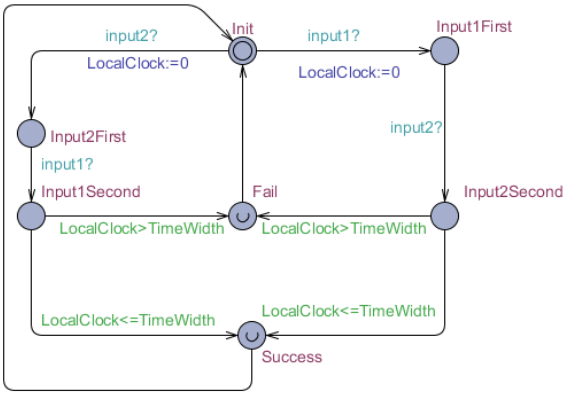


Fig. 9. Input synchronization constraint template

***Input and output synchronization constraints*:** An input synchronization constraint specifies the time width within which a set of stimuli of an event chain should occur. In order to model the input synchronization constraint the automata shown in Fig. 9 is used. The actions *input1?* and *input2?* represent two stimuli and the parameter *Time-Width* (represented as *width* in Fig. 3) determines the maximum time allowed between two stimuli.

The output synchronization event is similar to the input synchronization except that instead of response the stimuli are constrained to occur in a specified time width. Therefore, the same template can be used with the responses as inputs. In order to incorporate more events, additional states and transitions corresponding to different combinations of occurrences can be added between *Init* and *Fail* states similar to the reaction time constraint.

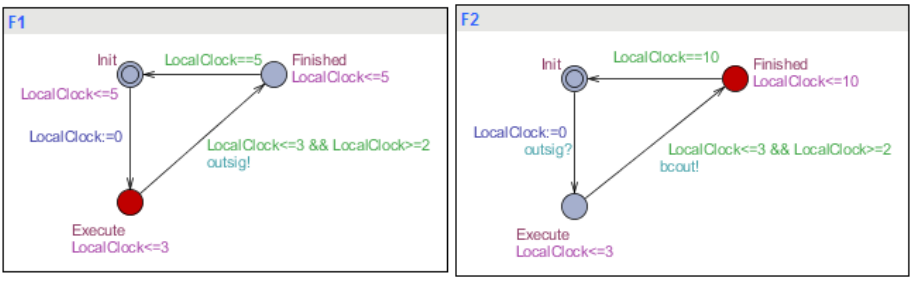
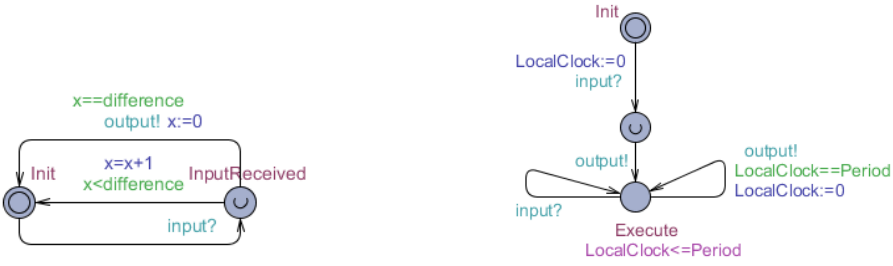


Fig. 10. Execution rate difference illustration

Port Buffer and Execution Rate Difference. An EAST-ADL system is inherently deadlock free due to the semantics of ports discussed in section 2. However, two UPPAAL processes with different periodicities synchronized using channels can theoretically lead to a deadlock situation due to timing mismatch. For example, the two functions *F1* and *F2* are in a deadlock condition shown in Fig. 10. Both *F1* and *F2* have the same minimum and maximum execution time but *F1* has a faster rate of execution (twice in Fig. 10). The deadlock is due to the fact that sender *F1* is ready to output a signal but *F2* is not in the state where it can receive it.



(a) Fast to slow rate transition (b) Slow to fast rate transition

Fig. 11. Rate transition templates

Due to the above mentioned issue, we introduce the concept of rate-transition templates shown in Fig. 11 inspired by the Matlab rate-transition block[3]. The template in Fig. 11a is used when the sender is running at a faster rate (less period) than the receiver. The actions *input?* and *output!* correspond to the input from the sender and output to the receiver respectively. The *difference* parameter is the *difference of frequency* obtained by dividing the period of the receiver with that of sender. For the case where the sender has low frequency, the template in Fig. 11b is used. This template mimics the the EAST-ADL assumed communication mechanism (over-writing semantics). The parameter *Period* corresponds to the period of the receiving function. This solution acts similar to a zero-order hold function in Simulink.

4.2 Verification

With the above defined relationship between EAST-ADL and timed-automata, the verification essentially becomes a reachability analysis in the following form:

1. A given constraint is satisfied iff for all initial conditions, the state "Fail" is never reached for all cases.
2. A system is free of any inconsistencies iff there is not deadlock and all the constraints are satisfied.

4.3 Usage Considerations and Limitations

Template Usage. The templates discussed above exist in several variants depending on the composition of EAST-ADL function and the applied timing constraints. The first variation is the existence of input and output channels. If an EAST-ADL function does not have any input port then a template without an input channel will be chosen and vice versa. Such type of function is shown as *F1* in Fig. 10. The second variation is the type of channel used for synchronization where the rule applied is that "*if an output of a function is an input for two or more functions then the channel is of type broadcast otherwise*". A function in this case also includes the modeled timing constraints. For example, the output of *F1* in Fig. 10 will be of type broadcast if there

[3] `http://www.mathworks.se/help/toolbox/simulink/slref/ratetransition.html`

another function *F3* exists whose input is the output of *F1*. The function *F3* can be another function or a timing constraint.

In addition to above, consistency between time measurements units have to be ensured by the user. For example, it is now allowed to use *ms* for timing constraint and *ns* for another. In addition, the UPPAAL processes have to be updated with new channel names when rate-transition is used. This is shown in Fig. 13 where BTC and GBC were supposed to communicate with *CalculatedTorque* channel but due to the use of rate-transition *CalcualatedTorque2* channel is introduced in the system description.

To verify a given set of timing constraints of a system specification, the following two query language syntaxes have to be used.

- *A [] (not deadlock)* to verify if there exist any deadlock.
  - Two possible reasons for a deadlock can be (i) the absence of one or more rate-transition templates in case of different frequency between two communicating functions and (ii) an incorrect synchronization channel type. This typically occurs if an ordinary channel type where a broadcast type is required.
- *A [] (not XX.Fail)* to verify that a timing constraint modeled with an UPPAAL process named XX never reaches the failed state.
  - In case a fail state is reached, the timing constraints have one or more inconsistencies.

4.4 Mapping Summary

EAST-ADL	UPPAAL	Remarks
Function Prototype	Fig 4	Type determined by the associated function trigger policy. *Min*- and *max* execution time from the associated execution time constraint and period from the periodic event constraint referring to the *EventFunction* associated with the function.
Periodic event constraint	Fig 5	Direct mapping of event name. *MinArrivalTime* is the *lower* time limit specified by the constraint. Other parameters are directly mapable.
Reaction constraint	Fig 6	Stimulus and response names from the *event chain* in scope, *reaction time* from the *upper* value the timing constraint.
Precedence constraint	Fig 7	Direct mapping of event names.
Arbitrary event constraint	Fig 8	Direct mapping of event name and parameters.
Input synchronization constraint	Fig 9	Event names from list of *stimuli* of the event chain in scope, time width from the upper time limit of the reaction time constraint associated with the event chain.
Output synchronization constraint	Fig 10	Event names from list of *responses* of the event chain in scope, time width from the upper time limit of the reaction time constraint associated with the event chain.
Functional Design Architecture	System	Each of the prototypes in FDA take the form of a process in UPPAAL. The connectors will take the form of synchronization channel.

5 Brake-by-Wire Case Study

The brake-by-wire (BBW) system is a representative industrial case study. This case has been used in several EAST-ADL related projects like ATESST2 and TIMMO. It provides coverage of EAST-ADL artifacts and methodology at multiple abstraction levels. A simplified version of the case is shown in Fig. 12. The figure is a snapshot from its EAST-ADL (UML profile[4]) implementation. For simplicity, three actuators, ABS functions and their connections are not shown in the figure. The triggers, events and constraints of the case study are listed in the following tables:

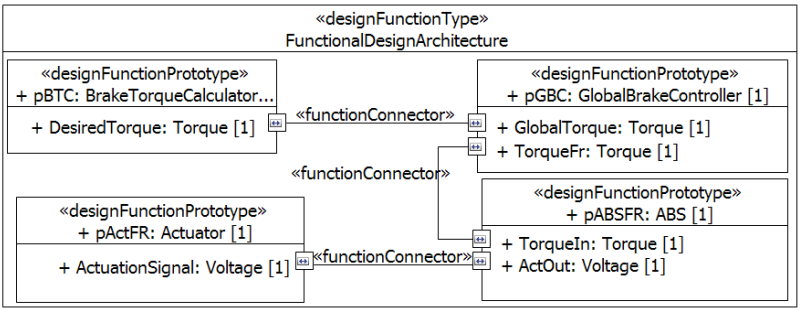


Fig. 12. UML implementation of the Brake-by-wire system

Table 1. Function Triggers

Trigger Name	Attributes
BTCTriggerEvent	TargetFunctionPrototype=pBTC, TriggerPolicy = Time
GBCTriggerEvent	TargetFunctionPrototype=pGBC, TriggerPolicy = Time
ABSFRTrigger	TargetFunctionFlowPort=pABSFR, TriggerPolicy=Event
ABSFLTrigger	TargetFunctionFlowPort=pABSFL, TriggerPolicy =Event
ABSRRTrigger	TargetFunctionFlowPort=pABSRR, TriggerPolicy =Event
ABSRLTrigger	TargetFunctionFlowPort=pABSRL, TriggerPolicy =Event

Table 2. Events

Event Name	Type	Attributes
BTCTriggerEvent	EventFunction	TargetFunctionPrototype=pBTC
GBCTriggerEvent	EventFunction	TargetFunctionPrototype=pGBC
CalculatedTorque	EventFunctionFlowPort	TargetFunctionFlowPort=DesiredTorque TargetFunctionPrototype=pBTC
actuation1	EventFunctionFlowPort	TargetFunctionFlowPort=ActOut TargetFunctionPrototype=pABSFR

[4] http://www.maenad.eu/public/EAST-ADL-ProfileSpecification_M2.1.9.pdf

Table 3. Event Chain

Event Chain Name	Attributes
EC1	Stimulus = CaclulatedTorque, Response = actuation1

Table 4. Constraints

Constraint Name	Type	Attributes
BTCExecution	Execution time	TargetFunctionPrototype=pBTC , Lower = 3 , Upper =5
GBCExecution	Execution time	TargetFunctionPrototype=pGBC, Lower = 2, Upper =6
ABSFRExecution	Execution time	TargetFunctionPrototype=pABSFR, Lower = 2, Upper =3
ABSFLExecution	Execution time	TargetFunctionPrototype=pABSFL, Lower = 2, Upper =3
ABSRRExecution	Execution time	TargetFunctionPrototype=pABSRR, Lower = 2, Upper =3
ABSRLExecution	Execution time	TargetFunctionPrototype=pABSRL, Lower = 2, Upper =3
RC1	Reaction	Scope = EC1, ReactionTime = 50 ms
PEC1	Periodic event	TargetEvent= CaclulatedTorque, MinArrivalTime= 3ms, Ideal-Period = 10 ms, Jitter = 9 ms
PCC1	Precedence	Preceding = CalculatedTorque, Successive = actuation1

An UPPAAL model of a subset of the constraints and functions described above is shown in Fig. 13.

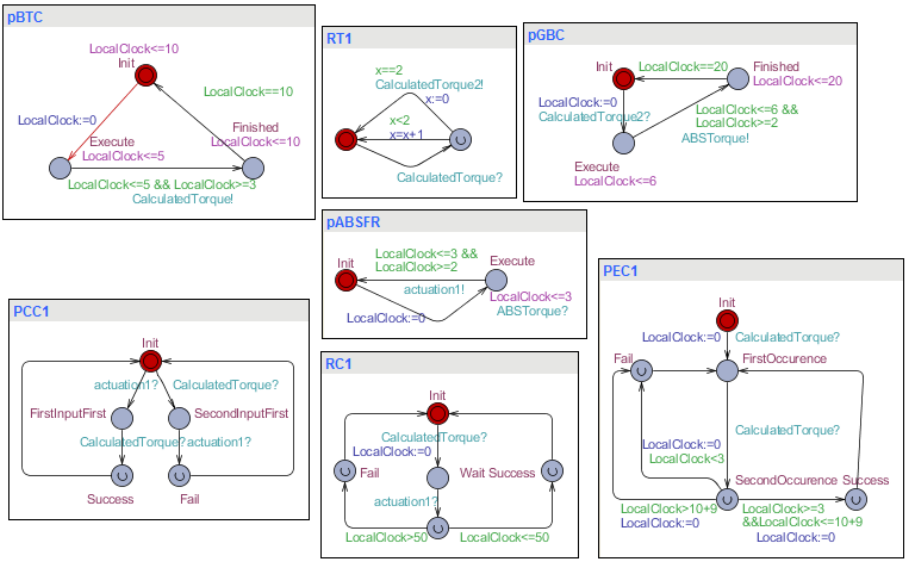


Fig. 13. Brake-by-wire model in UPPAAL

In the above figure, pBTC, pGBC and pABSFR are function prototypes shown in Fig. 12b. RT1 is a rate transition process added between pBTC and pGBC. PCC1, RC1 and PEC1 are listed in Table 4 where clock guards (in green) correspond to the listed timing values. As listed in Table 1 pABSFR is event triggered whereas pBTC and pGBC are time triggered. The stimulus and response of the event chain in scope of RC1 is listed in Table 3 with the corresponding events in Table 2.

All the constraints were found to be satisfying the specifications for the BBW system. Experiments were made to validate the templates. It included changing the periodicity of BTC and the order of inputs for the precedence constraint i.e. by using BTC output as the successive event instead of the preceding one. The latter case is shown in Fig. 13. Intuitively an increase in the period of BTC to a particular level should violate the periodicity constraint. The same observation was made with the UPPAAL model. Similarly, the change in the order of sequence leads to the *Fail* state of the precedence constraint.

6 Related Work

A number of efforts have been carried out to enable the analysis, verification and validation of system architecture design captured in EAST-ADL. This paper is an extension of [10] where the work was limited to the reaction time constraint and evaluation of the possibility of model transformation between EAST-ADL and UPPAAL. [11] presents an effort to integrate the SPIN model checker for formal verification of EAST-ADL models. The automata addressed by SPIN are untimed and the SPIN transformation needs to be updated for the latest EAST-ADL release. Furthermore, the timing constraint package used in this work is a subset of TADL (Timing Augmented Description Language) [12] developed by the TIMMO project consortium[5]. The authors of [13] proposed the use of MARTE[6] for complementing EAST-ADL to enable timing analysis. A method for timed-automata based analysis of EAST-ADL models is presented in [15] where timed-automata are used for behavior analysis and modeling. The work in this paper complements the above cited references by providing a method to check the consistency of the timing constraints before they are actually used for detailed timing analysis of any kind.

7 Discussion

A method to analyze consistency in timing constraints specified using EAST-ADL is presented. The proposed mapping scheme is a basis for transformations between EAST-ADL and timed-automata based tools. Our earlier work [10] shows that it is possible to automate model transformations between EAST-ADL and UPPAAL based on their meta-models. The transformation using the concept of templates is a part of the planned future work where the main challenge is the absence of one-to-one mappings, unidirectional links from EAST-ADL extensions to core language constructs and, the variations due to the usage criteria mentioned in section 4.

---

[5] http://timmo-2-use.org/

[6] http://www.omg.org/spec/MARTE/1.0/

There also exists possibility to generate test cases from the timing constraints which can later be used to test the final product or its prototype. A further investigation of EAST-ADL is required for such support. Another issue which requires investigation is the combined analysis of internal behavior and timing constraints and the method to transfer the results of analysis back for storing them as part of the EAST-ADL model using its verification and validation extension.

References

1. Törngren, M., Chen, D., Malvious, D., Axelsson, J.: Model-Based Development of Automotive Embedded Systems. In: Automotive Embedded Sytems Handbook (2009)
2. Lönn, H., Freund, U.: Automotive Architecture Description Languages. In: Automotive Embedded Systems Handbook (2009)
3. The ATESST2 Consortium, EAST-ADL Domain Model Specification, Project Deliverable 4.1.1 (June 2010), http://www.atesst.org/home/liblocal/docs/ATESST2_D4.1.1_EAST-ADL2-Specification_2010-06-02.pdf
4. AUTOSAR Website, http://www.autosar.org/ (accessed January 2011)
5. Road vehicles – Functional Safety, International Organization for Standardization, ISO 26262 (Draft International Standard) (2009)
6. PapyrusUML Website, http://www.papyrusuml.org (accessed January 2011)
7. 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)
8. Bengtsson, J.E., Yi, W.: Timed Automata: Semantics, Algorithms and Tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004)
9. The ATESST2 Consortium, Update Suggestions for Behavior Support, Project Deliverable 3.1, Appendix A3.4 (June 2010), http://www.atesst.org/home/liblocal/docs/ATESST2_Deliverable_D3.1_A3.4_V1.1.pdf
10. Qureshi, T.N., Chen, D.J., Persson, M., Törngren, M.: Towards the Integration of EAST-ADL and UPPAAL for Formal Verification of EAST-ADL Timing Constraint Specification. Presented at TiMoBD (Time Analysis and Model-Based Design, from Functional Models to Distributed Deployments) Workshop, October 9 (2011)
11. Feng, L., Chen, D.J., Lönn, H., Törngren, M.: Verifying System Behaviors in EAST-ADL2 with the SPIN Model Checker. In: IEEE International Conference on Mechatronics and Automation, Xi'an, China, August 4-7 (2010)
12. The TIMMO Consortium, TADL: Timing Augmented Description Language Version 2, Project Deliverable 6 (2009), http://www.timmo.org/pdf/D6_TIMMO_TADL_Version_2_v12.pdf
13. Mallet, F., Peraldi-Frati, M.A., André, C.: Marte CCSL to Execute East-ADL Timing Requirements. In: Proceedings of ISORC 2009, pp. 249–253 (2009)
14. Object Management Group. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, formal/2009-11-02 (2009), http://www.omg.org/spec/MARTE/1.0/PDF
15. Kang, E.-Y., Schobbens, P.-Y., Pettersson, P.: Verifying Functional Behaviors of Automotive Products in EAST-ADL2 Using UPPAAL-PORT. In: Flammini, F., Bologna, S., Vittorini, V. (eds.) SAFECOMP 2011. LNCS, vol. 6894, pp. 243–256. Springer, Heidelberg (2011)

Code Generation Nirvana

Petr Smolik and Pavel Vitkovsky

Metada
{petr.smolik,pavel.vitkovsky}@metada.com

Abstract. Life is fun and prospect of reincarnations is thus very attractive. People enjoy various ways how models may be transformed to executable code, how information may be derived, enriched, superimposed. It could take a number of complex transformations to reach the state of nirvana of a finally running application. Each such model transformation is like a reincarnation, new existence in a different body, the spirit mostly staying the same. We have been for years fascinated with this and tried different ways and approaches and we are experiencing a progress. We have extensively applied code generation in areas of enterprise systems integration and enterprise frontends. During time we have done code generation different ways into different target languages and we have also done a lot of direct model interpretation. More and more we value nirvana over many reincarnations, nevertheless there is still place left for code generation. In this paper we share our model-driven experience.

Keywords: Code Generation, Domain-Specific Modeling (DSM), Domain-Specific Modeling Languages (DSML), Model-Driven Engineering (MDE), Model Interpretation, Executable Models, XML, XSLT, XQuery.

1 Introduction

Model-driven approaches deal with system complexity by introducing higher levels of abstraction. Models abstract from technology specifics and attempt to focus on the important aspects of a given problem domain. The best models talk in terms of the domain expert's concepts and enable definition of systems by means of defining and relating instances of these concepts. The ability to directly express the domain-specific knowledge in models leads to higher efficiency in systems design and implementation. [1][2][3][4][5]

Nevertheless there are obstacles. It is hard to identify the right concepts on the right levels of abstraction. It is always easier to settle down on abstractions that are closer to the solution space where the resulting systems or applications are actually implemented, but it is much harder to find the right abstractions that are closer to the actual problem domain and its concepts.

Our experience comes mostly from the financial services domain, where one would expect to manipulate concepts like account, payment, transaction, or loan. But we did not manage to reach this level of abstraction in our models yet. We still do not configure current accounts and loan accounts, and enact their models in running applications. One of the reasons may be that financial institutions have many core systems

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 319–327, 2012.

already in place and their new needs are being satisfied by introducing dozens of new systems around the old ones and then heavily integrating their functionalities via integration layers, while trying to build unified frontends, since it is unbearable for the branch employees to tackle thirty different unintegrated ways to manage several customer's financial products.

For this reason the domain-specific modeling languages that we have so far designed were centered more around integration and frontends. The abstractions that are defined, manipulated, related, documented, and managed are not accounts, but components, operations, action flows, mappings, mapping tables, forms and their widgets, all this being composed into specific channel applications within multi-channel solutions. We still consider these modeling languages domain-specific, but their focus is in the integration or channel application domains, not in the financial services domain. Nevertheless, our ongoing goal is to reach higher and tackle the real business.

In the following chapters of this paper we do not present specifics of our metamodels for integration and multi-channel applications. These are not that interesting and could be shown on tools presentations. What we found interesting during the process of designing model-driven systems is the process of reaching model execution, either via code generation, or direct model interpretation, or something between.

2 Model Executability

Executability of models means that it is possible to transform models into executable form without any further coding in some general-purpose programming language. Models defined in domain-specific modeling languages (DSMLs) may be executable if these languages are designed to be executable.

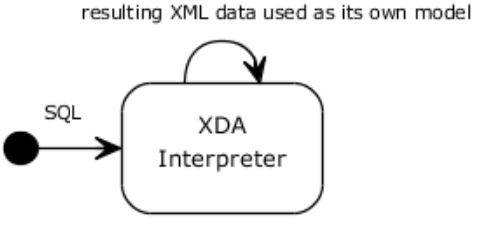
Model is executable when it is possible to create a running application just by creating the model. The goal is to model as little as possible, but still all that is needed. The core thing that makes models to be relatively simple and abstract is constraint. Only as little as possible set of concepts should be manipulated in the model and limits should be imposed on the variabilities provided by these concepts. Everything else is „an execution framework“ (or “domain framework”). An execution framework enables the models to „live“. It interprets the attributes of entities defined in the model. It enacts them. Framework may be written as components constructed using a general-purpose programming language. Framework may be composed also from various interpreters interpreting different transformed representations of models. It is possible to explore these various ways to make models executable.

We think of code generation as means to make models executable. Not to generate skeletons that need to be completed with hand-written final code. Code generation transforms, maybe in several steps (reincarnations), models into machine-executable code. It is also possible to directly execute models with interpreters specifically written to execute them based on their metamodels. Code generation and model interpretation are then two different alternative strategies to "implement" execution tools [6]. Interestingly, the process of code generation is a process of model interpretation. The execution of the resulting code is not.

3 Chicken or an Egg

The evolution of programming languages in themselves (bootstrapping) is well known [7]. It is interesting to see this in the domain-specific languages. Take for example a language to define how specific XML data should be retrieved from tables of a relational database. Such language is for example the XML Data Access (XDA) language [8]. This language has been used in our modeling tool and it represents a modeling language in the domain of retrieval of XML data from relational databases. Models in the XDA language are mostly concerned with "concepts" that correspond to tables in a relational database and "concept views" that define tree structures over the graph of related concepts.

XDA interpreter executes models defined in the XDA language. This interpreter will always take an XML definition of what data should be retrieved and will perform the actual retrieval by calling a relational database and join data from various tables into a complex XML document.




```
(start)-SQL>(XDA\nInterpreter)-resulting XML data used as
its own model>(XDA\nInterpreter)
```
[1]

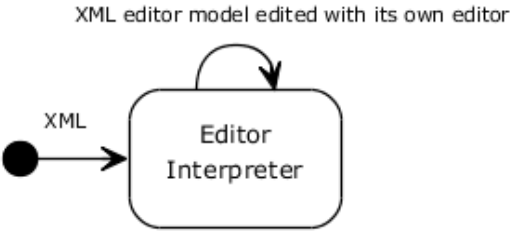
It starts to be interesting when we store the specific XDA models in the database. In such a case, in order to retrieve the XDA model from the database we need the model itself. Without itself the model could not be retrieved. The model is like a chicken and the result of interpretation is like an egg. But there is no egg without a chicken. In the beginning there had to be some, if the simplest, chicken or an egg. Thus for the XDA interpreter to start working, it is necessary to hand-create the chicken-like XDA model.

This seems to be also applicable in the area of GUI interpreters. Take for example an "Editor Interpreter". The purpose of an editor interpreter is to enable editing of a structure of data. This structure may be expressed as an XML. For a specific editor, the editor interpreter is configured with an editor model. We could imagine that this is also expressed as an XML, though it generally does not have to.

The editor interpreter takes the editor model and provides user a GUI for editing a chunk of data. Initially an editor model (a chicken) has to be hand-created, nevertheless it is thereafter possible to use the proto-editor to edit its own definition (an egg) and expand it to increase its ability. Of course this means that the editor interpreter may also need some expanding.

---

[1] Images were generated via simple UML activity diagram DSL and image generator available at `http://yuml.me/`.




```
(start)-XML>(Editor\nInterpreter)-XML editor definition
edited with its own editor>(Editor\nInterpreter)
```

When creating executable models this self-defining property is often encountered. It does not matter if interpreters are simple or complex, or whether there are many or few model transformation stages. Initially we started with interpretation via many complex transformations and intermediary interpretations and later we progressed to more directly interpreted solutions as shown in the following sections.

4 Integration Reincarnations

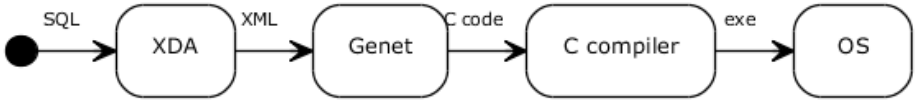
Enterprise application integration (EAI) enables several independent systems to communicate chunks of data among each other to fulfill particular purpose. Technically this may be implemented in various ways based on different technologies. From the modeling perspective the format is not really important. What is important is the way how data are structured, how the system interfaces differ, and how they are mapped to each other.

4.1 Many Reincarnations

We faced situation that large amount of integration was to be done and C++ was the required target language because of speed concerns. XSLT [9] and XQuery [10] were then too new and too slow. In order to get the large amount of production grade C++ integration code we created an integration modeling language.

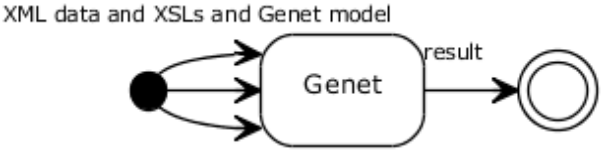
In our case integration models define specific integration services and their operations. Each operation has an input and output interface defined as a tree structure of data that may be composed directly or with the help of reusable complex types. An operation may directly represent some external functionality to be called, or may define a composite flow that calls several other operations. Each composite flow is composed of actions, mainly call actions that call other operations and decision actions that decide on what routes should be taken in the flow. Data mappings are defined on each action call to provide proper inputs to operations being called. These mappings use a simple expression language modeled as a computation tree that enables mapping of diverse structures to each other.

Models in the integration modeling language were persisted in a database and edited with our generated editors.




```
(start)-SQL>(XDA)-XML>(Genet)-C code>(C compiler)-
exe>(OS)
```

XML Data Access (XDA) [8] interpreter was used to obtain the model from a relational database. We hand-created several transformation XSLTs and used Genet interpreter to orchestrate complex transformations to create high-quality and optimized C++ code.




```
(start)->(Genet)-result>(end),
(start)->(Genet),
(start)-XML data and XSLs and Genet model>(Genet)
```

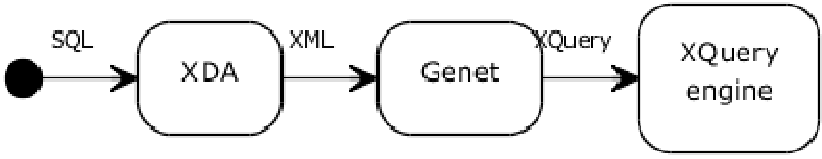
The Genet interpreter itself executes several steps of model transformations based on a Genet model. Genet is an interpreter of a configuration of a transformation pipeline. It interprets a network of gen nodes where each gen node represents one particular XSLT or XQuery transformation that may have outputs of several other gen nodes as its sources. So before resulting C++ comes out, there may be several intermediary transformations. Intermediary transformations are partial derivations of the source model, so that it is in the end easier to come-up with cleanest and nicest code possible. Intermediary transformations are themselves further model reincarnations into intermediary languages.

Finally, the resulting generated C++ code was compiled and executed. The mapping model contained service interface definitions on a service bus, service interfaces of backend services, and large amount of mapping and routing rules. The resulting executable implements transformation from one structure of message to another, for many message types, data formats (not only XML), and many integrated systems.

In order for the model to be morphed into an executable code this way it takes many transformations (SQL->XML...->XML…->XML->C code->exe).

4.2 Fewer and Fewer Reincarnations

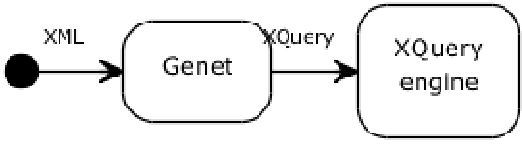
The previous many-reincarnations integration solution lived for several years before new integration solutions came to the market and enabled XQuery to be used for data mapping and routing effectively. We switched from C++ generation to XQuery generation. The models did not have to change, only model transformations did. It became easier to develop and test the integration solution because the step of lengthy C++ compilation was skipped.




```
(start)-SQL>(XDA)-XML>(Genet)-XQuery>(XQuery\nengine)
```

Model transformations were now less complex, because XQuery is a transformation language (a transformation DSL) whereas C++ is a general-purpose language. The step of lengthy compilation was also removed since XQuery is directly executed by the XQuery engine.

Further on, it slowly became clear that the relational database is not the greatest place to store models. So we decided that direct representation of models in XML would improve the situation.




```
(start)-XML>(Genet)-XQuery>(XQuery\nengine)
```

The situation was better, but it still took quite a while for the whole solution of hundreds to thousands of XQueries to be generated from models. So in the end one of our colleagues wrote a mapping interpreter. The interpreter takes an XML representation of the model and directly interprets it to provide the data mapping services.

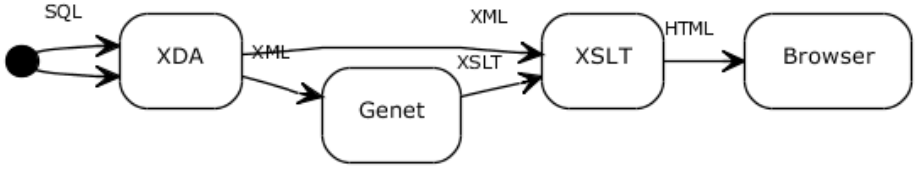

```
(start)-XML>(Mapping\ninterpreter)
```

Now there is no transformation at all. No code generation whatsoever. No reincarnations. Direct nirvana. Nevertheless, at the moment we do not have enough long-term experience with the mapping interpreter. The complexity of code generation may or may not have just moved into the complexity of the interpreter. Metamodel concepts are now represented as entities in the general-purpose programming language that the interpreter is written in (Java). Interpretation is provided real-time by manipulating the input data based on the metamodel entities. It is completely different than code generation. Interpretation directly produces behavior, not code to be executed.

5 User Interface Reincarnations

To enable creation and use of domain-specific modeling languages on our projects we developed a web-based modeling and metamodeling solution. Interestingly, with this solution we have also experienced similar history of lowering number of transformations. In this case it was about transformations of metamodels into functioning model editors.

In these user interface models we are mainly concerned with defining tree-structured editors for individual model object types. Model objects need to be listed, edited, and each object, based on its type, may have any number of levels with composite sub-objects that again need to be listed and edited.




```
(start)->(XDA)-XML>(XSLT)-HTML>(Browser),
(start)-SQL>(XDA)-XML>(Genet)-XSLT>(XSLT)
```

We started with a system that utilized XSLs to build an application GUIs and required a set of XML definition files to configure the XML Data Access interpreter used to obtain XML data from a relational database. In order to be able to create GUIs effectively, we developed an editor modeling language that was used to define editors and all the XML configurations for the data access. All the XSLT transformations were suddenly generated for the GUIs. There were XSLTs that were used to generate XSLTs. This enabled fast creation of modeling GUIs without having to write SQL queries or transformations to HTML.

Although this solution is quite interesting, it is also very complex and hard to maintain and extend. There are probably too many reincarnations of the model before it is really executed. Recently we work on removing this complex set of transformations and would like to reach the nirvana of a running application as fast as possible.

The newly designed editor interpreter will directly interpret the editor definition (a model) and enable to edit the corresponding data. Apart from producing HTML that is still interpreted at the browser, nirvana will be reached quickly without many complex reincarnations.


```
(start)->(Editor\ninterpreter)-HTML>(Browser),(start)-XML
editor definition and data>(Editor\ninterpreter)
```

We hypothesize that an ecosystem of various model interpreters will enable faster construction of applications. It is possible that there will be model interpreters that will encapsulate within themselves the actual stages of code generation, compilation, and final execution. The question may not be whether code generation or direct interpretation. Both may suit some situations better. They may also be combined. Interpretation and code generation should thus be seen as continuum and not as two alternatives [11].

6 Conclusion

Reaching code generation nirvana means getting out a running application out of a set of executable models. Models may be expressed in domain-specific modeling languages (DSMLs) that are interpreted by code generators that generate executable code or directly interpreted by purpose-built model interpreters. Such interpreters use their internal representation of a model to provide expected useful behavior. Direct model interpretation is different than code generation, because code generation focuses on creating an executable code. The code generation process itself is the process of model interpretation, but the final result of code generation is an executable that does the real work, not interpretation.

There are pros and cons of both approaches [12]. The complexity of code generation could just move into complexity of interpreters, though we hope not. Code generation may result in highly optimized code and faster execution, whereas interpretation removes the steps of generation and code compilation so that the models can be executed directly as they change, which may result in faster development. It is possible to generate code into several target languages and environments at the same time. Similarly it is possible to write the same interpreters in different languages or run the same on different platforms. Model interpreters just may have been undervalued and should be used more in practice [13]. There are arguments that model interpretation is a superior approach for developing the models themselves [14] and importance of in-IDE interpretation and testing is also being stressed [11].

An interesting area of research is what general-purpose languages are best for what types of model interpreters. Dynamic languages that enable to program themselves during runtime provide facilities for executing some steps of code generation during interpretation itself.

Life has its twists and turns and reincarnations will remain attractive. Not all code generation will be replaced by direct interpretation, but we expect there will be a growing number of various interpreters of different domain-specific modeling languages. If these interpreters will internally use code generation or not may remain their private implementation detail.

References

1. Stahl, T., Völter, M.: Model-Driven Software Development Technology, Engineering, Management. Wiley (2006)
2. Schmidt, D.: Model-Driven Engineering. IEEE Computer 39(2), 25–32 (2006)

3. Kelly, S., Tolvanen, J.P.: Domain-Specific Modeling. Enabling Full Code Generation. John Wiley & Sons, Inc. (2008)
4. Favre, J.M.: Towards a basic theory to model model driven engineering. In: Proceedings of the Workshop on Software Model Engineering, WiSME (2004)
5. Smolik, P.: Mambo Metamodeling Environment, Doctoral Thesis, Brno University of Technology (2006), http://www.mambomde.com/MamboMDE.pdf
6. Cabot, J.: Executable models vs code-generation vs model interpretation (2010), http://modeling-languages.com/executable-models-vs-code-generation-vs-model-interpretation-2/
7. Terry, P.T.: Compilers and Compiler Generators: An Introduction With C++. International Thomson Computer Press (1997)
8. Smolik, P., Tesacek, J.: Data Source Independent XML Data Access. In: Proceedings of Information System Modeling Conference 2000, Rožnov pod Radhoštěm, CZ, MARQ, pp. 17–22 (2000)
9. Kay, M. (ed.): XSL Transformations (XSLT) Version 2.0. W3C Recommendation (January 23, 2007)
10. Boag, S., Chamberlin, D., et al. (eds.): XQuery 1.0: An XML Query Language, 2nd edn. W3C Recommendation (December 14, 2010)
11. Völter, M.: MD*/DSL Best Practices (2011), http://www.voelter.de/data/pub/DSLBestPractices-2011Update.pdf
12. Den Haan, J.: Model Driven Development: Code Generation or Model Interpretation? (2010), http://www.theenterprisearchitect.eu/archive/2010/06/28/model-driven-development-code-generation-or-model-interpretation
13. Völter, M.: Model-Driven Development of DSL Interpreters Using Scala and oAW (2008), http://www.voelter.de/data/presentations/MDInterpreterDevelopment.pdf
14. Chaves, R.: Model interpretation vs. code generation? Both (2010), http://abstratt.com/blog/2010/08/07/model-interpretation-vs-code-generation-both/

A Plug-in Based Approach for UML Model Simulation

Alek Radjenovic, Richard F. Paige, Louis M. Rose, Jim Woodcock, and Steve King

Department of Computer Science, The University of York, United Kingdom
{alek,paige,louis,jim,king}@cs.york.ac.uk

Abstract. Model simulation is a credible approach for model validation, complementary to others such as formal verification and testing. For UML 2.x, model simulations are available for state machines and communication diagrams; alternative finer-grained simulations, e.g., as are supported for Executable UML, are not available without significant effort (e.g., via profiles or model transformations). We present a flexible, plug-in based approach to enhance UML model simulation. We show how an existing simulation tool applicable to UML behavioural models can be extended to support external action language processors. The presented approach paves the way to enrich existing UML-based simulation tools with the ability to simulate external action languages.

1 Introduction

The UML 2.x standard supports modelling of behaviour through a number of mechanisms, including state machines and activity diagrams, and pre– and post-conditions expressed in OCL. In parallel, executable dialects of UML have been developed to support more fine-grained specification of behaviour using *action languages*. As a result, these languages support rich simulation and code generation from models, but they are not directly supported by UML 2.x compliant tools, nor are they based on the same metamodels as UML 2.x. If modellers want to use action languages (and supporting simulators) with UML 2.x models and tools, they either have to acquire a tool that already provides such capabilities (which may not support their exact simulation requirements), or make use of, e.g., model transformations to a different set of languages and supporting tools.

We present a flexible, plug-in based approach for UML model simulation. We show how existing modelling and simulation tools capable of processing UML behavioural models can be extended to support external action language processors, including for their simulation. This is achieved through precisely modelling the interfaces between the modelling tool and an external action language processor, and implemented using a plug-in mechanism. The approach supports enrichment of UML modelling and simulation tools with simulation capability for action languages. The approach even enables addition of *further* action languages to UML tools that already possess one, thus allowing engineers to increase and tailor the simulation support available in the existing tools.

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 328–339, 2012.

This approach has been developed to meet industry requirements, in the context of the European FP7 project INESS, to (i) provide enhanced simulation support for existing railway interlocking models, particularly for safety analysis and validation; and (ii) without requiring new modelling tools to be purchased or developed.

As a result of our work, we have modified an existing UML 2.x modelling and simulation tool in several ways. We have: (i) extended its internal behavioural metamodel to support embedded expressions specified in external languages, (ii) enabled the tool to support a plug-in mechanism, and (iii) extended the tool's functionality with a new set of services compliant with the interfaces mentioned above. Importantly, the operation of the tool in the absence of external plug-ins is identical to the normal operation of the tool prior to the modifications. We have also implemented a plug-in based on an existing action language, and validated our approach using a number of real-world case studies from the railway signalling domain, as we describe later in the paper.

As opposed to existing approaches, we tried to generalise the interactions between a UML simulation tool and an *external* language processor, allowing the capability of existing tools to be augmented without further tool development.

The remainder of the paper is structured as follows. Section 2 presents the background and context. Section 3 presents the overall requirements for tool support (both modelling and simulation), and the mechanisms used to flexibly extend existing tools to support external action languages. Section 4 presents examples of industrial application and describes how the approach exploiting the plug-in based approach was assessed. Section 5 summarises related work, and we analyse the effectiveness of the approach in the conclusions in Section 6.

2 Context and Background

This work was undertaken within the INESS (Integrated European Signalling System), funded by the FP7 programme of the European Union. This industrial project focused on producing a common, integrated, railway signalling system within Europe. Signalling systems are perhaps the most significant part of the railway infrastructure. They are essential for the performance and safety of train operations. A significant number of large UML models produced by the signalling experts using a tool called CASSANDRA [7] (a plug-in for the UML tool Artisan [1]) had been developed. The input models comprise UML class and state machine diagrams and are used to model railway signalling systems and simulate their execution. In addition, the models were enriched with expressions described in CASSANDRA's bespoke action language, SIML. A strict requirement in the project was that modelling tools (Artisan) remain unchanged, for the purposes of validation (engineers were unwilling to change the tools or the way in which they used them, partly because model modification was too expensive).

Engineers also used CASSANDRA to perform simulations, in order to check functional properties and explore safety requirement violations. The simulations

were executed via a Prolog-based engine [8], and as such the simulator was considered to be slow and inefficient by engineers. Additionally, customisation of the simulator was not possible. Requirements for more fine-grained control of simulations were expressed by the industry engineers.

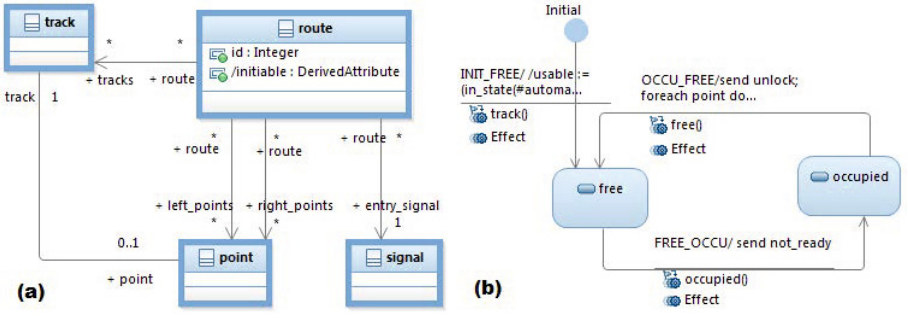


Fig. 1. (a) Class diagram; and (b) State machine for Track class

A simplified excerpt of a class diagram is shown in Fig 1 (a). The UML state machine in Fig. 1 (b) models the behaviour of Track objects. The expressions that follow the transition names, such as '*send not_ready*' are written using SIML [7]. SIML is composed of *four parts* that roughly deal with the following concerns of model simulation:

- *declaration* – allows users to define basic elements, typically corresponding to the UML model elements (classes, events, etc.) that can be read directly from the input model *only if CASSANDRA is used*; also, define elements not present in the original model (e.g. inputs from the environment)
- *expression* – assists users in building and evaluating complex expressions (classified according to the data type) formed from multiple elements
- *action* – provides mechanisms to define elementary pieces of behaviour (e.g. creation/deletion of instances and association links); invocation of behaviours defined in the source model (e.g. class operations or transition triggers)
- *control* – allows users to combine elementary actions into ordered sequences and iterations

Engineers working on INESS had created a substantial number of railway signalling models using the CASSANDRA extensions to Artisan. The simulation capabilities of CASSANDRA did not provide adequate performance or scalability to sufficiently validate the models, explore them, and provide assurance that safety properties were met. As a result, new requirements for simulation were specified. In particular, it was mandated that engineers would still be able to use CASSANDRA (and Artisan), that the existing action language would still be supported, but external simulators (that were more efficient, more scalable, or performed better) that would also simulate the action language could be exploited.

To satisfy the industrial requirements to provide richer and higher performance simulation capabilities while still retaining use of CASSANDRA/Artisan, we have developed a plug-in based approach, detailed in the next section. Our development so far has connected a specific external language processor and a simulation tool as proof-of-concept to the railway signalling engineers. The external processor is capable of parsing an action language and making requests into the simulation tool's API (e.g. to create objects, or fire events), but has awareness of the global clock, tasks, message queues, and other standard simulation tool's resources. The simulation tool is based on the SMILE platform [15,14]. The tool's architecture is shown in Fig. 2, and its operation is best described using the following scenario.

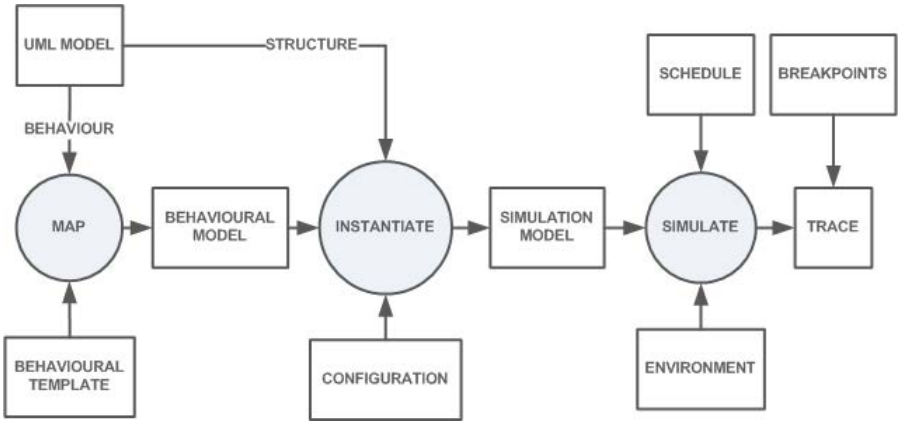


Fig. 2. Simulation tool architecture

The input UML model (created by any UML compatible tool) is first queried (as explained in [15]) in order to extract the behavioural information from state diagrams (states, transitions, triggers, etc.). This is then mapped to a state machine *behavioural template* defined in one of the SMILE family of languages to produce a set of types which describe only the behaviour of UML model components. The set of types is stored in a *behavioural model*. In parallel, the input model is also queried to generate structural information (e.g. from class diagrams). The next step involves using manual *configuration* (supported graphically by the tool) to create a *simulation model*. The simulation model is a set of instances of types from the behavioural model that also take into consideration the structural hierarchy obtained in the previous step.

The simulation tool supports concurrency in the form of tasks which provide execution separation. Consequently, in the *configuration* step, users may define multiple tasks and map each simulation object to one of them. After selecting a *scheduling mode*, the simulation can be run. Optionally, users can define the system's environment in the form of one or more stimuli. The tool produces a

simulation trace, providing a detailed report on events that occurred during the simulation (e.g. triggers, transitions, message queues, or unsatisfied conditions).

3 The Plug-in Based Approach

The existing tool could simulate basic UML behaviours (described by standard UML behaviour diagrams). More fine-grained behaviour descriptions could only be provided using external action language expressions. Thus, the key objective was to provide a flexible plug-in mechanism to allow connection and interoperation between the tool and the external language processor, and to allow us to detach, disable or replace the new simulation capability when not needed. The work was divided into the following packages:

1. Inclusion of *external* (SIML) behavioural data from the input model into the *simulation models* (Fig. 2), ensuring additional data in the simulation model is ignored by default, avoiding disruption of the existing tool operation.
2. Enhancement of the simulation tool so that it provides a mechanism for plug-ins capable of processing expressions in an action language
3. The implementation of the plug-in for the SIML action language
4. Evaluation by performing simulations and verifying model correctness
5. Generalising and formalising the interface(s) between the tool and the language plug-ins, enabling usage with a variety of (Executable UML) notations

3.1 Extensions to the Behavioural Metamodel

Behavioural templates are used to create *behavioural types* based on information extracted from a source UML model. Fig. 3 shows an extended metamodel for the types created based on the behavioural template for state machines. We observe that the behavioural types are composed of one or more *properties* (e.g. states) and *transitions*, which in turn comprise a *trigger* and one or more *conditions* (guards) and *actions*. The existing behavioural type metamodel was extended with the `External` class to enable the inclusion of the embedded action language expressions from the source model into the simulation. This class defines three attributes: *language* – to signify which action language is used, *classifier* – to describe which particular part of the behaviour the expression applies to (e.g. *effect*, for transitions; *exit* or *entry*, for a state change), and a *body* – which contains (external notation) expressions unknown to the simulation tool.

3.2 The Simulation Tool Plug-in Mechanism

To allow for an arbitrary action language to be used on the core UML models, a plug-in mechanism has been chosen. This choice ensures that there will be no need to modify the simulation tool to accommodate different kinds of executable UML. The loading of plug-ins (one per project) is dynamic. (A project includes input models, queries, a behavioural template, configuration files, schedule, breakpoints, etc.). The mechanism's implementation details are specific to the implementation platform and are not relevant here.

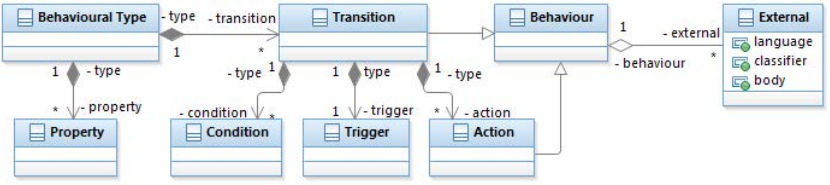


Fig. 3. Extended behavioural type metamodel

3.3 The SIML Plug-in

The SIML plug-in is the SIML-specific implementation of the external language processor interface, `IPlugin` (Fig. 4. (a)), and was developed incrementally, adding the functionality found in SIML expressions in a gradual manner.

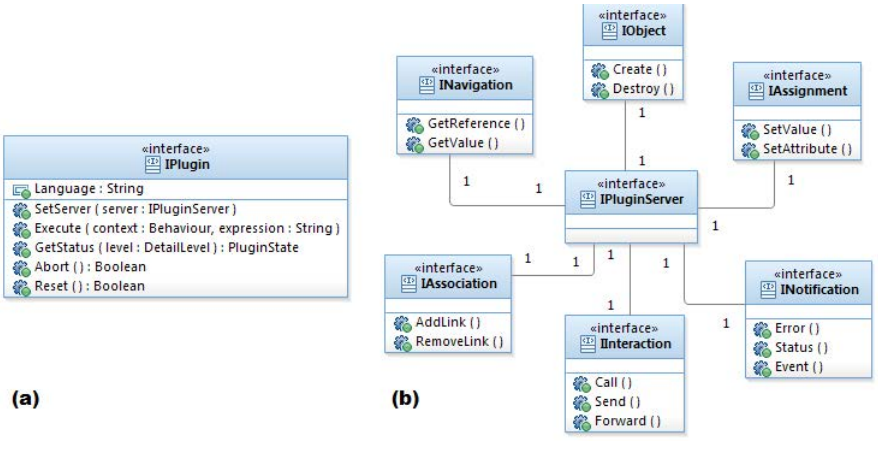


Fig. 4. (a)IPlugin and (b) IPluginServer interfaces

The `SetServer` method is called when the plug-in is loaded. The simulation tool then uses the `Language` attribute in order to determine which action language it is capable of processing. The `SetServer` method passes a reference to the simulation tool's component that is the implementation of the `IPluginServer` interface (described in the next section). When the `SetServer` call is made, the plug-in resets its internal state to the default initial state. The tool can also force this operation by using the Reset method. The `Execute` method is called when the plug-in is required to process an action language expression. This method takes two parameters: `context` – informing the plug-in which (behavioural) object is requesting the processing, and `expression` – the action language expression that needs to be processed. For debugging, management or reporting purposes, the tool can also request a snapshot of the plug-in's internal

state through the `GetStatus` method, as well as issue an `Abort` request as a safety mechanism to abandon further proecssing.

3.4 The Simulation Tool Plug-in API

`IPluginServer` essentially describes an API (a set of simulation tool's services that `IPlugin`–compatible plug-ins can use). The API is described by the `IPluginServer` interface (Fig. 4. (b)), and consists of the following (sub-) interfaces grouped by the functionality:

- **navigation** – providing access to model elements
- **object** – required to create and destroy objects
- **assignment** – required to set values to object references and to allocate values to objects' attributes
- **association** – required to create and remove relationships between objects
- **interaction** – providing mechanisms to pass data between objects in synchronous and asynchronous manner
- **notification** – asynchronous mechanism for providing management and operational notifications from the plug-in to the tool

Basic methods associated with each section are also shown in Fig. 4(b). With the exception of those in the `INotification` interface, these methods represent a minimum set of services required to 'drive' a state-machine based simulation scenario from an executable UML plug-in.

3.5 Normal Operation, Control, and Exceptions

The simulation tool was also required to implement a mechanism to control the overall simulation execution in the presence of an external plug-in (e.g. in order to avoid deadlock scenarios, either because of a faulty or partial plug-in implementation, or an incomplete/erroneous behavioural specification in the input model). The tool and the plug-in operate collaboratively during the simulation runs, using a master-slave mode of operation. The simulation tool is a passive master of the workflow, at times relinquishing control to the plug-in in full, but continually monitoring the execution and intervening in case of a problem by: disabling the plug-in, aborting the simulation, and reporting back to the user. In a nutshell, a simulation run goes through the following stages and sub-stages:

- **Power-up** – initialisation of tool and plug-in internal states and variables
- **System execution** – composed of *simulation steps*; in each step, the tool sequentially executes all system tasks; this stage is repeated until either (a) the simulation runs its natural course where all simulation objects have become idle, (b) the users halts the execution, either manually or through a breakpoint, (c) an error is reported by a plug-in, or (d) the plug-in has become unresponsive
  - **Task execution** – task's simulation objects are executed sequentially; if an object's message queue is empty or blocked, the object is skipped

    - **Object execution** – executing object's current behaviour (e.g. a transition in a state machine scenario, comprising multiple actions)
- **Power-down** – in this stage, the simulation trace may be logged, or perhaps a report generated if particular analysis is required

The above execution flow is controlled by the tool. The control is partially relinquished to the plug-in at the *object execution* level, if and when an external action language expression is read from the object's message queue. Before doing so, the tool sets a timer to guard against situations when the plug-in blocks further operation of the tool. If the control is returned to the tool before the time-out, the timer is cancelled; if not, the plug-in is forcefully disabled, the simulation is aborted, and feedback is given to the user.

4 Industrial Application and Assessment

The work presented is in response to strong industrial demands to provide enhanced simulation support for railway interlocking models, particularly for safety analysis and validation, without requiring new modelling tools to be purchased or developed. The enhanced control of simulations is achieved through external action languages that allow a more fine-grained specification of behaviours in UML models. We have demonstrated how to extend a simulation tool's capabilities to exploit external action languages, achieved through a dynamic integration of the tool and the external module using the plug-in mechanism. We have also formalised the interaction between these two by precisely modelling their interfaces.

Following the implementation of the extension to the tool and the SIML plug-in, we have used the following criteria in order to validate our approach, by verifying the tool operation:

- *without a plug-in* – in the absence of a plug-in, the tool's operation must be equivalent to that prior to the modifications made
- *with a plug-in* – in the presence of a relevant language plug-in, simulation is either fully automatic or hybrid (largely driven by the executable UML expressions embedded in the model, but manual user input is also supported)
- *with a malfunctioning plug-in* – the tool must prevent the plug-in from blocking the simulation indefinitely by disabling the plug-in under such circumstances, aborting the execution, and providing the feedback to the user

All scenarios above were tested and the criteria satisfied. In particular, the 'malfunctioning plug-in' was repeatedly tested during the incremental development; during this period, when the plug-in's functionality was only partially implemented, blocking situations were in abundance. The verification of the normal operation of the tool (without a plug-in) was compared against the unmodified version of the tool on the same set of case studies. Finally, the hybrid operation was tested by disabling selected functionality in the plug-in (e.g. sending a message) and replacing it by issuing a prompt to provide a manual user input.

The fully-automatic operation was tested on a number of case studies from the railway signalling domain, the smallest containing 7 classes and 25 SIML statements, and the largest 89 classes and several hundred SIML statements. Every class had its own state machine defined. Simulation runs were performed on all case studies, validating the approach and the tool/plug-in operation. Due to project's time constraints, it was not possible to fully verify the correctness ([2]) of the larger models. The modified tool outperformed Artisan-CASSANDRA combination, if only just – possibly attributed to the 'lightweightedness' of the tool, but because of its prototype status, further improvements are expected. The real power of the approach, however, lies in its extensibility and generalisation. The SIML plug-in was implemented in 3 weeks, but after the initial learning curve, we anticipate the future plug-in development to shrink to several days.

The following are typical scenarios in which the integrated solution (the simulation tool and the plug-in) was (can be) used:

- *normal execution* – performing a simulation run according to the specification derived from the source models. Two most typical outcomes are: (a) execution runs its natural course (i.e. no further activity detected), a snapshot of the current state of the system is taken, then analysed together with the simulation trace; (b) simulation runs indefinitely; user forcefully halts the execution and analyses trace.
- *property checking* – checking if specified property is satisfied. We use conditions (Boolean expressions composed of model elements, attributes, and Boolean operators) specified in breakpoints to signify properties. A breakpoint causes the simulation to pause (or come to a halt) when its condition evaluates to true.
- *error injection* – analysing the robustness of the modelled system, i.e. its behaviour in the presence of errors. Errors are purposely introduced into the system through user input.

Property checking and error injection are illustrated using the example in Fig. 5. Two routes R1 and R2 (comprising tracks T, signals S, and points P) are defined as:

```
R1 = {S1; T1,P1(left),T3} and R2 = {S1; T1,P1(right),T2}
```

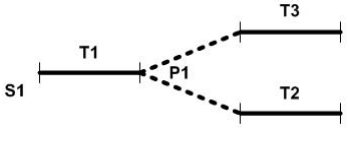


Fig. 5. An example railway interlocking model used for assessing the approach

We observe that that the routes share elements such as the entry signal S1 or track T1. When specifying properties, we use *negative application conditions* (NACs) (e.g. to verify that a system is safe, we define conditions that make

the system unsafe). For instance, we may want to verify that signal S1 is not in the `proceed` state if, at the same time, track T1 is in the `occupied` state. Thus the property that we want to check is defined using the Boolean expression:

```
(S1.state == 'proceed') AND (T1.state == 'occupied')
```

specified as a breakpoint condition. If, during a simulation run, the expression evaluates to *true*, the execution of the model is stopped. The user is then able to see which breakpoint triggered this, and can analyse the trace to find the design fault.

We can also deliberately inject errors into the system's behaviour. Using the same scenario, we can define a different breakpoint, as follows:

```
(R1.state == 'active') AND (R2.state == 'active') AND
(T1.state == 'occupied')
```

Routes essentially have two main states: *idle* and *active*. Since R1 and R2 share a common track, they cannot be both *active* at the same time when T1 is *occupied*. During the simulation, for instance, when R1 is active and T1 occupied, we can (through user input) change R2's state to 'active' and observe how the system behaves afterwards. This method works well; however, the simulation clock has to be slowed down substantially in order to inject errors at the right time. Work is in progress to provide a mechanism to use rule-based scripts for error injection.

5 Related Work

Shlaer and Mellor [18,19,17] developed a method in which objects are given precise behaviour specifications using state machine models annotated with a high-level action language. In the late 1990s, OMG started working on Action Semantics for the UML with an objective to extend the UML with a compatible mechanism for specifying action semantics in a platform-independent manner.

There are a number of research and academic platforms and tools that attempt to define behaviours in UML models more precisely, and to execute such enriched models [10,6,16]. There are also several commercial tools that define their own semantics for model execution [7,5,4,9,3]. They often include a proprietary (action) language. Consequently, models developed with different tools cannot be easily interchanged and cannot interoperate.

The Foundational UML (fUML) [12] specification (adopted in 2008) provided the first precise operational and base semantics for a subset of UML encompassing most object-oriented and activity modelling. fUML, however, does not provide a new concrete *surface* syntax; rather, it ties the precise semantics solely to the existing abstract syntax model of UML. Subsequently, OMG issued an RFP for Concrete Syntax for a UML Action Language. Currently, what is known as the Action Language for fUML (or Alf) [11] is in beta 2 phase (since 2010).

Alf is a textual surface representation for UML behaviours. Expressions in Alf can be attached to a UML model in any place where a UML behaviour

can be. Any Alf text that can be mapped to fUML can be reduced to a set of statements in first-order logic. Unfortunately, it does not allow us to use model-checking features or a theorem prover for validation or verification. In that respect, significant additional work remains in order to provide a complete formal verification for a system [13]. This is exactly why simulation in combination with Alf (or indeed any other executable UML language) is beneficial. Simulations do not provide formal proofs; nevertheless, they can significantly increase confidence in the tested models.

There is one other significant gap. Although we now have the action semantics (in fUML) and are close to seeing a fully adopted concrete syntax (in Alf), the question of *how* the models will be executed still remains. In other words, there are currently no attempts to standardise or even specify a simulation environment for UML that would define how such platform should connect to Alf or another (proprietary) action language. In the meantime, we should do as best as we can with what is available and try to adapt the existing UML compatible simulation tools for use with diverse executable UML notations. This paper contributes to this objective.

6 Conclusion

Model simulation is increasingly seen as a reliable complementary approach to formal verification or testing that attempts to establish the validity of model behaviours. Standard UML's limitation to describe these behaviours in greater detail is addressed by a number of action languages. Although the majority of these languages were designed following the same or similar philosophy, there is no common metamodel from which they can be derived. This presents a significant challenge for UML tool makers if they were to support a number of such languages in their tools.

We have tried to address this problem and have proposed a modular approach to UML model execution through simulation. We have demonstrated a scenario in which a UML-based simulation tool is extended in a non-disruptive, modular, manner to accommodate multiple action language 'engines' to collaborate with the tool during simulation runs. This was achieved by augmenting the tool's capability using a simple plug-in paradigm. We have formalised the solution by defining the interfaces required by the plug-in and the tool so that the two can be integrated. We have also illustrated the normal operation, the workflow control, as well as the exception handling mechanisms. We have evaluated our approach on examples from the railway signalling domain. The tool we used in our study was an in-house tool (based on the SMILE platform), while the language was CASSANDRA's action language SIML.

Our next steps will include investigation into more action languages, other simulation platforms, as well as further generalisation and formalisation of the approach. We anticipate an increased activity in the model simulation arena and are also mindful of the fact that new domain-specific action languages may emerge. As part of our desire for a broader research impact, our intention is to standardise the interaction between the simulation tools and action languages.

References

1. Atego. Artisan Studio (2011), http://www.atego.com/products/artisan-studio/
2. dos Santos, O.M., Woodcock, J., Paige, R.F., King, S.: The Use of Model Transformation in the INESS Project. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) FMCO 2009. LNCS, vol. 6286, pp. 147–165. Springer, Heidelberg (2010)
3. Dotan, D., Kirshin, A.: Debugging and Testing Behavioral UML Models. In: Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion (OOPSLA 2007), pp. 838–839 (2007)
4. IBM, Rational Rhapsody (2012), www.ibm.com/software/awdtools/rhapsody/
5. IBM, Rational Software Architect RealTime Edition (RSA–RTE) (2012), http://www.ibm.com/software/rational/products/swarchitect/
6. Jiang, K., Zhang, L., Miyake, S.: An Executable UML with OCL-based Action Semantics Language. In: 14th Asia-Pacific Software Engineering Conference (APSEC 2007), pp. 302–309 (December 2007)
7. Know Gravity. CASSANDRA (2011), http://www.knowgravity.com/eng/value/cassandra.htm
8. Mellish, C.S., Clocksin, W.F.: Programming in Prolog: Using the ISO Standard. Springer (2003)
9. Mentor Graphics. BridgePoint (2012)
10. Mooney, J., Sarjoughia, H.: A Framework for Executable UML Models. In: 2009 Spring Simulation Multiconference. Society for Computer Simulation International (2009)
11. OMG. Action Language for Foundational UML (Alf). Technical Report October 2010, OMG (2011)
12. OMG. Semantics of a Foundational Subset for Executable UML Models (fUML), v1.0. Technical Report, OMG (February 2011)
13. Perseil, I.: ALF formal. Innovations in Systems and Software Engineering 7(4), 325–326 (2011)
14. Radjenovic, A., Paige, R.F.: Behavioural Interoperability to Support Model-Driven Systems Integration. In: 1st Workshop on Model Driven Interoperability (MDI 2010), at MODELS 2010, Oslo, Norway. ACM Press (2010)
15. Radjenovic, A., Paige, R.F.: An Approach for Model Querying-by-Example Applied to Multi-Paradigm Models. In: 5th International Workshop on Multi-Paradigm Modelling (MPM 2011), at MODELS 2011. ECEASST, vol. 42, pp. 1–12 (2011)
16. Risco-Martín, J.L., de La Cruz, J.M., Mittal, S., Zeigler, B.P.: eUDEVS: Executable UML with DEVS Theory of Modeling and Simulation. Simulation 85(11-12), 750–777 (2009)
17. Shlaer, S., Mellor, S.J.: Object-Oriented Systems Analysis: Modeling the World in Data. Prentice Hall (1988)
18. Shlaer, S., Mellor, S.J.: Recursive Design. Computer Language 7(3) (1990)
19. Shlaer, S., Mellor, S.J.: Object Lifecycles: Modeling the World in States. Prentice Hall (1992)

MADES: A Tool Chain for Automated Verification of UML Models of Embedded Systems

Alek Radjenovic[1], Nicholas Matragkas[1], Richard F. Paige[1], Matteo Rossi[2], Alfredo Motta[2], Luciano Baresi[2], and Dimitrios S. Kolovos[1]

[1] Department of Computer Science, The University of York, United Kingdom
{alek,nikos,paige,dkolovos}@cs.york.ac.uk
[2] Politecnico di Milano, Italy
{rossi,motta,baresi}@elet.polimi.it

Abstract. The benefits of Model Driven Development may be achieved through exploitation of its potential for automation. Automated model verification is one of the most important examples of this. The usage of automated model verification in everyday software engineering practice is far from widespread. One of the reasons for this is that model designers do not have the necessary background in mathematical methods. An approach where model designers can remain working in their domain while the verification is performed on demand, automatically and transparently, is desirable. We present one such approach using a tool chain built atop mature, popular and widespread technologies. Our approach was verified on industrial experiments from the embedded systems domain in the fields of avionics and surveillance.

1 Introduction

Our research project – MADES (*Model–based methods and tools for Avionics and surveillance embeddeD SystEmS*) [16] – was born out of demand from the embedded systems industry to develop a model–driven approach to improve current practice in the development of embedded systems. This approach was to cover all phases, from design to code generation and deployment. Many embedded software systems, and particularly those from the avionics and surveillance domains, require high integrity, where verification is essential before deployment. Verifying properties of the system at the start of the development is highly desirable as the first line of defence against design faults which if detected at a later stage are very costly. At the same time, system verification at the model level is hard because model designers typically do not possess the necessary mathematical background. In this paper we present a tool chain that allows system designers to perform model verification on demand, automatically and transparently (without the need to understand the complexities of the mathematical formalisms that underpin the approach).

A model designer and a formal methods expert focus on significantly different things. Even the tools and techniques they use are at the opposite end of the

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 340–351, 2012.

software engineering spectrum. The former typically uses graphical modelling tools, whilst the latter uses textual notations that rely heavily on mathematics. Inadequate knowledge of model checking techniques can limit the scope of the types of verifications that can be performed. On the other hand, inadequate knowledge of the system under development can weaken the validity of the verification itself. The majority of research efforts have been on low–level detail, such as: how can we translate an OCL constraint into a mathematically sound form that we can use with a SAT solver? Or, how can we define correctness properties for UML class diagrams?

This paper addresses the big picture: how can we make the verification process more practical? How can we provide the model designers with a tool that enables them to automatically check the correctness of their models without the need to understand model checking? The solution we provide is a tool chain – a set of domain–specific tools communicating with each other. Two of the key traits of a successful tool chain are: (i) *transformational capabilities* – e.g., to transform models into transformation scripts and (ii) *usability* – a tool's ability to allow the user to specify another domain's documents using concepts from its own domain. Consequently, the work described in this paper focuses on providing one such tool chain that enables model designers to utilise mathematical methods 'under the hood'.

We were able to use and combine several existing and mature technologies. On the verification side, we build on current model–checking technology to provide decision procedures more specifically tailored to the project domain. By exploiting domain abstractions and model fragments, the designers are allowed to define properties in a way close to their domains that hides the formalism. An enabling technology that underpins our framework is model transformation. We provide support via Epsilon [15] for various kinds of model transformations. The transformations support the verification tasks by allowing platform models (e.g., in subsets of UML/MARTE [19]) to be mapped to verification technology, such as Zot [24] or Alloy [17].

Our solution provides model designers with the ability to verify their design without the need to understand underlying formalisms. It allows on–demand and automatic verification at *any* stage of the development process. Our approach works with the entire system model, a segment of it, or even a partial model implementation. In case of verification failures, counter examples are provided with the ability to trace back to the source of errors. Our solution is practical, usable, reusable, generic, and underpinned by proven mature technologies. Importantly, it is in direct response to specific industrial requirements.

The remainder of the paper is structured as follows. Section 2 presents the background and context. Section 3 summarises the related work. Sections 4 and 5 present the overview of the approach, and the details of the implementation. Section 6 presents an example of industrial application and describes how the approach was assessed. We conclude by analysing the effectiveness of the approach in Section 7.

2 Background

The key ambition of the MADES project [2] is to develop a model–driven approach to improve the current practice in the development of embedded systems. The proposed approach is holistic in that it covers all phases of the development life–cycle, from design to code generation and deployment.

MADES makes several key contributions. Firstly, a dedicated (MADES) language was developed as an extension to OMG's MARTE Profile [19]. Secondly, approaches have been developed for verification of key properties on designed artefacts, closed-loop simulation based on detailed models of the environment, and the verification of designed transformations. And thirdly, code generation techniques have been devised which addresses both hardware description languages as well as conventional programming languages, with features for compile–time virtualisation of common hardware architecture features, including accelerators, memory, multiprocessor and inter–processor communication channels, to cope with the fact that hardware platforms are getting more and more complex.

Our work is part of the validation effort in which we use model transformations to generate various software artefacts from the MADES models. The documents include verification scripts, simulation scripts, hardware architecture descriptions, architecture agnostic source code, software and hardware mappings for compile–time virtualisation, and hardware architecture descriptions for compile–time virtualisation. The model transformation work is in direct response to the high level requirements to achieve tool interoperability, code generation, and traceability of the model–based activities of the development life–cycle of embedded systems.

3 Related Work

3.1 Model Checking

Over three decades ago, two seminal papers [25], [9] founded what has become the highly successful field of *model checking*, for automatically assessing whether a system model satisfies specified properties. In recent times, as model driven development (MDD) became more widespread, model verification has come at the forefront of research in this arena. Model verification can take many different forms, including formal (mathematical) analyses such as performance analysis based on queuing theory or safety–and–liveness property checking. Very often, it means executing models on a computer as an empirical approach to verification [27].

Holzmann [13] states that in the classic approach to logic model checking, verification requires a manually constructed model to be written in the language that is accepted by the model checker. The construction of such a model typically requires good knowledge of both the application being verified and of the capabilities of the model checker that is used for the verification.

Schmidt [26] points out that traditionally model checking has been performed very late in the development (in the testing phase), though he argues for the

necessity to be able to do this at any stage of the development lifecycle. Moreover, now that many of the verification technologies have matured, there is an emergent need for verification to be usable in practice [6].

Unsurprisingly, because of its widespread usage, the focus of attention in recent years has been the Unified Modeling Language (UML) [20,21]. A lot of research activity has been around OCL (Object Constraint Language) [23] used in UML specifications. Some examples include [6,7,5,8,12,28,29]. Despite some insightful approaches, the majority of these valuable contributions focus solely on the structural aspects (namely, UML Class Diagrams).

In "Verified Software: A Grand Challenge" [14], Jones et al. point out that formal methods used in verification are intended to predict software behaviour. A verification method for UML cannot therefore be complete if it does not include behaviours described, for instance, through UML state machine or sequence diagrams. Approaches such as [31] which uses formal analysis on concurrent systems specified by collections of UML state machines, or [4] that deals with automatic translation of statecharts and sequence diagrams into generalized stochastic Petri nets, are a steps in the right direction.

3.2 Model Transformation

Model transformation plays a key role in model driven development. Although there is still no mature foundation for specifying transformations among models [10], there are many worthwhile theoretical approaches as well as several that are practical, too.

One of the platforms that is at the forefront of model transformations, not only due to its maturity, but more importantly in terms of its usability, is Epsilon [15] a family of interoperable task specific languages for interaction with EMF (Eclipse Modeling Framework) [11] models. In particular, the Epsilon Transformation Language (ETL), a hybrid, rule–based language, provides not only the usual transformation features but can be used to query, navigate, or modify both source and target models. ETL can transform many input to many output models. In addition, the Epsilon Generation Language (EGL) is typically used hand–in–hand with ETL for model–to–text transformations (e.g. translating UML models into Java code).

The enabling technology that helps us achieve full automation in model driven system verification is model transformation.

4 Approach/Framework

As stated earlier, the overall objective of the MADES project is to improve the model–driven design for embedded platforms. The overview of the various artefacts in the MADES approach can be found in [1]. Verification of system properties at different phases of the development process plays a key role.

Fundamental to this is reducing the overall effort associated with the verification process. One way to achieve this is to hide the complexity of the formal

models from the domain experts and allow them to specify the system of interest in a notation they are familiar with. To this end, the MADES approach uses model transformations to provide a seamless integration between design tools and the verification tools. We have accomplished this by means of a tool chain whose workflow has three main stages. In *Modelling*, a modelling tool (Modelio [30]) is used to generate design models using the MADES modelling language (a combined subset of OMG MARTE [19] a UML profile for modelling real–time embedded systems and SysML [18] – a general–purpose modelling language for systems engineering applications). These models serve as the input to the next stage – the *Transformation*. In this stage, Java code is generated, instantiating objects needed for the verification platform. The final stage – *Verification* – uses the objects from the previous stage to produce verification scripts (containing lists of temporal logic formulae) as an input to the Zot tool [22] that performs formal proofs.

Most importantly, users interact only with the modelling tool. The transformation and the verification take place 'under the hood', transparently to the users, who do not have to deal with anything beyond their domain, making this a true MDE approach.

5 Implementation

5.1 Modelling

The proposed approach is model–driven. Hence, the verification process is entirely guided by the MADES design model, comprising a set of mandatory and optional UML/MARTE diagrams, including:

- *Class diagrams* – besides standard UML features, these may define MARTE clock types (CT) that constrain the temporal behaviour of components (e.g. associating a CT with a class is equivalent to declaring clock instance associated with all the objects of that class)
- *Object diagrams* – contain class and clock instances declared in class diagrams
- *State diagrams* – describe the state–based behaviour of system objects. These include standard features like states, transitions, triggers, guards and actions
- *Sequence diagrams* – describe partial behaviours in the system, capturing message (class operation instance) exchanges between objects (as defined in the Object Diagram). Time constraints (capturing metric timing relationships between events in the diagram) can be also added using MARTE stereotypes
- *Interaction Overview Diagrams* – provide a high–level structuring mechanism to compose sequence diagrams through common operators such as *sequence*, *iteration*, *concurrency*, or *choice* (this compositional solution differs from the one used in scenario–based approaches, because sequence diagrams are not used to render valid, invalid, or contingent traces, but rather to describe portions of the behaviour of the system)

Finally, diagrams share a common set of events such as interrupts, beginnings and ends of messages, or clock ticks, enabling different diagrams (system views) to communicate with each other.

5.2 Tool Chain

In real–time embedded systems, the most intuitive way to describe operational behaviours has the following main characteristics: (i) time is explicitly represented in the diagram, (ii) the timing constraints can be represented directly in the diagram, and (iii) the level of abstraction can be easily refined through the development process. For example, in this context, the sequence diagrams represent what is performed by different functional blocks in a time–based visualisation, and they can be used as a starting point to determine if various timing properties of interest remain valid during the development process. Some examples of timing properties are:

- Is the system able to complete *Task X* within t time units?
- Does *Event E* always precede *Event F*?
- If *Event E* occurs, will *Event F* occur within t time units?

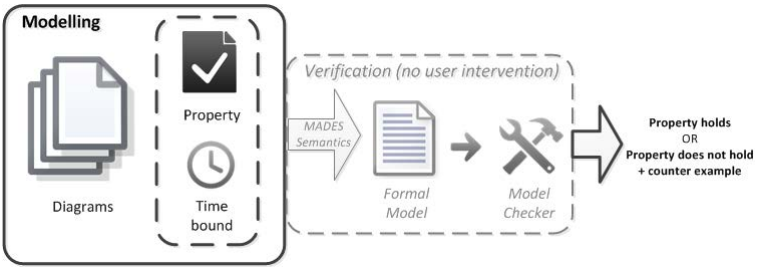


Fig. 1. Verification workflow

The MADES verification workflow (Figure 1) comprises a modelling stage and a verification stage (that requires no user intervention). As stated earlier, the entire verification process is fully automatic. However, minor user input is required as explained next. During the modelling stage, in addition to the UML diagrams, the user defines a *property* to be verified, as well as and the *time bound*. In MADES approach, this is achieved through a user–friendly interface (Figure 2), where users can also choose a SAT solver and a model checker they wish to use. The current implementation model checker is Zot, but any model checker that supports TRIO temporal logic can be used instead.

Once the models, the property to be checked, and the time bound are defined, the user can automatically run the verification and check whether the property is satisfied or not. If the property is not satisfied, the model checker generates

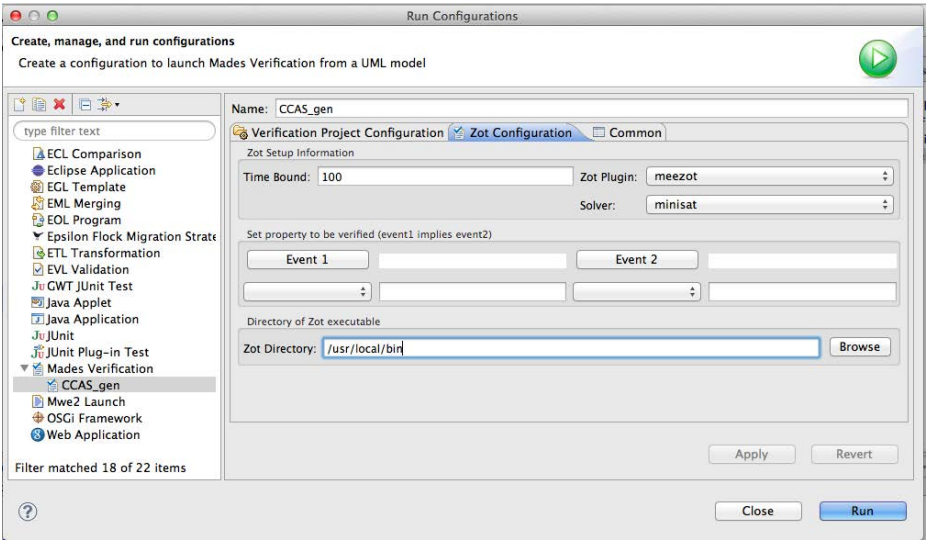


Fig. 2. Property and time bound configuration

a counter–example showing a system execution that violates the property. An integrated traceability mechanism allows us to trace back from the elements in the counter–example to the original UML model using trace links that are visualised in a dedicated editor. As it is apparent from this workflow, the user does not need to be familiar with temporal logic formulas to be able to formally verify the design models.

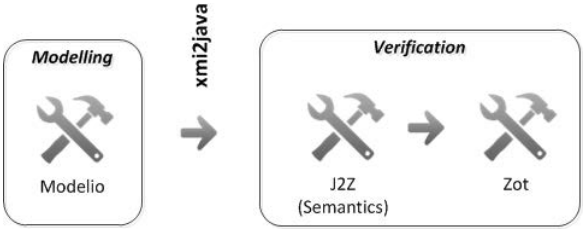


Fig. 3. MADES verification tool chain

MADES verification tool chain is shown in Figure 3. The generated models (using an XMI–compliant CASE tool, such as *Modelio* or *Papyrus*) are transformed (xmi2java) to Java code that acts as an input to the Java–to–Zot tool (*J2Z*). J2Z then produces a corresponding set of TRIO formulae that are fed into the verification engine (*Zot*). Zot encodes satisfiability (SAT) and validity problems for discrete time TRIO temporal logic formulae as propositional SAT problems, which are in turn checked with off–the–shelf SAT solvers. The TRIO

formulae define the semantic representation and provide a formal semantics to MADES diagrams [3].

The code snippet below illustrates the Java primitives associated with an Interaction Overview Diagram, using the API provided by J2Z. The bottom part shows the commands to start generating the semantic representation written in Lisp for Zot.

```
...
//IOD declaration
core.diagrams.IOD CCAS_IOD=new IOD();
iod.Node i_n1= new InitialNode();
iod.Node i_n2= new Merge();
iod.Node i_n3= new InterruptNode(BrakeInterrupt);
iod.Node i_n4= new FlowFinalNode();
CCAS_IOD.addControlFlow(i_n1,i_n2);
CCAS_IOD.addControlFlow(i_n2,SDSendSensorDistance);
CCAS_IOD.addControlFlow(SDSendSensorDistance,i_n2);
CCAS_IOD.addControlFlow(SDSendBrakeCommand,i_n4);
CCAS_IOD.addControlFlow(i_n3,SDSendBrakeCommand);
//J2Z Wrap--up
MadesModel madesModel=new MadesModel();
//[...] Add diagrams to the MADES UML model
madesModel.addIod(CCAS_IOD);
//ZOT Configuration
ZOTConf zot=new ZOTConf(100, ‘‘meezot’’, ‘‘minisat’’, madesModel);
zot.writeZOTFile(‘‘CCAS_Verification.zot’’);
```

5.3 Transformation

The transformation of XMI files (the output of the modelling stage) into Java code (the input to the verification stage) is performed using the `xmi2Java` script (Figure 3) that uses the model transformation technology in order to convert various relevant portions of the input model into Java code. The transformations are specified using ETL (Epsilon Transformation Language) and the output code is generated in EGL (Epsilon Generation Language).

6 Industrial Application and Assessment

The operation of the MADES verification tool chain is illustrated using a real-world case study from the real-time embedded systems domain (the automotive industry) - a *Car Collision Avoidance System* (CCAS). One of the key functions of CCAS is to detect the position of the car on which it is installed relative to other objects in its environment (such as other vehicles or pedestrians). The distance between the car and the external objects is read through the on-board *Radar* that sends data to the *Controller* every 100ms via the system bus (CAN).

The full CCAS specification includes a large number of properties for timing (e.g. data transfer), safety (e.g. when to brake if in a critical state) or liveness. Using the MADES modelling language, CCAS is described with one class diagram, one object diagram, one interaction overview diagram, two sequence diagrams, and three state diagrams. Examples of these are provided in Figure 4 and Figure 5.

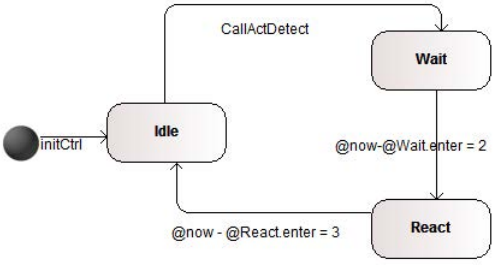


Fig. 4. One of CCAS state diagrams

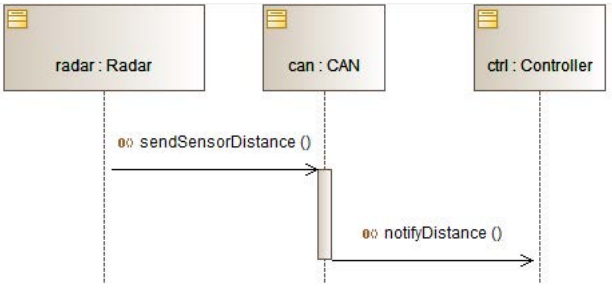


Fig. 5. An example sequence diagrams in CCAS

Consider the following safety requirement as an example of property checking:

> *"If the sensor measuring the distance between the car and the external objects continuously reads a value that is less than 2 meters for a period of 50 time units, then the system should brake within those same 50 time units."*

This requirement can be expressed as a property formalised by the following TRIO formula:

$$\texttt{Alw(Lasted([distance<2],50)} \rightarrow \texttt{WithinP(brakeS braking,50))} \quad (1)$$

where `[distance<2]` represents a Boolean predicate. The formula did not initially hold, and the Zot tool produced a counter-example within around 30 seconds (we used the `meezot` plugin and the `minisat` SAT solver, and the time

bound was set to 100). The counter-example showed that the violation of the property was the consequence of a cumulative delay composed of the delay related to the *radar* component (governed by a logical clock with a period of 10 time units), and the transmission and reaction delays that are present throughout the model. By modifying the time bounds in above formula we can show that, by giving the system a little more leeway in the required reaction time, we can make the property hold. Specifically, the time bounds in formula were changed from 50 to 56 time units. By repeating the verification with the new values, the Zot tool reported (within 20 seconds) the UNSAT result. This is interpreted as: "*it is not possible to find a system trace that violates the property, hence the property holds for the system*".

For reference, the output is produced in two steps: (i) initially, Zot shows the Boolean predicates used to represent the UML model in temporal logic, and (ii) *z3* (a well-known and efficient SMT solver by Microsoft Research) solves the satisfiability problem and reports the UNSAT result.

This simple example illustrates how the MADES tool chain enables modellers to conduct their own experiments, detect and better understand design faults, and improve the system models.

7 Conclusion

This work was done within the MADES project in response to a strong and specific demand from the real–time embedded (avionics and surveillance) industry to develop a holistic, model–driven approach to improve the current practice in the development of hardware and software systems. By holistic we mean encompassing all stages of the development life–cycle, from design to code generation, testing and deployment. In this document we have discussed a particular solution that addresses the verification aspect of system development. In particular, this solution was in response to specific industrial requirements to enable modellers to verify their designs on demand, automatically and transparently (i.e. protecting them from the need to understand the underlying complexities of the formal methods).

Here, we have presented an approach in the form of an interoperable tool chain that builds exclusively on technologies that are mature, widespread and open source. In addition, all the new tools built for this project are (or will be) open source and widely available to the public. Importantly, our approach is complete in that it provides support for both structural models (e.g. UML class and object diagrams) as well as behaviour models (e.g. UML sequence and state machine diagrams).

The approach has been evaluated on a number of real world case studies from the embedded domain in collaboration with our industrial partners Cassidian and TXT.

Future work includes a more elaborate, bidirectional, traceability mechanism as well as a tool extension to support system simulation for validating dynamic behaviours.

Acknowledgements. This research was supported by the European Community's Seventh Framework Program (FP7/2007-2013) under grant agreement n. 248864 (MADES), and by the Programme IDEAS-ERC, Project 227977-SMScom.

MADES is a Specific Targeted Research Project (STREP) of the Seventh Framework Programme for research and technological development (FP7) – the European Union's chief instrument for funding research over the period 2007 to 2013.

References

1. Audsley, N.C., Gray, I., Indrusiak, L.S., Kolovos, D., Matragkas, N., Paige, R.: Model-based development of embedded systems - the MADES approach. In: 2nd Workshop on Model Based Engineering for Embedded Systems Design (MBED 2011), pp. 1–4 (2011)
2. Bagnato, A., Sadovykh, A., Paige, R.F., Kolovos, D.S., Baresi, L., Morzenti, A., Rossi, M.: MADES: Embedded Systems Engineering Approach in the Avionics Domain. In: 1st Workshop on Hands-on Platforms and Tools for Model-Based Engineering of Embedded Systems (HoPES 2010), p. 5 (2010)
3. Baresi, L., Morzenti, A., Motta, A., Rossi, M.: Towards the UML-Based Formal Verification of Timed Systems. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2010. LNCS, vol. 6957, pp. 267–286. Springer, Heidelberg (2011)
4. Bernardi, S., Donatelli, S., Merseguer, J.: From UML Sequence Diagrams and Statecharts to analysable Petri Net models. In: 3rd International Workshop on Software and Performance, pp. 35–45 (2002)
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.: UML/OCL Verification In Practice. In: ChaMDE Workshop (MODELS 2008), pp. 31–35 (2008)
7. Cabot, J., Clariso, R., Riera, D.: UMLtoCSP: A Tool for the Formal Verification of UML/OCL Models Using Constraint Programming. In: 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), pp. 547–548. ACM, New York (2007)
8. 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). IEEE (2008)
9. Clarke, E.M., Emerson, A.: Design and synthesis of synchronization skeletons using branching time temporal logic. In: Workshop on Logics of Programs. Springer, Heidelberg (1981)
10. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 621–645 (2006)
11. The Eclipse Foundation. Eclipse Modeling Framework (EMF) (2012), http://www.eclipse.org/modeling/emf/
12. Gogolla, M., Kuhlmann, M., Hamann, L.: Consistency, Independence and Consequences in UML and OCL Models. In: Dubois, C. (ed.) TAP 2009. LNCS, vol. 5668, pp. 90–104. Springer, Heidelberg (2009)

13. Holzmann, G.J., Joshi, R.: Model-Driven Software Verification. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 76–91. Springer, Heidelberg (2004)
14. Jones, C., O'Hearn, P., Woodcock, J.: Verified software: a grand challenge. Computer 39(4), 93–95 (2006)
15. Kolovos, D.S., Paige, R., Rose, L., Polack, F.: The Epsilon Book. Technical report, The University of York, York, UK (2010)
16. MADES. Model-based methods and tools for Avionics and surveillance embeddeD SystEmS (2012), http://www.mades-project.org/
17. MIT. alloy (2012), http://alloy.mit.edu/alloy/
18. OMG. OMG Systems Modeling Language (OMG SysML), v1.2. Technical report, OMG (2007)
19. OMG. UML Profile for MARTE : Modeling and Analysis of Real-Time Embedded Systems. Technical Report, OMG (November 2009)
20. OMG. Unified Modeling Language - Infrastructure. Technical Report, OMG (May 2010)
21. OMG. Unified Modeling Language - Superstructure. Technical Report, OMG (May 2010)
22. OMG. MOF 2 XMI Mapping Specification. Technical report, OMG (2011)
23. OMG. OMG Object Constraint Language (OCL) v2.3.1. Technical Report, OMG (January 2012)
24. Pradella, M., Morzenti, A., Pietro, P.S.: The symmetry of the past and of the future: bi-infinite time in the verification of temporal properties. In: Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, ESEC-FSE 2007, pp. 312–320. ACM, New York (2007)
25. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: 5th International Symposium on Programming, Springer, Heidelberg (1982)
26. Schmidt, D.C.: Model Driven Engineering. Computer 39(2), 25–31 (2006)
27. Selic, B.: The pragmatics of model-driven development. IEEE Software 20(5), 19–25 (2003)
28. Shaikh, A., Wiil, U.K., Memon, N.: UOST: UML/OCL Aggressive Slicing Technique for Efficient Verification of Models. In: Kraemer, F.A., Herrmann, P. (eds.) SAM 2010. LNCS, vol. 6598, pp. 173–192. Springer, Heidelberg (2011)
29. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying UML/OCL Models Using Boolean Satisfiability. In: Conference on Design, Automation and Test in Europe (DATE 2010). European Design and Automation Association, pp. 1341–1344 (2010)
30. SOFTEAM. Modelio (2012), http://modelio.org/
31. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: A state/event-based model-checking approach for the analysis of abstract system properties. Science of Computer Programming 76(2), 119–135 (2011)

Time Properties Verification Framework for UML-MARTE Safety Critical Real-Time Systems

Ning Ge and Marc Pantel

University of Toulouse, IRIT/INPT
2 rue Charles Camichel, BP 7122, 31071 Toulouse cedex 7, France
{Ning.Ge,Marc.Pantel}@enseeiht.fr

Abstract. Time properties are key requirements for the reliability of Safety Critical Real-Time Systems (RTS). UML and MARTE are standardized modelling languages widely accepted by industrial designers for the design of RTS using Model-Driven Engineering (MDE). However, formal verification at early phases of the system lifecycle for UML-MARTE models remains mainly an open issue.

In this paper[1], we present a time properties verification framework for UML-MARTE safety critical RTS. This framework relies on a property-driven transformation from UML architecture and behaviour models to executable and verifiable models expressed with Time Petri Nets (TPN). Meanwhile, it translates the time properties into a set of property patterns, corresponding to TPN observers. The observer-based model checking approach is then performed on the produced TPN. This verification framework can assess time properties like upper bound for loops and buffers, Best/Worst-Case Response Time, Best/Worst-Case Execution Time, Best/Worst-Case Traversal Time, schedulability, and synchronization-related properties (synchronization, coincidence, exclusion, precedence, sub-occurrence, causality). In addition, it can verify some behavioural properties like absence of deadlock or dead branches. This framework is illustrated with a representative case study. This paper also provides experimental results and evaluates the method's performance.

Keywords: Real-Time System, Time Property Verification, Model Transformation, UML, MARTE, Time Petri Net, Model Checking.

1 Introduction

Safety Critical Real-Time Systems (RTS) have strong timing requirements concerning system's reliability. Model-Driven Engineering (MDE) allows verifying system's properties since the early phases of system lifecycle and iteratively improving the models according to the verification results. One important issue

[1] This work was funded by the French ministries of Industry and Research and the Midi-Pyrénées regional authorities through the ITEA2 OPEES and FUI Projet P projects.

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 352–367, 2012.

is how to assess the properties for semi-formal models. UML [14] and its profile for modelling non-functional concerns, MARTE [15] are standardized modelling languages widely accepted by industrial designers for RTS. However, to our knowledge, no formal specification for the whole language is currently available. Thus, before verification, UML models must be transformed to executable and verifiable models, supported by state-of-the-art model checkers. Meanwhile, time properties must also be transformed to verifiable time assertions. A key issue in the use of model checkers is to avoid the combinatorial explosion of state space and to guarantee the verification method's performance. Combemale et al. have proposed in [5] to design *Property-driven* formal verification tools to handle many different kinds of properties for complex system models. The translation is thus dedicated to each kind of properties to improve verification performance. This work follows the same approach to design a time property verification toolset for UML-MARTE models of safety critical RTS.

This paper presents the resulting verification framework that can assess time properties like upper bounds for loops and buffers, Best/Worst-Case Response Time (B/WCRT), Best/Worst-Case Execution Time (B/WCET), Best/Worst-Case Traversal Time (B/WCTT), schedulability, and synchronization-related properties (synchronization, coincidence, exclusion, precedence, sub-occurrence, causality). In addition, it can verify some behavioural properties like absence of deadlock or dead branches. The framework relies on three steps.

Firstly, the property-driven transformation from UML-MARTE to Time Petri Nets (TPN). This method translates semi-formal UML models into executable TPN models for verification purpose. TPN [13] is selected as the verification model, as it allows expressing and verifying time properties under both logical and chronometric time models. This framework uses the TINA toolset [3] as model checker. Fig. 1 is a TPN example. Compared to Petri Nets, the transitions in TPN are extended with a time constraint that controls the firing time. For example, transition T_1 is attached with time constraint [19,27]. When the token arrives at place P_1, the local timer of T_1 starts. Between 19 and 27 time units, T_1 can be fired. This transformation covers UML architecture models (using the full Composite Structure Diagram) and behaviour models (using the full State Machine Diagram and a major subset of Activity Diagram). It is property-driven in order to limit the state space during model checking. Property-driven means that the transformation of some UML elements can be different depending on the assessed property; meanwhile the transformation does not conserve all the information in UML, but only those concerning the property verification. Another issue is raised from TPN theoretical limits. As model checking and reachability are undecidable in TPN when using stopwatch [4], the transformation method should avoid using stopwatches.

Secondly, the translation from time properties into time property patterns. Time properties are expressed using MARTE. These expressions cannot be directly verified by TPN model checker. The proposed method translates them into a set of verifiable property patterns, which are quantitative. Their values can be computed in our proposal by iterative use of the model checker relying on

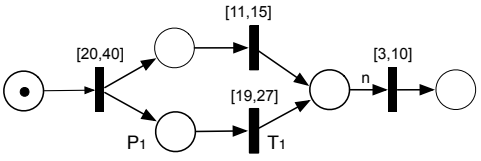


Fig. 1. Time Petri Net Example

dichotomy search. The time property patterns are independent of both the user modelling language and the verification language, making the method reusable in other verification frameworks. To make our method practical for end users, we focus on the time properties at both the event and the task levels. For the synchronization-related properties, we introduce the concept of *time tolerance*, because two simultaneous events cannot be measured without errors in the real world. The related work CCSL [2] focuses only on more symbolic time constraints at the event level without time tolerance.

Finally, the verification of time property patterns. The usual methods for verifying time properties in TPN rely mostly on LTL (Linear Time Logic), CTL (Computation Tree Logic) and μ-calculus. First, end users are not accustomed to their use. Second, these languages are not always powerful enough for some quantitative properties, in terms of both their semantic expressiveness and computation resource consummation. This issue is reduced by using the observer-based model checking. For one property pattern, an additional TPN structure extends the original TPN, then the reachability graph is generated using the highest abstraction. Complex LTL assertions become marking existence assertions, which can be cheaply computed. The observers do not change the original TPN's behaviour. This observer-based verification method relying on TPN is independent of the user modelling language, making it reusable.

This paper is structured as follows: Section 2 compares this proposal with the related works; Section 3 briefly presents the verification framework for UML-MARTE; Section 4 introduces a representative case study; Section 5 presents the property-driven transformation method from UML-MARTE models to the target TPN models; Section 6 proposes the time property translation method and the time property patterns verification method based on observers in TPN; Section 7 evaluates the proposed framework by verifying the time property in the case study and analyses the method's performance; Finally, comments on conclusion and further works are discussed in Section 8.

2 Related Works

Some works are also aimed to verify the properties in UML. The difference lies in the type of verification model and the capacity of the verification methods. Lilius and Paltor use the PROMELA language in [11] to specify UML models and exploit the SPIN model checker. This method did not involve LTL verification and the author thought SPIN is not the most efficient solution. André and Mallet use

Esterel in [2] as verification language. Although its time constraint specification language CCSL covers logical and chronometric constraint in UML, its verification approach only supports logical constraints so far to our understanding. Gagnon et al. translate UML diagrams into Maude language and verify deadlock property using LTL in [6]. This work does not handle other time properties. In terms of performance, this work denotes that they still need to test this approach on larger examples. Knapp1 and Wuttke transform UML to a special class of timed automaton in [10], then translate them to concrete programs for model checkers SPIN. It verifies the consistency between different system descriptions. This work does not concern the time aspect. Medina and Cuesta present in [12] MARTE2MAST, a tool that enables the extraction of schedulability analysis models and their direct analysis using MAST [9]. It supports analysis by using simulation tool and static analysis techniques. It defines a complete package for system analysis including the scheduling algorithms. However, as the simulation is not exhaustive, it cannot prove the correctness in all possible cases. Shousha et al. describe in [16] a search-based UML-MARTE model analysis method for starvation and deadlock detection. It uses genetic algorithms to search through the state space. As the genetic search method cannot ensure the building of the full state space including all the final states, this method cannot guarantee that the whole space will be exhaustively searched. It can detect errors but cannot prove their absence, which is a significant drawback for safety critical systems.

Petri Nets are powerful models for describing system behaviour and for verifying the properties. In [1] Andrade et al. map SysML Activity to ETPN (extended TPN with energy constraints) to estimate the energy consumption and the execution time of system. Compared with this work, the advantages of our work lie in 3 aspects: for the time property scope, we verify a large scope of time properties, while [1] covers only the execution time; for the transformation method, we propose the property-driven transformation method and consider both the architecture and the behaviour models, while [1] only considers the behaviour model; for the verification method, we propose a novel observer-based TPN verification method.

3 Overview of UML-MARTE Verification Framework

The objective of the UML-MARTE verification framework (Fig. 2) is to verify whether the design of *UML-MARTE RTS Model* satisfies the expected *Time Property*. The *System Model* consists of both *Behaviour Model* and *Architectural Model*. The former defines how the system will act and response to the outside world, while the latter describes the interconnection relation between sub-components of the system. In practice, the behaviour model is described by Activity and State Machine diagrams, and the architecture is defined by Composite Structure diagrams. All time related specifications and *Time Properties* are modelled using MARTE profile. *System Models* are translated into *TPN* models through *Behaviour/Structure Transformation*. *Time Properties* are translated into *Time Property Patterns* by *Time Property Transformation*. All

the transformations are performed automatically and the formal activities are transparent to the end user. The model checking is performed on the generated *Tag Pattern TPN* models and the corresponding *LTL/CTL/Marking Assertion* by using TINA model checker. The verification is based on the observers added in the TPN. An observer is used to observe the value of one *Property Pattern*. Finally, *Verification Result Computation* is performed to combine the *Property Pattern Results*, then the target *Time Property Verification Result* is available.

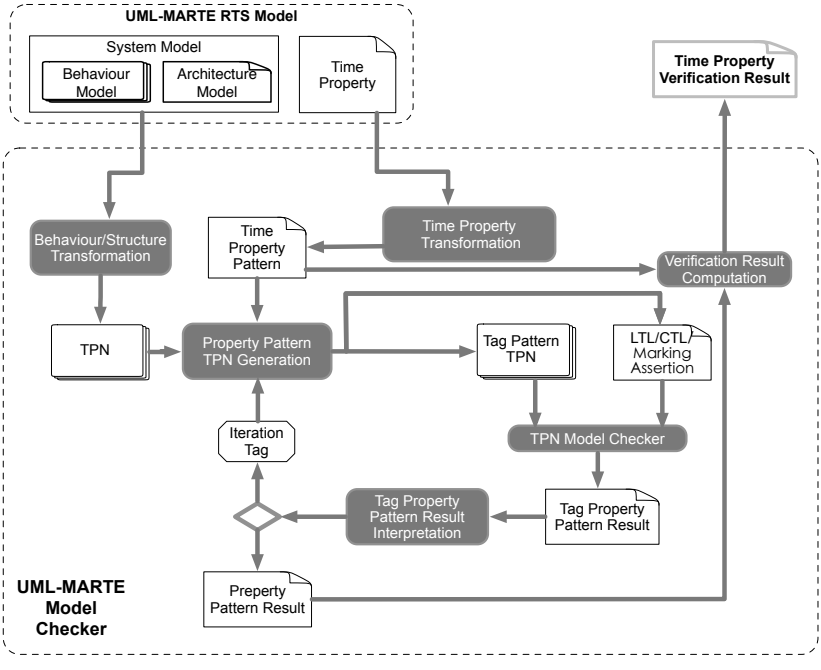


Fig. 2. UML-MARTE Model Checker

4 Case Study

We use a classical asynchronous RTS model, IMA-based airborne system, to present the application of the proposed methods. According to the general asynchronous message-driven pattern, the *sender* will regularly distribute data to the two *receivers* through the communication networks that have transfer delays and jitters. The receivers will do some computation. Fig. 3 presents the architecture model. The *sender* represents a data-collecting sensor. The *router* represents a virtual link of Avionics Full DupleX (AFDX). The two *receivers* represent two identical calculators that provide redundant control. The input data of computation is sent by the *sender* through the *router*.

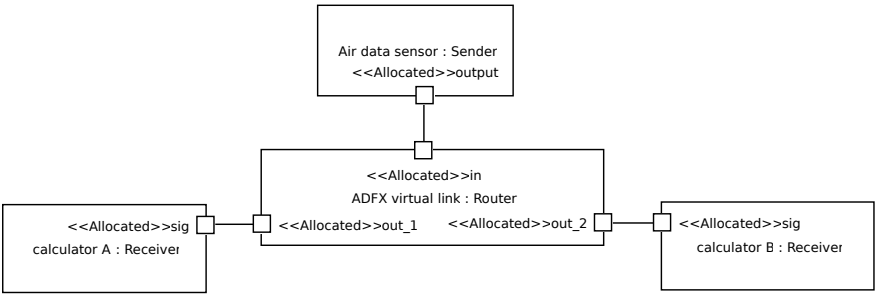


Fig. 3. IMA-based Airborne Architecture Model

The elements from MARTE in Table 1 are used to describe time specification. The redundant controller design requires that the output of the two calculators must be available at the same time in each working cycle; otherwise, the servo of the corresponding actuator cannot correctly unify the redundant command. In this case, we need to verify *the coincidence of computation tasks between calculators A and B*. As it is impossible to respect a strict simultaneous timing with an explicit local synchronisation, a time tolerance is defined. Once the two time instants fall into the same time window (size of window equals to tolerance), they are considered as coincident.

Table 1. MARTE Profile Usage

MARTE Profile	Time Specification
GRM::ResourceUsage	task's execution time
GRM::CommunicationMedia	communication delay
Alloc::Allocated	mapping the soft data pin to the hard data port

The AFDX only guarantees the communication delay upper bound, which means that the delay varies in $[t_{min}, t_{max}]$. Thus, it is obvious that the computation of calculators A and B are coincident only if the time tolerance is superior to $(t_{max} - t_{min})$, which is twice of the network jitter. In some cases, however, we need to design some supplementary protocol between the receivers to decrease the coincidence time window in order to get better system robustness.

The designer implements a *naive* protocol relying on the *hand shake* paradigm of Fig. 4, in which the two receivers are distinguished by respectively setting as active and passive modes (Fig. 5). The active one, after getting the data from the sender, sends an asynchronous notification to the passive one and automatically waits for a fixed time duration to launch its computation. The passive one will start its redundant computation once it gets the notification from its active master. As the notification message is also passed by the same AFDX network, the designer could wonder if this protocol really solves the tolerance-reduction requirement. By modifying the wait time of the active receiver and the network

jitter, the designer can use the proposed methods to verify whether the computations are still coincident under the new protocol, and then refine his design according to the verification results. In this case study, we illustrate how our approach helps verifying time properties and assists the protocol designer with guaranteed correctness and performance.

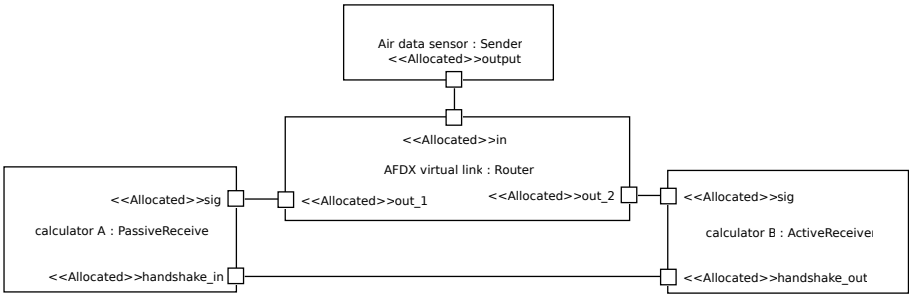


Fig. 4. IMA-based Airborne System Architecture Model with Handshake Protocol

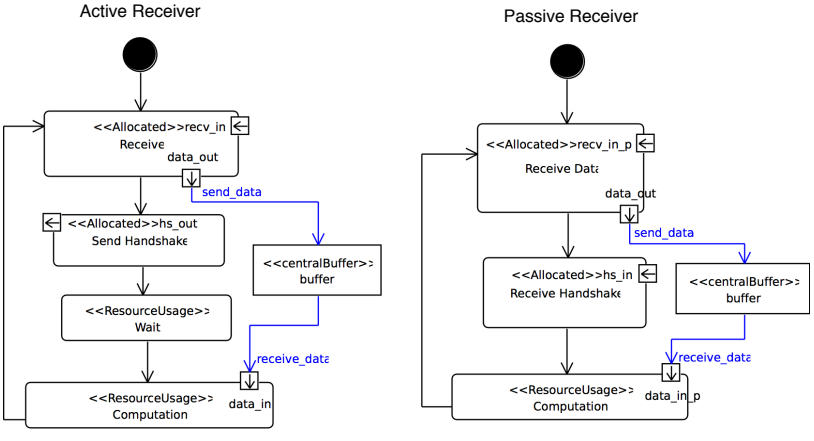


Fig. 5. IMA-based Airborne System Behaviour Model with Handshake Protocol

5 Transformation from UML-MARTE to TPN

We present the principles of the UML transformation method and illustrate the transformation for a significant subset of UML elements of both the architecture and behaviour models. Due to page limits, the complete transformation rules for UML Activity Diagram can be consulted in [8]. The description of the UML State Machine Diagram transformation will be submitted later on.

5.1 Principles

The transformation approach is property-driven, aiming to limit the state space of model checking. The approach respects the following 6 principles:

1. The framework verifies each time property in one state space generation. This means one transformation keeps only the information for one property to be verified.
2. The transformation of one UML element may be different according to the time property.
3. For some UML elements not influencing time properties, the target TPN semantics can be standardized and homogeneous for all the properties.
4. The transformation should guarantee the consistency between high-level and lower-level models. However, a correct transformation here does not imply a 100% semantic preservation, but rather to ensure the semantics necessary for the property verification are preserved through the transformation.
5. The target TPN models should ensure high performance verification, especially for large-scale asynchronous applications.
6. The patterns resulted from each element transformation should be easy to assemble. This may degrease the verification performance. But it can be compensated later by a model optimization phase that eliminates the elements irrelevant to the verification.

5.2 Architecture Model Transformation

The architecture parts in the model aim to connect the different parts to build a whole system, using communication media or shared resource. The transformation method aims to replace each component of the architecture part by its relevant behaviour part, respecting a correct instance-mapping, context-based naming and their connection relationship. In the Composite Structure Diagram (CSD) from the case study, the significant elements are *Part*, *Port* and *Connector*. The others remain important, but due to page limits, we only describe the mapping rules for *Part* and *Port* in this paper.

Part. There are two patterns for *Part*: hierarchical and primitive (Fig. 6). The behaviour is described, for the former, by the Part's inner structure, and, for the later, by the Part's associated behaviour model. For the hierarchic pattern, the architecture model is considered as a tree-like structure, and the mapping approach is applied recursively.

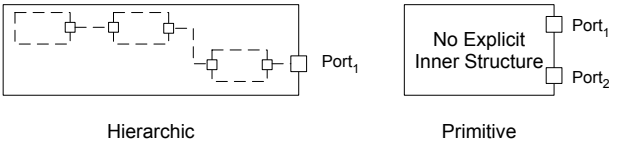


Fig. 6. CSD-Part

Port. *Port* is used to connect outside structure and inner behaviour. MARTE *Alloc::Allocated* profile is used to map the logical *PIN* in behaviour model and the physical *Port* in architectural model. *Port* is transformed to an empty TPN place to represent a data buffer concept. In a bad designed system, data quantity may overfill the buffer size, it is thus important to detect this undesired property before doing the verification, as this may cause an undecidable boundedness problem in TPN verification. In order to avoid a non-terminating verification problem, a supplementary structure is added (Fig. 7). It ensures that if the buffer is overfilled, it will raise an overflow. The overflow is represented by an ever-large marking that cannot exist in normal system. As TINA can detect on-the-fly any marking exceeding the pre-set threshold and stop state graph generation at once, this transformation method guarantees that all verification will finally terminate.

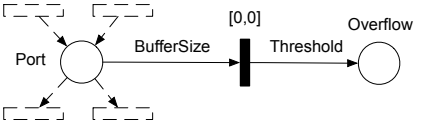


Fig. 7. CSD-Port

5.3 Behaviour Model Transformation

A general transformation pattern is defined (see Fig. 8) to automate the assembly of the TPNs generated from the behaviour model. For all non-link elements in UML, the generated TPN must contain some C_IN transitions to connect with other predecessors in static model structure. In the same manner, some C_OUT places must exist to connect with its structural successors.

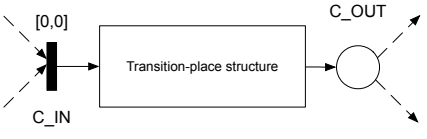


Fig. 8. General Transformation Pattern

The main elements in the UML Activity Diagram (AD) are related to control, action, resource, object, and connection. We present the transformation for *OpaqueAction* for system with single and multiple clocks (see later). An action is the fundamental unit of executable behaviour. It takes a set of inputs and converts them into a set of outputs. Depending on the abstraction level, an action could represent either a complex processing flow or a primitive one carrying out a computation. In UML-AD, there are 55 kinds of actions. Each kind covers a certain range of semantics for different usage. In order to focus on the core semantics related to time properties, we generalize the concept using the *OpaqueAction*.

Action Transformation Pattern. The transformation method is illustrated by Fig. 9. All input data-related flows should link to *B*. All output data-related flows should link to *C*. All input resource-related flows should link to *A*. All output resource-related flows should link to *D*. The execution time of one action is specified by the time constraint on transition *C*.

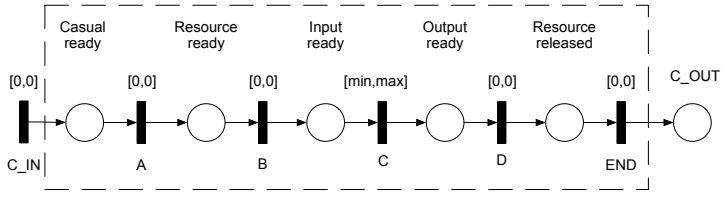


Fig. 9. UML Action Transformation to TPN

Mono-clock & Multi-clock. In the real world, each clock has an independent drift. For systems with a single clock (mono-clock) this drift can be ignored, because the difference between tick duration and real physical time is of the same proportion at any given time for any part of the system. However, in systems with several clocks (multi-clock), the model transformation should be able to represent the correct time semantic of the system by considering the different clocks' drifts. The solution is to assume a global physical clock and to project each time consumption and drift on this unique precise time reference. In our study, we use strictly the physical time notion as the exact reference for both mono and multi-clock base system.

Mono-clock Action. For mono-clock actions, the execution time is directly used after a global normalization of the time units. For example, if action A takes [3.4 ms, 4.7 ms] and actions B [78.9 us, 463.5 us], the correspondent min time and max time on the TPN transition is [34000, 47000] and [789, 4635] respectively, with the common unit of 0.1 us.

Multi-clock Action. For multi-clock actions, the execution time is specified by the expected physical time. Before integrating this time into a multi-clock based system, first we need to translate the expected physical time into tick numbers. Then its real physical time can be deduced by associating the clock's drift. We use the same example as the previous one. Let clock A and B tick theoretically every 1 us, and their backward and forward drift are both 1%, therefore action A's tick number is [3400, 4700] and B's is [78.9, 463.5]. As tick number must be integer, a rounding strategy must be taken, without introducing unreasonable conversion error. In our study, we use the floor function for t_{min} and ceiling function for t_{max}. Therefore, we have A for [3400, 4700] and B for [78, 464] as tick numbers after the rounding.

As the method assumes each component has independent clock, the drawback is that it can be too strict for those devices that share a clock. We still decide to choose this abstraction paradigm, because in the verification viewpoint, this will only lead to a false-violation. It means that if a time property is satisfied under independent-clock hypothesis, it must be also satisfied in a shared-clock system. This sufficient but not necessary condition may only cause a performance trade-off in practice, but never gives out a wrong verification result when property's proof is positive.

6 Translation and Verification of Time Property

The time properties should be translated into TPN-compatible analyzable formalism. In our case study, the property is the coincidence between two tasks. We illustrate the property translation and verification methods for this property. The translation for all the synchronization-related properties can be consulted in [7]. The verification of other properties will be presented in other papers.

6.1 Translation of Coincidence Property

Definition (Task Level Coincidence). *Task X and Y are coincident iff. the* n^{th} *occurrence of X occurs simultaneously with the* n^{th} *occurrence of Y*, $n \in \mathbb{N}$.

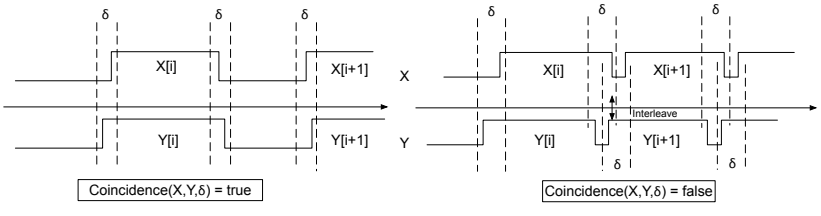


Fig. 10. Coincidence Property

As shown in Fig. 10, the coincidence between two tasks is determinated by the coincidence between the Event_{start} and the Event_{end} of tasks. For the n^{th} occurrence of task X and Y, if the two Event_{start} are coincident and the two Event_{end} are coincident within time tolerant δ, task X and Y are coincident. Formally, task coincidence is translated into the following 3 equivalent assertions.

$Coincidence(X, Y, \delta) \equiv$

$$\forall t \in \mathbb{R}_+ : (|O(X_s^t) - O(Y_s^t)| < 2) \wedge (|O(X_e^t) - O(Y_e^t)| < 2) \tag{1}$$

$$\forall t \in \mathbb{R}_+ : (|T(X_s^t) - T(Y_s^t)| < \delta) \wedge (|T(X_e^t) - T(Y_e^t)| < \delta) \tag{2}$$

$$\forall i \in \mathbb{N}^* : (T(X_e^i) + \delta < T(Y_s^{i+1})) \wedge (T(Y_e^i) + \delta < T(X_s^{i+1})) \tag{3}$$

In the above assertions, X represents task; X_a the inner event a of task X, particularly X_s for start event, X_e for end event; X_a^i the i^{th} occurrence of inner event of task X; X_a^t the occurrence of X_a which is the nearest (forward or backward) to the time instant t; $T(X_a^i)$ the occurring time instant of X_a^i; $T(X_a^t)$ the occurring time instant of X_a^t; $O(X_a^t)$ the occurrence count for X_a at time t; and δ is the time tolerance for coincidence. There are 4 time property patterns in the assertions (Table 2). The task-level property is then represented by a set of event-level property patterns.

Table 2. Time Property Patterns

Time Property Pattern	Definition
X_s^{i+k}	Representation of event X_s^{i+k}
$\lvert O(X_a^t) - O(Y_a^t)\rvert < \delta$	Occurrence difference between events X_a^t and Y_a^t
$\lvert T(X_a^t) - T(Y_a^t)\rvert < \delta$	Relative T_{max} between events X_a^t and Y_a^t
$T(X_e^i) + \delta < T(Y_s^{i+1})$	Relative T_{min} between events X_a^t and Y_b^t

6.2 Verification of Time Property Pattern $|T(a^t) - T(b^t)| < \delta$

To assess the time property, the observer pattern is added into the original TPN, and then the TINA model checker is used to verify the observer-dedicated LTL/CTL/Marking assertions for the TPN. As model checking significantly consumes time and memory resource, we use the following 2 approaches to ensure the verification performance.

- When doing the model checking, the TPN shall perform the highest possible abstraction to unfold the reachability graph. This high abstraction model should preserve the desired time property. The model-checking is on-the-fly.
- Each assertion's verification is independent in terms of reachability graph generation, so a parallel computation is possible.

We choose one of the property patterns, $|T(a^t) - T(b^t)| < \delta$, to illustrate the verification method. The principle of deciding whether two events are always occurring in a given bound is to find out whether one could advance another by time δ.

An observer pattern (Fig. 11) is added in the original TPN. The middle transition will always instantly neutralize the tokens from the places *Occ A* and *Occ B* except when one token waits for a time longer than δ that leads to the firing of *Pass* transition. To guarantee the termination of model checking, the pattern is extended by adding a large overflow number on the tester's incoming arc. We use places *tester A* and *tester B* to detect this exception. In the generated reachability graph, it only requires to verify if *tester A* or *tester B* has marking. The assertion is: $\Diamond(testerA = 1) \vee \Diamond(testerB = 1)$.

Once it is known how to verify $|T(a^t) - T(b^t)| < \delta$, it is possible to change δ to compute a near optimal tolerance. If $|T(a^t) - T(b^t)| < \delta + 1$ is verified as true, but false for $|T(a^t) - T(b^t)| < \delta$, then the near optimal tolerance is $\delta + 1$.

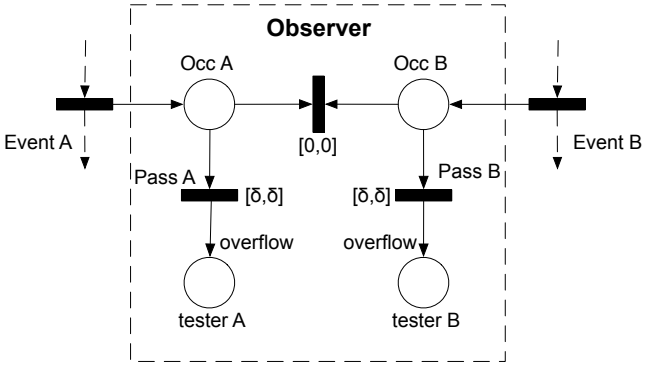


Fig. 11. $|T(a^t) - T(b^t)| < \delta$ Pattern TPN Observer

In order to improve the computation efficiency, a dichotomy search is used to reduce the complexity from $O(N)$ to $O(\log N)$.

7 Verification Result and Performance Analysis

7.1 Verification Result

In the case study, the designer aims to design a protocol relying on our verification framework, and then evaluate the system performance. The designer alters the *wait time of the active receiver* and the *jitter*, selects the *network average delay* and computes the *coincidence tolerance*. The result is shown in Fig. 12. Different coloured lines represent the result with different jitters. The variation is regular and linear because the modelled system is conceptually simple without resource sharing. It is obvious that the best wait time of the protocol is 1600ms.

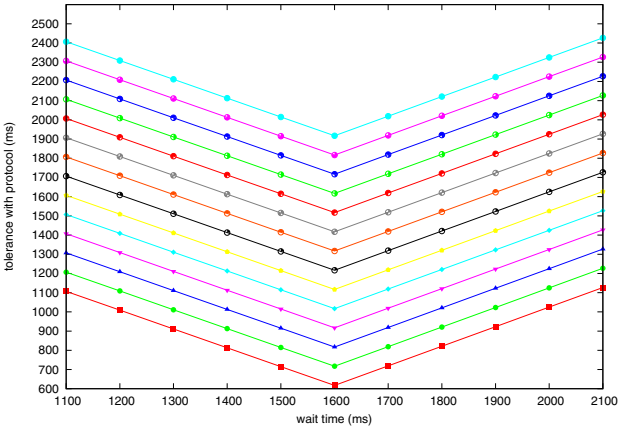


Fig. 12. Verification Result: Best Wait Time of Active Receiver

Then the user aims to evaluate this protocol by verifying the coincidence property. In the verification result (see Table 3), comparing the minimum coincidence tolerance in original system and that in the protocol system, it is obvious that the protocol succeeds in decreasing the tolerance value. We can say the system is more robust than the original.

Table 3. Verification Result: Independence with Designed Hand shake Protocol (ms)

Network Average Delay	Network Jitter	Time Window	Min Coincidence tolerance	
			Original System	Protocol System
1600	100	200	685	617
1600	300	600	1085	817
1600	500	1000	1485	1017
1600	700	1400	1885	1217
1600	900	1800	2285	1417
1600	1100	2200	2685	1617
1600	1300	2600	3085	1817
1600	1500	3000	3485	2017

7.2 Verification Performance Analysis

The performance of model checking is a very important issue for the end user. In this verification framework, we have used property-driven transformation, observer-based verification in highest abstraction mode and parallel computation methods to avoid the explosion of state space problem and to ensure a high performance. To validate the performance, we focus on two aspects: efficiency and scalability. The objective is to find out that within an acceptable time range

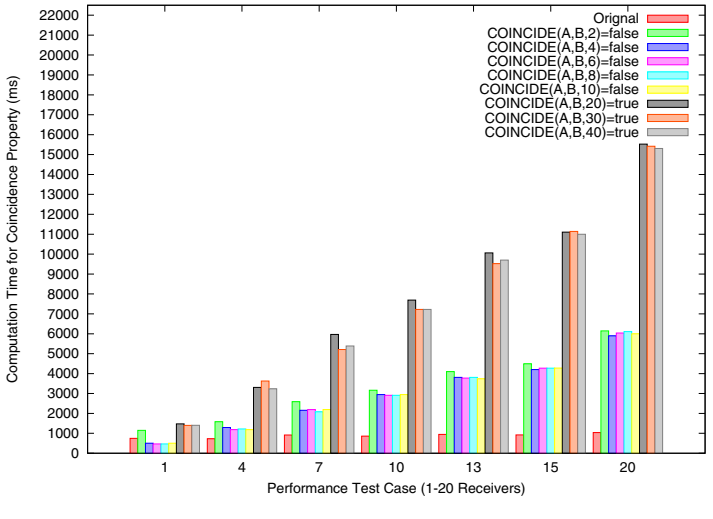


Fig. 13. Performance Evaluation

for rapid system prototyping (less than 1 minute), the framework is able to verify the coincidence property of system with a scale of *2 senders, 2 routers, and 1-20 pairs of active-passive receivers* which is representative of current avionics systems.

The performance evaluation result (Fig. 13) shows both the efficiency and the scalability of the performance. The efficiency is shown by the verification's computation time. For a system with 20 pairs of receivers, the original reachability graph generation time is less than 1000ms on a common computer; the computation time proving that the property is true is less than 16000ms, and the computation time proving that the property is false is about 6000ms. It is straightforward that proving a property false needs less time than the truth proof, because once a violation is detected the checking terminates. The scalability is shown by the linear relation between the time-over-cost of verification and the system's scale. When the increasing ratio is constant, it guarantees that if the original system reachability graph could be generated, then all the verifications of its time properties using our framework will take an appropriate time.

8 Conclusion and Further Works

In this paper, we propose a time property verification framework for UML-MARTE safety critical RTS. This verification framework can assess time properties like upper bound for loops and buffers, B/WCRT, B/WCET, B/WCTT, schedulability and synchronization-related properties (synchronization, coincidence, exclusion, precedence, sub-occurrence, causality). In addition, it can verify some behavioural properties like the absence of deadlock or dead branches. We evaluate the framework with a representative case study focusing on the property of coincidence between two tasks. The verification result demonstrates this framework can not only verify the time properties, but also assist the system's design at early phases of the lifecycle. The performance test and analysis illustrate the efficiency and scalability of the framework. Due to page limits, we will present the other properties' verification in other contributions.

One contribution is the proposition of the property-driven transformation, time property translation and observer-based verification methods. The time property translation method is independent of both the design modelling language and the verification language; the observer-based verification method is independent of the design modelling language. This independence allows these methods to be integrated in other verification frameworks. Another contribution is the approaches for reducing the state space combinatorial explosion problem, including the property-driven transformation, the highest abstraction on-the-fly model checking, and the parallel computation.

In the future, we will focus on extending this framework. On the technical side, we will optimize the TPN models by finding some reducible structural patterns without influencing the property. On the methodological side, we will experiment with other kind of properties, like the functional property, to improve the *Property-driven* approach to DSML (Domain Specific Modelling Language) model verification that started in the TOPCASED project.

References

1. Andrade, E., Maciel, P., Callou, G., Nogueira, B.: A methodology for mapping sysml activity diagram to time petri net for requirement validation of embedded real-time systems with energy constraints. In: Digital Society, ICDS 2009 (2009)
2. André, C., Mallet, F.: Specification and verification of time requirements with ccsl and esterel. In: Proceedings of the 2009 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2009, pp. 167–176. ACM, New York (2009)
3. Berthomieu, B., Ribet, P.-O., Vernadat, F.: The tool tina - construction of abstract state spaces for petri nets and time petri nets. International Journal of Production Research 42(14), 2741–2756 (2004)
4. Berthomieu, B., Lime, D., Roux, O.: Reachability problems and abstract state spaces for time petri nets with stopwatches. Discrete Event Dynamic Systems 17, 133–158 (2007)
5. Combemale, B., Crégut, X., Garoche, P.L., Thirioux, X., Vernadat, F.: A Property-Driven Approach to Formal Verification of Process Models. In: Filipe, J., Cordeiro, J., Cardoso, J. (eds.) ICEIS 2007. LNBIP, vol. 12, pp. 286–300. Springer, Heidelberg (2008)
6. Gagnon, P., Mokhati, F., Badri, M.: Applying model checking to concurrent uml models. Journal of Object Technology 7(1), 59–84 (2008)
7. Ge, N., Pantel, M.: Verification of synchronization-related properties for uml-marte rtes models with a set of time constraints dedicated formal semantic, http://hal.archives-ouvertes.fr/hal-00677925
8. Ge, N., Pantel, M., Crégut, X.: Time properties dedicated transformation from uml-marte activity to time petri net. Submitted to 5th International Workshop UML and Formal Methods (UML&FM 2012) (August 2012), http://hal.archives-ouvertes.fr/hal-00686986
9. Gonzalez Harbour, M., Gutierrez Garcia, J., Palencia Gutierrez, J., Drake Moyano, J.: Mast: Modeling and analysis suite for real time applications. In: 13th Euromicro Conference on Real-Time Systems (2001)
10. 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)
11. Lilius, J., Paltor, I.: vuml: a tool for verifying uml models. In: 14th IEEE International Conference on Automated Software Engineering, pp. 255–258 (October 1999)
12. Medina, J.L., Cuesta, Á.G.: From composable design models to schedulability analysis with uml and the uml profile for marte. SIGBED Rev. 8(1), 64–68 (2011)
13. Merlin, P., Farber, D.: Recoverability of communication protocols–implications of a theoretical study. IEEE Transactions on Communications 24(9), 1036–1043 (1976)
14. Object Management Group, Inc.: OMG Unified Modeling Language^{TM}, Superstructure (February 2009)
15. Object Management Group, Inc.: UML profile for MARTE: modeling and analysis of real-time embedded systems version 1.0 (November 2009)
16. Shousha, M., Briand, L., Labiche, Y.: A uml/marte model analysis method for uncovering scenarios leading to starvation and deadlocks in concurrent systems. IEEE Transactions on Software Engineering 1, 99 (2010)

Unification of Compiled and Interpreter-Based Pattern Matching Techniques

Gergely Varró*, Anthony Anjorin**, and Andy Schürr

Technische Universität Darmstadt,
Real-Time Systems Lab,
D-64283 Merckstraße 25, Darmstadt, Germany
{gergely.varro,anthony.anjorin,andy.schuerr}@es.tu-darmstadt.de

Abstract. In this paper, we propose a graph pattern matching framework that produces both a standalone compiled and an interpreter-based engine as a result of a uniform development process. This process uses the same pattern specification and shares all internal data structures, and nearly all internal modules. Additionally, runtime performance measurements have been carried out on both engines with exactly the same parameter settings to assess and reveal the overhead of our interpreter-based solution.

Keywords: model transformation, pattern matching interpreter, compiled pattern matcher.

1 Introduction

As model transformation undoubtedly plays an immense role in the process of model-driven development, efficiency and scalability are, therefore, important issues. In many state-of-the-art tools [1,2], model transformations are governed by imperative control flow statements, which apply declarative rules as basic transformation units. Such tools offer the usual advantages of declarativity like an easily understandable specification language, and readily available solutions provided by the underlying execution engine for many performance-critical tasks, whose optimal implementation requires years of specialized expertise. One such task is the efficient checking of the application conditions of rules, which requires identifying those parts in the system model on which the rule is executable.

This application condition checking process (as well as several other subtasks in bidirectional model synchronization and on-the-fly consistency checking scenarios) can be described as a general pattern matching problem. In this context, a pattern consists of constraints, and the matching process determines those parts of the underlying model that fulfill all these constraints. Structural constraints

* Supported by the Postdoctoral Fellowship of the Alexander von Humboldt Foundation and associated with the Center for Advanced Security Research Darmstadt.
** Supported by the 'Excellence Initiative' of the German Federal and State Governments and the Graduate School of Computational Engineering at TU Darmstadt.

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 368–383, 2012.

express restrictions that can be checked by using the services of the modelling layer (e.g., type checks, navigation along links), while non-structural constraints are handled by some other means (like integer or textual comparison). The rest of the paper will focus on handling structural constraints, which corresponds to the graph pattern matching problem [3]. Nonetheless, our approach is left open w.r.t. the integration of non-structural constraints as well.

When implementing a pattern matching engine, developers must decide on several important issues (see Sec. 2) already in the early phase of design, which are hardly modifiable in later development phases as they have radical consequences on the overall architecture. One of these critical topics is the decision whether a compiled or an interpreter-based engine is to be built.

A *compiled engine* only consists of program code that is directly executable on a certain platform without an extra module for performing pattern matching. A compiled engine typically features better runtime performance, as the algorithms are represented as machine or byte code of the underlying execution system and no operation handling layer is needed. In contrast, an *interpreter-based engine* requires a specific module (the interpreter), which is responsible for executing the operations needed to perform pattern matching. Such a technique could offer more flexibility (e.g., model-sensitive performance optimization [4]) and provide additional services such as high-level debug support, as the interpreter can access and exploit runtime information more easily.

There exists a large variety of advanced compiled pattern matcher implementations [1,5], and several sophisticated interpreter-based approaches [2,6] in different rule-based model transformation tools. Although some of them provide solutions for both cases, the resulting engines can be considered to be separate programs. The following statement [7] is, therefore, still valid: "Interpretation and code generation are often seen as two alternatives, not as a continuum". In order to allow different combinations of these alternatives, techniques are needed that handle compiled and interpreter-based pattern matchers in a uniform and tightly integrated way.

In this paper, we propose a pattern matching framework that can produce both a standalone compiled and an interpreter-based engine as a result of a uniform development process, which shares (i) the pattern specification, (ii) all internal data structures, and (iii) all internal activities except for one engine-specific module. Furthermore, applying exactly the same settings in this uniform process wherever possible, runtime performance measurements are carried out on both engines to assess and reveal the overhead of our interpreter-based solution. To our best knowledge, our proposed approach can be considered the first pattern matcher to support both a compiled and an interpreter-based setup in a unified, configurable and integrated manner and can, therefore, be easily embedded and used by different rule-based model transformation tools.

The remainder of the paper is structured as follows. Related work is discussed in Section 2. Section 3 introduces basic metamodelling terminology, pattern specification constructs, and the process of pattern matching. Sections 4 and 5 present our data structures and algorithms used in the unified pattern matching engine.

Section 6 gives a quantitative assessment and performance comparison of our compiled and interpreter-based engines. Section 7 concludes our paper.

2 Design Space of Pattern Matchers and Related Work

A widely deployable pattern matching engine should support many different application scenarios like the execution of rule-based model transformations on a desktop computer, as well as performing security monitoring tasks on an embedded system. As the computational power and the amount of available resources of these architectures significantly differ, the development of a pattern matcher requires considering several design issues that influence the applicability and performance of the approach.

The design space of pattern matching engines can be characterized by the following properties:

(1) Dependency on separate pattern matching modules. The first property, which has the closest relation to the topic of this paper, expresses whether pattern matching requires a specific interpreter (I), or can be performed without any separate modules in a standalone manner as a compiled program (C).

(2) Existence and granularity of intermediate data structures. Pattern matching interpreters and code generators that produce compiled engines usually operate on data structures with different granularity. One group of solutions directly processes the declarative, pattern specification either in a low-level form as an abstract syntax tree representation (AST), or in a high-level form as a pattern definition (P). The other group operates on a preprocessed (and typically optimized) intermediate data structure, which can either be a low-level byte code representation (BC), or a high-level search plan (SP).

(3) Generation schedule of intermediate data structures. When intermediate data structures are used by the pattern matcher, their generation schedule can also be an important design decision due to its time consuming nature. Intermediate data structures can be calculated clearly at compile time (CT), at runtime in an on-demand fashion (OD) by using a caching mechanism, or at runtime (RT) before each pattern matching process.

(4) Availability of model sensitive pattern matching strategies. The size and the structure of the underlying model often influence the runtime performance of a pattern matcher. As both characteristics can significantly change as a transformation proceeds, the runtime selection of a pattern matching strategy in a model sensitive (MS) way (i.e., by using statistics from the model) is a feasible optimization compared to approaches that rely only on metamodel-level, domain-specific (DS) information.

(5) Incrementality. As matches for a given pattern are often requested several times during the life cycle of several application scenarios, exploiting the reuse of already calculated matches is a feasible optimization possibility. In this sense, batch engines (B) restart the pattern matching process from scratch at each invocation, while incremental approaches (I) store a set of (partial) matches,

and update this set according to a defined schedule that depends on changes in the underlying model.

(6) Implementation/target language. As the applicability of a pattern matcher in a specific environment is largely determined by the implementation language of the interpreter, or the target language of the code generator, this property has also been included in our survey. The categorization here indicates support for a single (1) or multiple (*) languages.

(7) Reusability in different modelling spaces. Another important factor, in the evaluation of pattern matchers, is their reusability in different modelling environments. In this sense, an engine can operate on non-standard (NS) or standard (S) (e.g., EMF, MDR) model repositories. In the latter case, a star (*) suffix is added, if a tool provides clear interfaces to several standard modelling environments.

(8) Model access. When a tool operates in a standard compliant modelling environment, the underlying model can be accessed via tailored (T) or reflective (R) interfaces, which obviously affects both the runtime performance, and the resource (disk and memory) demand of an approach.

As a categorization of general model transformation tools is already available [8], this survey, which cannot be complete due to space restrictions, focuses on the pattern matching modules of state-of-the-art, rule-based transformation engines, and systematically compares them based on the previously listed criteria, which has been preceded by a manual inspection of the available source code (or a related publication). Table 1 presents the evaluation of these pattern matchers, which are enumerated in alphabetical order.

Table 1. Tool comparison

Tool name	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)
ATL [2]	I	BC	CT	DS	B	1	S*	R
Epsilon [9]	I	AST	N/A	DS	B	1	S*	R
Fujaba @ KS [1]	C	SP	CT	DS	B	1	S*	T
Fujaba @ PO [10]	I	SP	RT	MS	B	1	S	T – R
GReAT [11]	I C	P	N/A	DS	B	1	NS	T
GrGen [5]	C	SP	OD	MS	B	1	NS	N/A
Groove [12]	I	SP	OD	MS	B I	1	NS	N/A
Henshin [13]	I	P	N/A	DS	B	1	S	R
PROGRES [14]	I C	BC	OD	DS	B I	*	NS	N/A
VIATRA [15,16]	I	SP	OD	DS MS	B I	1	NS S	T
Our approach	I – C	SP	CT OD – RT	DS	B	1	S	T
Perfect tool	I – C	BC,SP	CT – OD – RT	DS – MS	B – I	*	NS – S*	T – R

The N/A mark shows if a categorization is non-applicable, while the '–' notation is used to express that a tool is able to cover the whole range of values in an integrated and configurable manner. The last two lines represent the evaluation of our current approach, and a hypothetic ideal pattern matching engine that could be deployed in many different application scenarios.

Table 1 clearly shows that many aspects of the ideal solution have already been solved separately by the different existing tools; however, the coverage of *design space ranges* along several properties is still not satisfactory. The main challenge here is that each of the above-mentioned design space properties represents a decision that is hard-wired into the tool architecture making reengineering tasks difficult in this context.

3 Modelling Concepts and Data Structures

In this section, we introduce basic metamodelling terminology required to present our approach, define concepts related to pattern specification, and illustrate the process of pattern matching and its runtime data structures.

3.1 Metamodels and Models

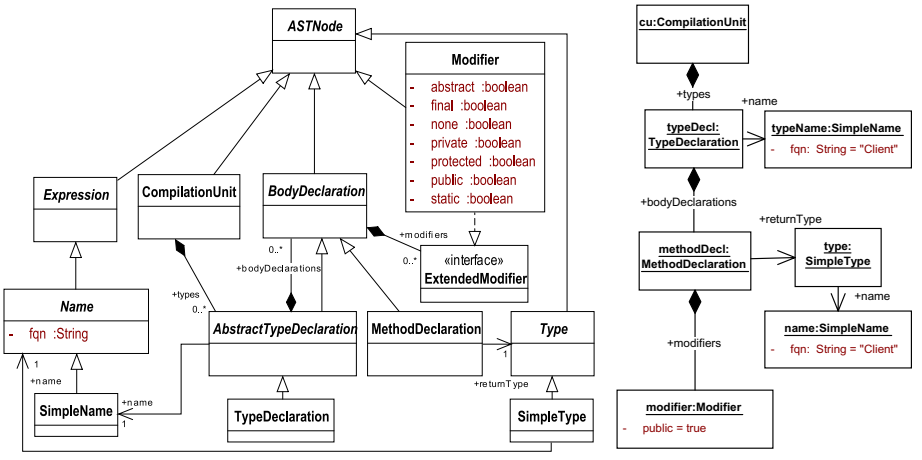
A *metamodel* is the specification of the concepts and relationships in a certain domain. Figure 1(a) depicts an excerpt of the metamodel from a case study [16] for the GraBaTs'09 transformation tool contest [17], which poses a program comprehension challenge based on the Eclipse Java Development Tools (JDT) API [18]. Using the common UML class diagram notation, parts that are relevant for our running example are depicted. The metamodel has been taken unchanged from the case study, and defines concepts as *classes* (e.g., a `CompilationUnit`). Classes can *inherit* from other classes (e.g., every `MethodDeclaration` is a `BodyDeclaration`), can contain *attributes* (every `Name` has an `fqn` as an attribute of type `String`), and can *reference* other classes (`CompilationUnit`s contain arbitrary many `AbstractTypeDeclaration`s, which each have exactly one `SimpleName`). Attributes and references are referred to as *structural features* in the rest of the paper.

A *model* is an abstraction of a system, created with an intended goal in mind. In alignment with the UML notation, nodes and edges are referred to as *objects* and *links*, respectively. A model that is expressed using concepts specified in a metamodel is said to *conform to* the metamodel. Figure 1(b) depicts a model, which corresponds to the Eclipse JDT representation of a Java class `Client` with a single public method, which returns a `Client`.

3.2 Pattern Specification

This subsection introduces the concepts needed for specifying patterns. The following definitions are based on [19].

A *pattern* is a set of constraints over a set of variables. A *variable* is a placeholder for an object in a model. *Interface variables* constitute a subset of all variables used in a pattern, and represent the variables that can be accessed outside of the pattern. All other *local* (i.e., non-interface) variables can only be



(a) An excerpt of the metamodel of the Eclipse JDT (b) A sample model

Fig. 1. An excerpt of the metamodel of the Eclipse JDT API and a sample model

accessed and used internally in the pattern. A *constraint* specifies a condition on a set of variables that must be fulfilled by the objects, which are assigned to the variables during pattern matching. A constraint consists of a *constraint type* and a set of variables (also referred to as *parameters* in this context), for which the constraint must hold. In the following, an explicit reference to the type of a constraint shall be denoted by adding a 'type' suffix.

The pattern matcher has a pluggable infrastructure for the constraints that can be used for specifying patterns.[1] In this paper, only a subset of constraints is presented. The support for extending the framework with constraints for attribute handling, positive and negative pattern invocations is already available, however, the implementation for pattern calls is still a task for the future.

A *class constraint* (`cls`) restricts the type of the objects that can be assigned to its single parameter. A *structural feature constraint* (`sf`) prescribes the existence of a link, which connects the assigned objects and conforms to a given structural feature. Both constraints `cls` and `sf` have references to types in the corresponding metamodel. A *Boolean constraint* must evaluate to `true` for the assigned values to its single parameter.

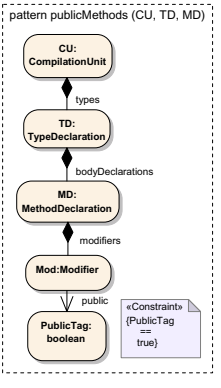
The textual specification of patterns, used in this paper, is defined by the following simplified EBNF grammar:

```
patternSpecification ::= "pattern" signature "=" body
signature ::=  NAME "(" interfaceVariables ")"
body ::= "{" constraint* "}"
constraint ::=  NAME typeReference? "(" variables ");"
typeReference ::= "<" NAME ">"
```

[1] Quantifiers can be defined at runtime in our framework, when the pattern matching is invoked, and consequently, they are not part of the pattern specifications.

```
interfaceVariables ::= variables
variables ::= ( NAME ",")* NAME
NAME ::= [a-zA-Z]+
```

Example. The graphical representation and the textual specification of pattern `publicMethods`[2] are presented in Fig. 2. This pattern requires the existence of a compilation unit `CU`, which has a type declaration `TD` with a public method declaration `MD`. Pattern `publicMethods` has three interface variables `CU`, `TD`, and `MD` (line 1). The class constraint on line 3 prescribes that variable `CU` must be mapped to a `CompilationUnit`. The structural feature constraint on line 10 requires a `types` reference that connects the objects that are assigned to variables `CU` and `TD`. The Boolean constraint on line 16 prescribes that the object assigned to variable `PublicTag` must be `true`. Please note that (i) the order of constraints (rows) in the textual representation of a pattern is arbitrary, (ii) constraints on references and attributes are handled in a uniform way (line 13), and (iii) the pattern matcher has access to all the properties of a metamodel element (information whether a class is abstract, a reference is a composition, etc.) via the references maintained in the class and structural feature constraints.




```
1  pattern publicMethods(CU,TD,MD) = {
2      // Class constraints
3      cls<CompilationUnit>(CU);
4      cls<TypeDeclaration>(TD);
5      cls<MethodDeclaration>(MD);
6      cls<Modifier>(Mod);
7      cls<boolean>(PublicTag);
8
9      // Structural feature constraints
10     sf<types>(CU,TD);
11     sf<bodyDeclarations>(TD,MD);
12     sf<modifiers>(MD,Mod);
13     sf<public>(Mod,PublicTag);
14
15     // Boolean constraints
16     true(PublicTag);
17 }
```

Fig. 2. Pattern `publicMethods` in a graphical and textual representation

3.3 Pattern Matching and Runtime Data Structures

Pattern matching is the process of determining mappings for all variables in a given pattern, such that all constraints in the pattern are fulfilled. The mappings of variables to objects are collectively called a *match*, which can be a *complete match* when all the variables are mapped, or a *partial match* in all other cases.

An *adornment* represents binding information for a sequence of variables and is indicated in the textual syntax by a sequence of letters B and F of the same length, which appears as a superscript on the name of the concept to which

[2] A more complex example scenario can be found in [4].

the adornment is attached. The letter B or F in an adornment, means that the variable in that position is bound or free, respectively.

When pattern matching is invoked, interface variables can be already bound to objects to restrict the search. The corresponding binding information of all interface variables is called a *pattern adornment*.

An *operation* represents a single atomic step in the matching process and it consists of a constraint and a constraint adornment. A *constraint adornment* prescribes which parameters must be bound when the operation is executed. A *check operation* has only bound parameters. An *extension operation* has free parameters, which get bound when the operation is executed.

Example. Suppose a matching process for the pattern `publicMethods` (Fig. 2) is to be run on the model of Fig. 1(b), with the interface variable `CU` bound to the compilation unit `cu` at pattern invocation. This single mapping itself constitutes a partial match, and the corresponding pattern adornment is BFF,[3] since only the first interface variable has been bound. When the pattern matching process terminates, a complete match is returned, which maps variables `CU`, `TD`, `MD`, `Mod`, and `PublicTag` to objects `cu`, `typeDecl`, `methodDecl`, `modifier`, and a Boolean `true` value, respectively.

4 Workflow of Compiled Pattern Matching

This section presents the workflow for generating a compiled pattern matching engine. In this process, a pattern matcher class is generated for every pattern. Although an adornment is part of runtime binding information, the generated engine must be prepared to handle a fixed set of pattern adornments, which are selected in advance at compile time. For each selected pattern adornment, a method that performs the actual pattern matching is generated into the corresponding pattern matcher class according to the approach depicted in Fig. 3, which operates as follows:

Section 4.1 For each constraint type used in the pattern specification, the set of allowed constraint adornments is calculated.
Section 4.2 For each pair of constraint and allowed constraint adornment, an operation is loaded.
Section 4.3 Operations are filtered and ordered by a search plan algorithm to produce an efficient search plan.
Section 4.4 Based on the search plan, generators are created, which control the code generation process.

When the pattern matcher is invoked at runtime, the pattern adornment is determined and used to select and execute the corresponding generated method.[4] The selected method performs the complete pattern matching process, collecting

[3] The pattern adornment only contains binding information for interface variables (`CU`, `TD` and `MD` in this example).
[4] If no such method exists, an exception is thrown, which might initiate a pattern matcher regeneration process.

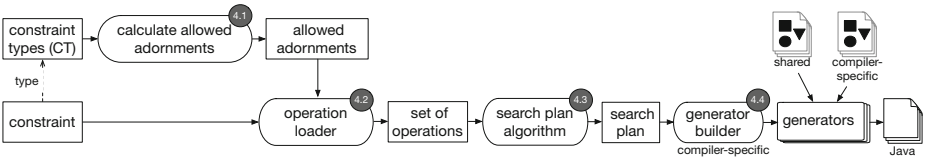


Fig. 3. Process for producing a compiled pattern matching engine

complete matches in a result set. The following subsections discuss the steps of the process in detail.

4.1 Allowed Adornment Calculation

In general, not every possible adornment is valid for every constraint type. Our framework allows the underlying modelling layer to define the set of allowed adornments for every available constraint type in a configurable way. For presentation purposes, standard Ecore/EMF is assumed as the modelling layer in the following: A structural feature constraint type, referring to a bidirectional reference, would allow adornments `BB`, `BF`, and `FB`, where `BB` means checking the existence of a link, while `BF` and `FB` denote forward and backward navigations, respectively. In the case of unidirectional references, only `BB` and `BF` adornments make sense. Analogously, only `BB` and `BF` adornments are allowed for structural feature constraint types referring to an attribute. Only the adornment `B` is allowed for class and Boolean constraint types.[5]

Example. The framed part of Figure 4(a) shows the complete list of constraint types and allowed adornments for pattern `publicMethods`. Lines 1–5 show that class constraint types have the adornment `B` as type checks can be performed only on a bound parameter. Line 6 represents a check for the existence of a `types` link, while line 7 means a forward navigation along a link of type `types`.

4.2 Operation Loading

The operation loader prepares all the operations that might be needed to execute pattern matching by iterating through all constraints of a pattern. It looks up the allowed adornments for the type of the constraint, and creates a new operation for each combination of constraint and allowed adornment.

Example. Figure 4(a) lists the set of operations that might be needed to calculate matches for pattern `publicMethods`. For example, line 1 shows that variable `CU` must already be mapped to an object before a corresponding type check can be performed. Line 7 expresses that a forward navigation along `types` links is executable only when an object has already been bound to variable `CU`.

[5] The set of allowed adornments for standard MOF/JMI, in contrast to Ecore/EMF, would also allow `FF` for associations, and `F` for class constraint types. Similarly, a modelling layer with additional EMF services could also support an extended set of allowed adornments for EMF supporting e.g., `FB` for indexed attributes.

```
1  cls<CompilationUnit>^B(CU)
2  cls<TypeDeclaration>^B(TD)
3  cls<MethodDeclaration>^B(MD)
4  cls<Modifier>^B(Mod)
5  cls<boolean>^B(PublicTag)
6  sf<types>^BB(CU,TD)
7  sf<types>^BF(CU,TD)
8  sf<bodyDeclarations>^BB(TD,MD)
9  sf<bodyDeclarations>^BF(TD,MD)
10 sf<modifiers>^BB(MD,Mod)
11 sf<modifiers>^BF(MD,Mod)
12 sf<public>^BB(Mod,PublicTag)
13 sf<public>^BF(Mod,PublicTag)
14 true^B(PublicTag)
```

(a) Operations

```
1  publicMethods^BFF(CU,TD,MD) = [
2    cls<CompilationUnit>^B(CU);
3    sf<types>^BF(CU,TD);
4    cls<TypeDeclaration>^B(TD);
5    sf<bodyDeclarations>^BF(TD,MD);
6    cls<MethodDeclaration>^B(MD);
7    sf<modifiers>^BF(MD,Mod);
8    cls<Modifier>^B(Mod);
9    sf<public>^BF(Mod,PublicTag);
10   cls<boolean>^B(PublicTag);
11   true^B(PublicTag);
12 ]
```

(b) Search plan for the sample pattern `publicMethods`

Fig. 4. Data structures used by both engines

4.3 Search Plan Generation

Operations are filtered and ordered by a search plan algorithm to produce efficient search plans. Due to space restrictions, search plans generation techniques like [1,4,6] and their details (e.g., cost functions, optimization algorithms) are not discussed in this paper.

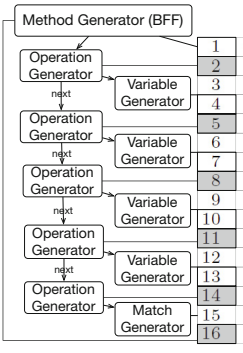
A *search plan* is defined as a sequence of operations satisfying the following conditions:

1. Each constraint in the pattern has exactly one corresponding operation in the search plan.
2. Each variable that must be *bound* according to the constraint adornment of an operation is either already bound in the pattern adornment, or appears as a free variable in one of the preceding operations.
3. Each variable that must be *free* according to the constraint adornment of an operation is not bound in the pattern adornment and does not appear in any of the preceding operations as free variables.
4. Each extension operation must be immediately followed by class check operations that perform the type checking of the free variables of the extension operation.

Example. Figure 4(b) shows a search plan for pattern `publicMethods`, when the first interface variable (`CU`) is bound when pattern matching is invoked. The search plan has been derived from the set of operations (Fig. 4(a)) by filtering and reordering, and it fulfills all conditions 1–4. The constraint adornment on line 3, for example, is valid, as `CU`, which must be bound, is indeed bound in the pattern adornment. Similarly, `TD`, which must be free, is free in the pattern adornment, and does not appear in any preceding operation as a free variable.

4.4 Code Generation for a Compiled Pattern Matcher

The final step is code generation, which is controlled by a set of different *generators* that each maintain a link to a template.[6] As depicted in detail in Fig. 5 for the pattern adornment `BFF`, a *method generator* references a series of *operation generators*, responsible for navigation in the model (e.g., lines 2 and 5), which in turn reference *variable generators*, that store the determined values for variables on the JVM stack (e.g., lines 3–4 and 6–7). A *match generator* is responsible for the code that should be executed when a (complete) match is found (line 15). Code generation is initiated by starting the template of the method generator.




```
1  private void matchBFF(CompilationUnit cu) {
2    for (AbstractTypeDeclaration tempTD : cu.getTypes()) {
3      if (tempTD instanceof TypeDeclaration) {
4        TypeDeclaration td = (TypeDeclaration) tempTD;
5        for (BodyDeclaration tempMD : td.getBodyDeclarations()) {
6          if (tempMD instanceof MethodDeclaration) {
7            MethodDeclaration md = (MethodDeclaration) tempMD;
8            for (ExtendedModifier tempMod : md.getModifiers()) {
9              if (tempMod instanceof Modifier) {
10               Modifier mod = (Modifier) tempMod;
11               Boolean tempPublicTag = mod.getPublic();
12               if (tempPublicTag != null) {
13                 Boolean publicTag = tempPublicTag;
14                 if (publicTag.booleanValue()) {
15                   // Complete match
16 } } } } } } } } }
```

Fig. 5. Generated code to handle pattern matching

5 Workflow of Interpreter-Based Pattern Matching

Interpreter-based engines typically carry out all their pattern matching related tasks at runtime. As such, the pattern specification should be considered as a starting point when the matching process is invoked. Furthermore, in order to avoid any dependencies on generated data structures, interpreter-based pattern matchers typically use a reflective API of the modelling layer to access objects and to navigate along links.

In contrast to this general tendency, our interpreter-based solution (whose workflow is presented in Fig. 6) uses tailored interfaces (just like the compiled pattern matcher) to completely eliminate the performance effects that would be caused by the different model access strategies, and thus, to enable a fair quantitative comparison of our compiled and interpreter-based engine variants. This requires generated operation classes (and a loader class), which are subclasses of their generic counterparts and are produced at compile time (as shown by the solid arrows). A generated operation, thus, represents an atomic step in the pattern matching process and uses a tailored interface for model access purposes.

[6] Velocity is used as a template language and engine.

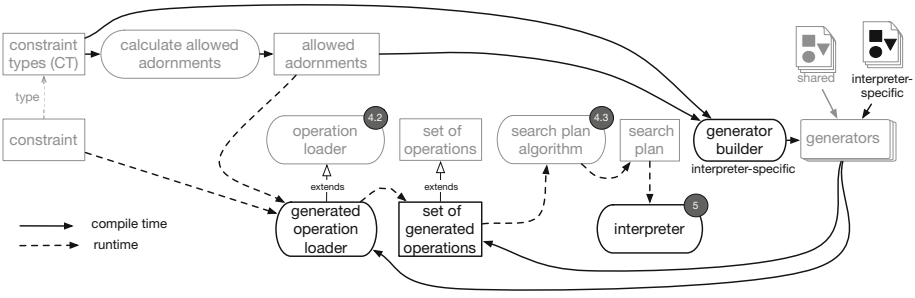


Fig. 6. Process for producing an interpreter-based engine

Although generated operations are produced at compile time, *all the remaining activities* are executed at *runtime* as highlighted by dashed arrows in Fig. 6. The runtime part of the interpreter-based pattern matching approach, which reuses all parts from Sec. 4 that are shaded in grey, works as follows:

Section 4.2. When the pattern matcher is invoked, a (generated) operation is loaded for each pair of constraint and allowed constraint adornment. As constraints are part of the pattern, this behaviour completely aligns with the expectation that an interpreter-based engine should start processing the pattern specification at runtime.

Section 4.3. Operations are filtered and ordered by a search plan algorithm to produce a search plan. The algorithm is obviously influenced by the pattern adornment, that has been determined by examining which interface variables have been mapped to objects in the model at pattern invocation.[7]

Section 5. Finally, the interpreter performs pattern matching by executing the search plan. The details of this interpreter module are presented in the remaining part of this section.

The interpreter uses a *match array* for storing the current match and the operations in the search plan. Every operation has a link to the next operation and a mapping, that identifies the slots in the match array, which correspond to the parameters of the operation.

When the interpreter is invoked, it prepares the initial match array, whose size is determined by the number of variables in the pattern. The initial match array is filled according to the input mapping of interface variables to objects in the model, and is passed to the first operation in the search plan. When an extension operation is performed, the structural feature is traversed forwards (`BF`) or backwards (`FB`) to bind the corresponding free variable, the type of the accessed object is validated, and the execution is passed on to the following operation for subsequent processing together with the extended match array. For a check operation, the operation is performed and, in case of success, the current match array is simply left unchanged and passed on. When the match

[7] Search plans can be cached and reused depending on the configuration settings.

array passes beyond the last operation, it represents a complete match and is stored in a result set. A single match array is used for storing all (partial) matches, and only complete matches are copied and stored in a result set.

This behaviour is a depth-first traversal[8] of a state space (just like the compiled approach), where a *state* represents the processing of a partial match by an operation. The state space can alternatively be described as a tree, whose root is the initial match, while internal nodes and leaves correspond to partial and complete matches, respectively.

6 Measurement Results

In this section, we present measurement results from comparing our compiled engine with our interpreter-based pattern matching engine, both produced via a uniform process, using our framework as discussed in Sec. 4 and Sec. 5. The pattern used for the measurements is from [16] and describes all classes, excluding inner classes, that contain at least one public static method, whose return type is the same as the class itself. Five models (Set0 – 4) of different size were taken unchanged from the same case study.

A 1.57 GHz Intel Core2 Duo CPU with 2.96 GB RAM was used for all measurements. Windows XP Professional SP 3 and Java 1.6 served as the underlying operating system and virtual machine, respectively. The maximum heap size was set to 1536 MB. *User time*, which is the amount of time the CPU spends performing actions for a program, was measured. On the used machine, this could only be measured with a precision of ±12.625 ms. As a single pattern matching task takes less time than this value, each measurement was performed as a series of blocks. In a measurement block, the pattern matching task of the compiled engine was performed 100 times, while in the case of the interpreter-based engine, search plan generation and pattern matching were executed 500 times, and 20 times, respectively.[9] This block-based measurement was repeated 50 times in all cases to provide stable average values.

The generated search plans and resulting matches in both cases were validated manually to be equivalent. To obtain a fair comparison, search plan generation and pattern matching were measured separately for the interpreter.

Table 2 presents the measured execution times. The first column indicates the model used (Set0 – 4), the second and third columns the size of the model in number of Java classes and objects, respectively. The fourth column denotes the corresponding state space size in number of states, the fifth column the time (ms) for the compiled engine, while the last two columns show the time (ms) for search plan generation and pattern matching for the interpreter-based engine.

[8] Alternative strategies (e.g., breadth-first traversal) would typically duplicate match arrays during the execution of extension operations, which would cause an increased memory consumption.

[9] The only reason for selecting different block length values was to have raw data approximately on the same scale, which already belongs to the measureable range.

Table 2. Measurement results

	Java classes	Model size	State space	Compiled PM	Interpreter-based SP	Interpreter-based PM
	#	#	#	ms	ms	ms
Set0	14	70447	232	0.03	0.12	0.19
Set1	40	198466	549	0.08	0.12	0.53
Set2	1605	2082841	37348	12.99	0.12	41.91
Set3	5769	4852855	94300	31.17	0.41	142.34
Set4	5984	4961779	103122	36.01	0.84	230.38

Table 2 shows that the compiled engine, which represents algorithms in a byte code form, and thus, requires no operation handling tasks to be executed, outperformed the interpreter-based engine in all cases, and was about 4–6 times faster. For the interpreter-based engine, the time spent for search plan generation increased for the larger models (Set3 and Set4), which are loaded into memory prior to search plan generation. We believe this is caused by garbage collection, which becomes necessary due to the substantial difference in size of the models as compared to Set0 or Set1.

In order to clarify the exact reason for the large execution times of the interpreter-based engine in case of large models, and to justify our assumption on the role of garbage collection, further measurements should be performed in the future with larger heap size settings, on more patterns and on different application domains e.g., like the ones mentioned in [20]. An additional comparison using reflective interfaces for our engines would also be interesting. Furthermore, other dimensions could be measured including memory footprint, number of loaded classes and RAM consumption.

Note that the main goal of our measurements was neither to quantitatively analyze pattern matchers with search plans [4,20], nor to draw any conclusion regarding the performance of compiled and interpreter-based engines in general, nor to repeatedly justify the obvious comparative statements about the runtime superiority of compiled techniques, but to assess the *exact extent* of performance differences between our pattern matcher variants. In our measurements, both variants accessed the EMF-based modeling layer via the same tailored interfaces. Additionally, they applied the same algorithm to produce the same search plan, which resulted in the same state space traversed in the same depth-first order. We think that this unified measurement setup could only be achieved by using a framework-based solution, and any modification in this setup would introduce performance influencing factors that are independent from the main issue under investigation (i.e., the exact effects caused by the selection of our compiled or interpreter-based engine).

7 Conclusion

In this paper, we proposed a pattern matching framework that can produce both a standalone compiled and an interpreter-based engine in a uniform process that

shares all internal data structures and the majority of modules. As main advantages, our framework-based solution (1) eases the task of reengineering a tool with respect to its pattern matcher module, and (2) enables a switching possibility between the compiled and interpreter-based engines at runtime. Additionally, we carried out performance measurements on both engines with the same parameter settings to assess the overhead of our interpreter-based solution.

The proposed approach has been implemented in the context of the Democles project, whose goal is to provide a model-based pattern matcher implementation, which integrates several advanced pattern matching algorithms in one framework, and can be embedded into different tools. Contributions of this paper cover one aspect of this project, which was to present the unified process for handling compiled and interpreter-based pattern matchers. The model-sensitive search plan algorithm of the pattern matcher has been published in [4]. The interpreter additionally supports (a yet unpublished) step by step execution possibility, which can be the basis of a high-level debugger in the future.

References

1. Geiger, L., Schneider, C., Reckord, C.: Template- and modelbased code generation for MDA-tools. In: Giese, H., Zündorf, A. (eds.) Proc. of the 3rd International Fujaba Days, Paderborn, Germany, pp. 57–62 (2005), ftp://ftp.upb.de/doc/techreports/Informatik/tr-ri-05-259.pdf
2. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
3. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation, vol. 1: Foundations. World Scientific (1997)
4. Varró, G., Deckwerth, F., Wieber, M., Schürr, A.: An Algorithm for Generating Model-Sensitive Search Plans for EMF Models. In: Hu, Z., de Lara, J. (eds.) ICMT 2012. LNCS, vol. 7307, pp. 224–239. Springer, Heidelberg (2012)
5. Geiß, R., Batz, G.V., Grund, D., Hack, S., Szalkowski, A.: GrGen: A Fast SPO-Based Graph Rewriting Tool. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 383–397. Springer, Heidelberg (2006)
6. Varró, G., Varró, D., Friedl, K.: Adaptive graph pattern matching for model transformations using model-sensitive search plans. In: Karsai, G., Taentzer, G. (eds.) Proc. of International Workshop on Graph and Model Transformation. ENTCS, vol. 152, pp. 191–205. Elsevier (2005)
7. Voelter, M.: Best practices for DSLs and model-driven development. Journal of Object Technology 8(6), 79–102 (2009)
8. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal 45(3), 621–645 (2006)
9. Kolovos, D., Rose, L., Paige, R.: The Epsilon book, http://www.eclipse.org/gmt/epsilon/doc/book/
10. Giese, H., Hildebrandt, S., Seibel, A.: Improved flexibility and scalability by interpreting story diagrams. In: Margaria, T., Padberg, J., Taentzer, G. (eds.) Proc. of the 8th International Workshop on Graph Transformation and Visual Modeling Techniques, ECEASST, vol. 18 (2009)

11. Agrawal, A., Karsai, G., Neema, S., Shi, F., Vizhanyo, A.: The design of a language for model transformations. Software and Systems Modeling 5(3), 261–288 (2006)
12. Rensink, A.: The GROOVE Simulator: A Tool for State Space Generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer, Heidelberg (2004)
13. Arendt, T., Biermann, E., Jurack, S., Krause, C., Taentzer, G.: Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 121–135. Springer, Heidelberg (2010)
14. Schürr, A., Winter, A.J., Zündorf, A.: The PROGRES Approach: Language and Environment. In: Handbook on Graph Grammars and Computing by Graph Transformation, vol. 2: Applications, Languages and Tools, pp. 487–550. World Scientific (1999)
15. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental pattern matching in the VIATRA model transformation system. In: Karsai, G., Taentzer, G. (eds.) Proc. of the 3rd International Workshop on Graph and Model Transformation, pp. 25–32. ACM (2008)
16. Sottet, J.S., Jouault, F.: Program comprehension, http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/cases/grabats2009reverseengineering.pdf
17. Levendovszky, T., Rensink, A., van Gorp, P.: 5th International Workshop on Graph-Based Tools: The Contest, http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/
18. Beaton, W., des Rivieres, J.: Eclipse platform technical overview. Technical report, The Eclipse Foundation (2006)
19. Horváth, Á., Varró, G., Varró, D.: Generic search plans for matching advanced graph patterns. In: Ehrig, K., Giese, H. (eds.) Proc. of the 6th Int. Workshop on Graph Transformation and Visual Modeling Techniques, vol. 6 of ECEASST (2007)
20. Varró, G., Schürr, A., Varró, D.: Benchmarking for graph transformation. In: Proc. of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing, Dallas, Texas, USA, pp. 79–88. IEEE Computer Society Press (2005)

OCL-Based Runtime Monitoring of Applications with Protocol State Machines

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

Abstract. This paper presents an approach that enables users to monitor and verify the behavior of an application running on a virtual machine (like the Java virtual machine) at an abstract model level. Models for object-oriented implementations are often used as a foundation for formal verification approaches. Our work allows the developer to verify whether a model corresponds to a concrete implementation by validating assumptions about model structure and behavior. In previous work, we focused on (a) the validation of static model properties by monitoring invariants and (b) basic dynamic properties by specifying pre- and postconditions of an operation. In this paper, we extend our work in order to verify and validate advanced dynamic properties, i. e., properties of sequences of operation calls. This is achieved by integrating support for monitoring UML protocol state machines into our basic validation engine.

1 Introduction

When one faithfully follows the Model-Driven Development (MDD) paradigm, abstract representations of all artifacts, in particular of code, are needed in form of models. Model-like descriptions can be used as central parts in the software development process and are considered to be a promising paradigm for effective software production. Models can be employed in all development phases and for different purposes. Consequently and despite all justified criticism, the Unified Modeling Language (UML) is playing a pivotal role as a modeling language. Nearly every software engineer understands at least the UML core concepts, while other more specialized modeling languages first need to be explained from the scratch. This central role of the UML can also be observed by looking for transformation approaches from UML to more formal and specialized languages or tools such as the Alloy [24] language, SAT [25] or model checkers [19].

When using UML models for abstractions of concrete software systems, model quality is important. It has to be ensured that the developed models correspond to the implementation to be abstracted from. Otherwise formal quality assurance techniques would verify some disconnected abstract model and not the concrete implementation. This is especially true, if the implementation is not fully generated from the model and finalized by a developer. This is currently the most common case.

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 384–399, 2012.

In [17] simulation of the model is proposed in the overall process of model checking. The process is shown in Fig. 1 which is adapted from [17, p. 8]. Our contribution and extension to the process is shown in the parts having a grey background.

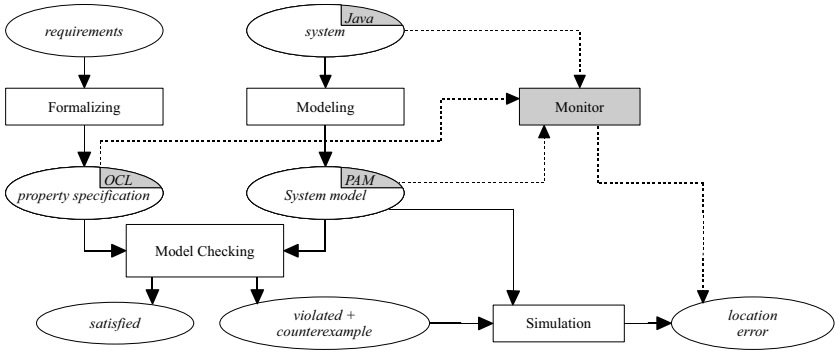


Fig. 1. Monitoring in the context of Model Checking (c.f. [17])

In our approach, we do not only simulate the model. We combine the system model with the implementation (the system) in order to be able to detect mismatches between the implementation, the system model and the property specification. We do so while executing the actual implementation. As systems we consider applications running inside a virtual machine, such as Java application running inside the Java Virtual Machine (JVM). Our system model will be defined as a UML class model extended by UML protocol state machines [20] and augmented with property specifications resp. assumptions formulated as OCL (Object Constraint Language) [21] state invariants and OCL operation pre- and postconditions. Since the elements of this system model need to be identified by our monitor, the system model needs to be aligned to the implementation. We call such a model a platform aligned model (PAM). We connect these components with a monitor in order to verify assumptions about the components at runtime. Our monitor can be started at any time that the concrete system, i. e., the Java application, is running. As an extension to our work presented in [15] and [16], we show how a state machine extension of the employed validation engine can be used without modifying our monitor component. Here, we show how UML protocol state machines (psms, singular psm) can be used to validate the correct sequence of operation calls, i. e., a protocol definition for a given class. We will further discuss some threads to validity which have to be considered when using a monitor approach like ours.

The rest of this paper is structured as follows. In Section 2 we put forward the basic ideas of our proposal for analyzing applications running in the Java virtual machine. Section 3 gives an overview on the integration of protocol state machines into our validation engine USE [11]. Section 4 explains the employment of protocol state machines in combination with our monitoring approach by

means of a middle-sized case study applied in our tool USE. Section 5 discusses related work. The paper ends with a conclusion and ideas for future work.

2 Monitoring

In this section we explain our monitoring approach. A more detailed description can be found in [15]. The main idea of our approach is to monitor a running implementation of a system and to extract a more abstract representation of the current system state into a validation engine. We call this abstract representation a *snapshot* of the system under monitoring (SUM), because in general it is a small subset of the artifacts of the running system. Since we want to focus only on central parts of the implementation we leave out unimportant parts. The basis for this snapshot is a model which is more abstract than the implementation, e. g., by defining associations which are not present in programming languages, but specific enough to be able to find relevant parts inside the SUM, e. g., by specifying concrete package names. Because of this alignment between the most specific platform model, e. g., byte code and platform independent models we call this model level *platform aligned model (PAM)*.

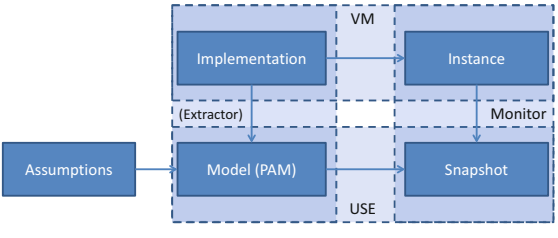


Fig. 2. Overview of the monitoring approach

As shown in Fig. 2 the PAM is enriched with assumptions about the running system. These assumptions are verified during the monitoring process by our validation engine USE. In order to be able to verify assumptions specified in a model USE needs an instance (i. e., objects and links) of it. In the monitoring context we call this instance a snapshot. Figure 2 shows this relation at the bottom. The monitor ensures, that the instance required by USE is a valid snapshot of the monitored instance inside the virtual machine. The virtual machine itself as shown at the top of the figure uses the implementation and an instance, i. e., the (heap) memory, stack, stack pointer, etc. of a running program. The PAM can be defined in several ways. For example, it can be step-wise refined when developing a system or it can be extracted by using reengineering techniques as shown in [16]. Furthermore, it can be generated when using model driven development.

Using modern virtual machine implementations like the JVM or the CLR of Microsoft .NET allows our monitor to use a rich pool of debugging and profiling

interfaces. For example, the Java Platform Debugger Architecture[22] enables third party tools to easily access applications running inside a local or remote virtual machine. An important part of this interface is the possibility to retrieve information about instances of a specific type. This is used as an entry point for our monitoring approach described next.

First, the validation engine needs to be configured with the corresponding PAM and the SUM needs to be started. Next, the monitor needs to be connected to the running system. If the startup of a SUM is important, the user can also start the application with specific parameters, so that it suspends directly when started and is resumed only if the monitor signals this to the application. When the monitor is connected after the application is already running, the monitor creates a snapshot of the current system state. The following descriptions of the steps to create this abstract snapshot are explained in more detail in [15].

1. For all classes in the PAM which can be matched to an already loaded class in the VM, all existing instances of them are mapped to newly created instances of the platform aligned model.
2. For each created instance in the previous step the values of the attributes defined in the PAM are read. This step includes a mapping for values of primitive types to built-in OCL types, e. g., String and Real (c. f. [28]). Attribute values with a type of a class defined in the PAM need to be mapped using the mapping created in the first step.
3. For all associations in the PAM, links are created between corresponding instances.
4. By using the current stack-trace of the monitored system the current operation call sequence relevant to the monitored elements can be rebuilt. For this, the deepest operation call to a monitored operation (an operation specified in the PAM) on the call stack acts as an entry point for the following monitored operations on the call stack.

After such a snapshot has been constructed, the monitor needs to register to several events that occur in the VM in order to keep the snapshot synchronized with the running system and to allow a dynamic monitoring of the SUM. Currently our monitor makes use of the following breakpoint and watchpoint locations:

1. At class initialization to allow the registration of all other breakpoints. This ensures, that classes which were not loaded while taking the snapshot are also monitored.
2. At constructors of monitored classes. This allows the monitor to keep track of newly created instances and therefore enables an incremental construction of the system state in contrast to always construct a new snapshot of the running system when needed.
3. At the start of a monitored operation. This enables the monitor to validate preconditions at runtime and to follow the call sequence.
4. Just before the exit of an operation call. This enables the monitor to validate postconditions. The break must occur after the result of the operation is calculated.

5. When a monitored attribute is modified. A monitored attribute might be an attribute or association end inside of the PAM.

Monitoring an application in the presented way in combination with our validation engine USE allows a user to monitor the validity of UML constraints like multiplicities or composition properties, invaraints, pre- and postconditions *without* the need to modify the source code of the application or to use special bytecode injection mechanism. In addition, without changing the monitor component, improvements made to the validation engine can be used. For example after adding support for protocol state machines to USE, as described next, only the PAMs of the monitored systems needed to be extended to allow a more detailed monitoring of call sequences. Without the use of protocol state machines, only a very small part of a call sequence could be validated in one step, because OCL only allows access to the state just before an operation was called. Using protocol state machines it is possible to validate operation call sequences of arbitrary length.

3 Protocol State Machines in USE

The UML specifies two kinds of state machines: behavioral and protocol state machines [20]. As the name suggest, the former kind is used to specify the behavior of UML elements including actions attached to transitions to specify changes inside a system while taking a transition. The latter one specifies the allowed call sequences of a protocol. In USE we added support for protocol state machines in the context of a class. Following the general idea of USE, we start with a small well-defined subset of the many features for UML state machines. In the following we describe this implemented subset and its semantics.

First of all, all state machines in USE are flat, i. e., they have only one region and no composite states. They have only a single initial and a single end state. All other states are proper states and no pseudo states, which means that there are no forks or joins. States can have a state invariant which needs to be valid if a given psm instance is in the corresponding state. The context of a psm instance and also for the state invariant (accessed by using `self` in an OCL expression) is the instance of the context class of the psm which owns the psm instance. For the initial state only an unnamed transition or a transition with the event `create` is allowed as an outgoing transition. An initial state has no incoming transitions while an end state has no outgoing transitions. The transitions between states specify the valid call sequences of operations for the context class. As described in the UML the protocol state transitions between states consist of three parts:

1. the referred operation (`op`),
2. an optional guard (`G`), i. e., a precondition and
3. a postcondition (`PC`) which is also optional.

In an state machine diagram the transitions are labeled using the following schema: $\xrightarrow{\texttt{[G] op()/ [PC]}}$. A state can have multiple outgoing transitions that refer

to the same operation. To be still able to choose a single transition the guard, post condition and state invariant of the target state for all transitions referring to the same operations are considered by USE. In some situation the usage of all this information still leads to multiple possible transitions. When USE encounters such a situation it reports an error to the user.

When an operation on an object whose class defines at least one psm is called, the selection of the transition to be taken for each psm is done in the following way. First, it is checked if the operation call needs to be ignored, i. e., no transition must be taken. This is the case if

- none of the transitions inside the protocol state machine covers the called operation (see [20, p. 545]) or
- the psm is not in a stable state, i. e., a transition is currently active.

If the operation cannot be ignored it is checked

- if at least one outgoing transition of the current state is enabled, i. e., the state has one or more outgoing transitions which refer to the called operation while having a valid precondition.

All enabled transitions are saved as possible transitions which could be taken after the operation call is completed. When the called operation finishes its execution, for all possible transitions the postcondition and the state invariant of the target state are validated. If only one transition fulfills the postcondition and the state invariant the transition is taken. Otherwise an error is reported which also explains if either no transition could be taken or multiple transitions would be possible.

3.1 State Determination

One benefit of our monitoring approach is the possibility to connect to a monitored system at any time. While this allows a SUM to run without overhead until the monitoring starts, this ability leads to some issues to be considered. One major problem is the lack of information of previously called operations, so that all protocol state machine instances are in an undefined state. To allow a correct monitoring of psms it is important to determine the correct states of all psm instances. To be able to determine the states after an initial snapshot has been taken we use state invariants. These state invariants need to be well-defined because otherwise the snapshot would be in an unsound state. For example, all psm instances should be in a given state after the state determination check. In this context well-defined means that the state invariants should be independent of each other, i. e., at any state only one state invariant evaluates to true for every instance referring to the psm.

When using complex state invariants the task of verifying the independence of state invariants can be accomplished by using automatic model finding techniques. These are similar to the one presented in [13] which allows a user to show the independence of invariants. In [13] the independence of invariants is

slightly different form the independence of state invariants we want to achieve. In [13] an invariant is defined as independent if it cannot be removed without loss of information meaning, there exists at least one system state where this single invariant is violated. For the independence of state invariants required for the state determination, we consider state invariants as independent if for all system states only a single state invariant is fulfilled.

Formally, given the set of all possible system states $\sigma(M)$ of a Model M and the invariants $i_1, \ldots, i_n$ the independence of an invariant i_k is defined in [13] as

$$\exists \sigma \in \sigma(M)(\sigma(i_1) \wedge \cdots \wedge \sigma(i_{k-1}) \wedge \sigma(i_{k+1}) \wedge \cdots \wedge \sigma(i_n) \wedge \neg\sigma(i_k))$$

whereas in this work the independence of state invariants $i_1, \ldots, i_n$ for a single psm is defined as

$$\forall \sigma \in \sigma(M)(\sigma(i_k) \Rightarrow \neg\sigma(i_1) \wedge \cdots \wedge \neg\sigma(i_{k-1}) \wedge \neg\sigma(i_{k+1}) \wedge \cdots \wedge \neg\sigma(i_n))$$

However, the same validation techniques apply, but as the universal quantification indicates, a full verification requires a complete search through all possible system states, which implies the well-known state space explosion problem and is therefore not a trivial task and we restrict ourselves to checking occurring test cases.

4 Case Study

In this section we apply our extensions to the monitoring approach to the public available, mid-sized application we used in [15]. The case study will demonstrate the advantages of our approach.

- Assumptions about a running implementation can be validated without the need to modify the source code.
- The state of an implementation can be examined in an abstract way to discover inconsistencies or design decisions.
- Using protocol state machines the correct usage of the defined protocol of a class can be validated.
- Concrete usage scenarios can be visualized by means of a sequence diagram.

This will be exemplified by the following case study using an open source computer game called *Free Colonization*[1] or in short *FreeCol*. It is a modern Java-based implementation of the 1994 published game *Sid Meier's Colonization*[2]. The game itself is a round-based strategy game with the goal to colonize America and finally to achieve independence. The game takes place on a matrix-like map which consists of tiles with different types, e. g., water, mountain, forest. Different units operate on this map and can explore unknown territory, build

[1] Project website: `http://www.freecol.org`

[2] The corresponding Wikipedia article gives detailed information about the game play. `http://en.wikipedia.org/wiki/Sid_Meier%27s_Colonization`

colonies, trade goods, etc. Figure 3 shows an example state transition of a running game. One unit (i. e., a pioneer) is placed in the center of the shown map on the left side and is surrounded by several different tile types. The right map shows the game state after the pioneer has build a new colony called Jamestown. The sketched state machines displayed below the two maps exemplify our new contribution. We want to be able to monitor the transition of the pioneer state from one state before she or he built a colony to another state after she or he joined the colony (note, that this is a single step in the game).

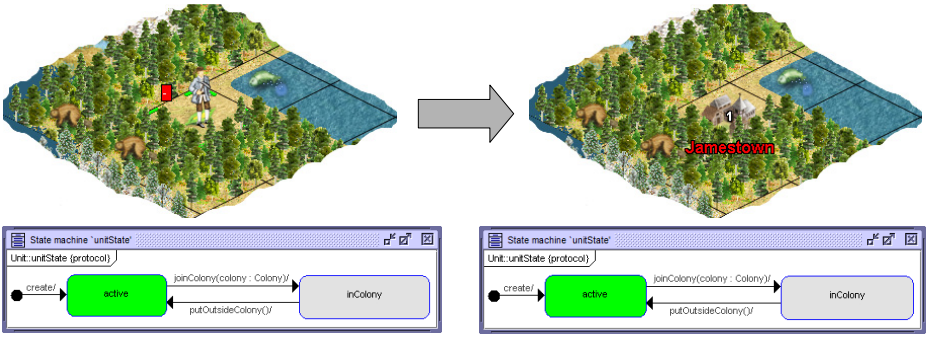


Fig. 3. Sample game situation in FreeCol

To be able to monitor this transition, we extended the PAM presented in [15] in a step-wise manner. First we added the enumeration `UnitState` to our PAM and defined a new attribute `state:UnitState` to the class `Unit` as shown in Fig. 4. The presence of this attribute simplified the definition of the state invariants as we will see later.

For our purpose the class diagram shown in Fig. 4 with an overall of 14 classes is detailed enough. When compared to the 551 classes which are present in version 0.9.2 of FreeCol we used for the monitoring this illustrates that the PAM for an application only needs to represent a small subset of the monitored implementation. Because we focus on state transitions we do not show any constraints defined for the PAM. Examples can also be found in [15].

Except for one case, we modeled attributes of a Java class which use a class present in the PAM as associations. The exception is the attribute `Tile::type` which reduces the number of links in an object diagram, but still allows to directly see that tiles differ in their type. For a first definition of a psm which monitors the entrance and the exit of a colony for a unit, we only need to consider the class `Unit` and its operations `joinColony(aColony:Colony)` and `putOutsideColony()` in combination with the attribute `state:UnitState`. These elements are present in the concrete implementation of the class `Unit` and can directly be monitored. The enumeration `UnitState` defines nine different enumeration literals which express different states of a unit. Since we are only interested in the state `IN_COLONY` and do not consider the other states we

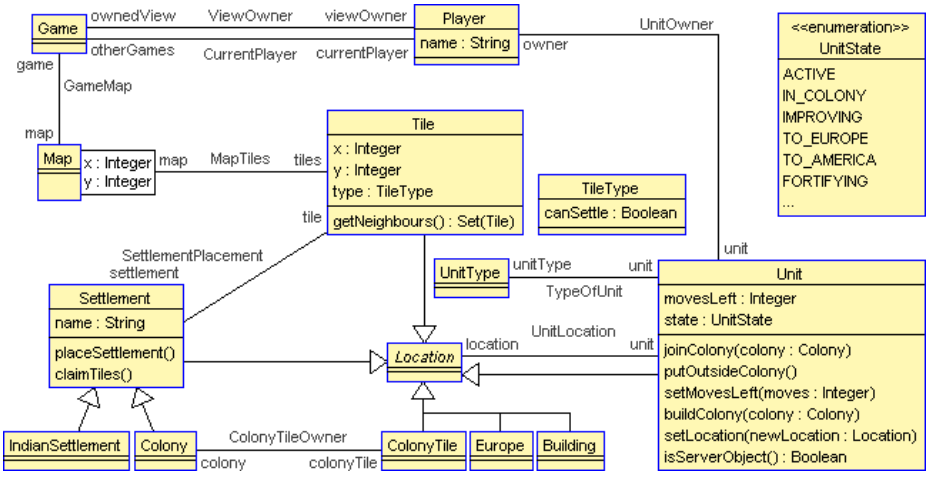


Fig. 4. Platform aligned model

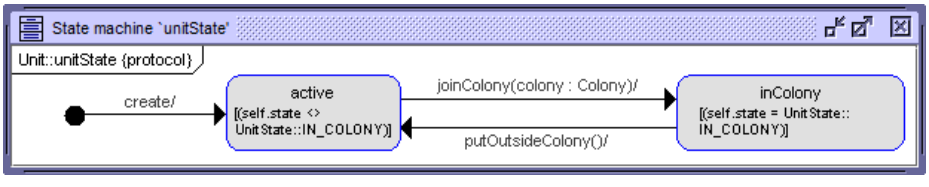


Fig. 5. Protocol state machine for the class Unit

can specify a protocol machine with two states. One for the state `IN_COLONY` and one for all other game states. Our assumption about the protocol of the class `Unit` is that an operation call to `putOutsideColony()` is only valid after the operation `joinColony()` has been called on the same object sometime before.

To be able to set the correct state of a monitored instance we need to specify state invariants for these two states. As stated earlier, the presence of the attribute `state` for the class `Unit` simplifies this task, because we only need to check the value of the attribute to determine the current state after a snapshot has been taken. Therefore, the state invariant for the psm state `inColony` is `self.state = UnitState::IN_COLONY` and the other state invariant only changes the comparison from equal to not equal. Given the previously expressed assumptions, this leads to a psm which has two states and two transitions leaving out the transition for the creation. This psm is shown in Fig. 5. A `Unit` object starts in the state `active` after it is created and enters the state `inColony` when the operation `joinColony(colony:Colony)` was executed. If the operation `putOutsideColony()` is called the state changes back to `active`. Any other operation call to a unit instance is ignored as described in the UML specification for operation not mentioned in a psm. This means, the psm only allows a state change when one of the two operations is called.

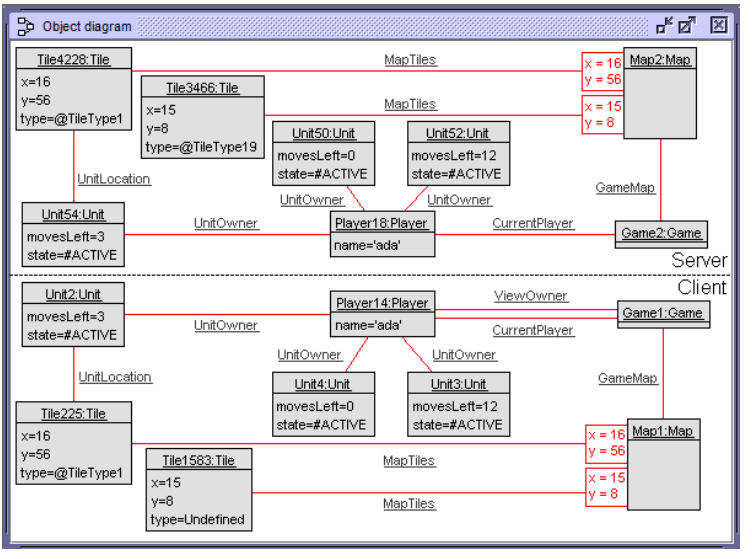


Fig. 6. Parts of the snapshot taken at runtime

Figure 6 shows the relevant part of the snapshot after connecting to the running game when it is in the game state shown on the left of Fig. 3 as an object diagram. The overall snapshot consists of nearly 6000 objects and 4000 links which makes it impossible to manually extract an informative object diagram. USE allows a user to select objects which should be shown or hidden in an object diagram by using several features. Two useful ones are the selection by an OCL expression and the selection of related objects by path length (see [12] for more information). The shown part of the snapshot is divided into two parts, which are important while validating the assumptions about the state transitions. Because we monitored a single user game on a single machine the instance of the game contains both, the data used by the game server and the client. By looking at the instances `Tile3466` on the server side and `Tile1583` on the client side one can see that the server part has more information about the game than the client part. Both instances represent the same tile on a map, because their positions are equal, but the client instance does not know of what type the tile is. To be able to determine if an object belongs to the server or client side we also monitored the class game with the association `ViewOwner`. If a game object is not linked to a player by this association it is the server game. The equivalent OCL expression (`self.owner.ownedView->isEmpty()`) is used as a body for the operation `isServerObject()` of the class `Unit`. This operation is marked as a query operation and is therefore ignored by the monitor. The object diagram further shows the owned units of the player named 'ada' and the object for the tile on which we want to build a colony (`Tile4228` resp. `Tile225`).

After taking this initial snapshot, the states of the protocol state machines for the existing unit objects need to be determined. This can be done by a single

command in USE which also informs the user about objects for which the psm instance could not be set to a single state. This happens, if no state invariant or multiple state invariants evaluate to true w. r. t. the given snapshot. Because this state determination is a common task after a snapshot has been taken, the monitor plugin can automatically execute the state determination after the construction of a snapshot. After the states have been determined the states of the relevant units of the snapshot are as expected (`active`). After resuming the game and building the new colony *Jamestown* we get a valid sequence of operation calls which can be seen in the monitored sequence diagram shown in Fig. 7. We observed, that the execution of the operation `joinColony()` indeed leads to the attribute value `IN_COLONY` of the attribute `Unit::state`, because no violation of a transition is reported.

To get further information about our assumptions we can instruct USE to validate the current state invariants of all psm instances. After the validation of our current snapshot USE reports an error for the psm instance of the client object of the unit which has built the colony. This is due to the fact that the operation `buildColony` is only called on the server object and only the new values are transfered to the client object. Therefore, USE did not execute a transition from the source state `active` to the target state `inColony` for the client unit but monitored the change of the attribute `state` to `IN_COLONY`. Now, the new attribute value violates the state invariant of the state `active`.

Because the separation of the client and server objects seems to be a valid design decision we can ignore these violations and continue the monitoring process to retrieve further information about the validity of our assumptions. To test the defined protocol we use another unit and let it join and exit the colony. While executing this scenario another issue arises because entering an existing colony, i. e., a unit only enters a colony without building it before, does not lead to an operation call to `joinColony()`. Instead, only `setLocation()` is called which is not handled by the psm and therefore does not execute a transition keeping the psm instance in the state `active`, but the attribute value of the runtime instance is set to `IN_COLONY` which violates the state invariant of the state `active`.

Using this information a user of the monitor needs to decide where the error is located: in the implementation or in the PAM. For our example, we assume that the PAM needs to be modified although it seems to be an unsound usage of the `Unit` class. This assumption is backed by the fact, that the developers of FreeCol refactored this part of the game in newer releases. If we want to adapt our psm to the last discovered facts, we need to handle the client server separation and the additional operation calls. The modified psm is shown in Fig. 8. The additional operation `setLocation(newLocation:Location)` leads to two new transitions in the psm. Both transitions have as their source state the state `active` but differ in their target and guard. If the new location is of type `ColonyTile`, which represents special tiles related to a colony, the new state after the execution is `inColony` otherwise the state does not change. Interestingly, when a `Unit` object leaves a colony this leads always to a call to `putOutsideColony()`.

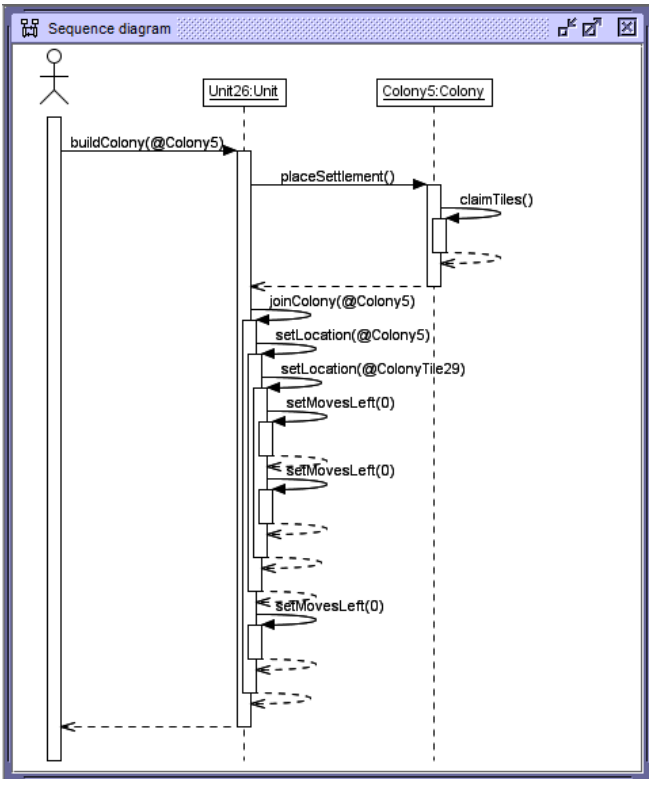


Fig. 7. Sequence diagram of the monitored execution

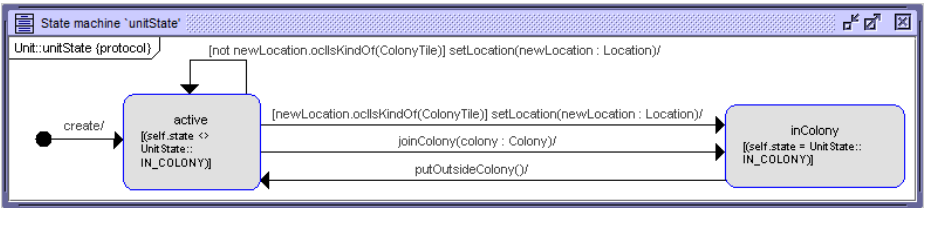


Fig. 8. Extended PSM for the class Unit

However, the problem how to differentiate the server and client objects still exists. When taking a snapshot all state invariants should be independent to allow a valid determination of the current state. If we would introduce a new state for client objects with the state invariant `not self.isServerObject()` and add a new conjunction `self.isServerObject()` to the two existing state invariants the state determination would work, because each instance can be mapped to a single state. Either it is a client object or it is a server object and the two server states are independent because of the different comparison

operators. The problem with this approach is the dynamic monitoring after existing instances are read. A new instance would be in the state `active` which is now only defined for a server object due to the new conjunction. If a new client instance is created this would violate the state invariant. Further, the new client instance would never change the state because none of the monitored operations is called for client objects as described before. Using an implies condition would violate the independence of the two server states because for client objects both invariants would be fulfilled.

A simple solution would be to add a transition which covers the operation that specifies the new instance to be a client instance. However, FreeCol does this inside of the constructor which is represented as the `create` transition and the UML explicitly forbids multiple outgoing transitions for the initial state and also no post condition on it. If it was allowed the distinction could be done using postconditions on two different create transitions. Another solution for this is to use a change event on a transition. By specifying the change expression `not self.isServerObject()` a psm instance can move to a specify client state. A drawback of this solution is the relative high calculation cost of such change events. Because a change expression can generally access every object or property of the current snapshot all currently valid transitions with change events would need to be checked after every change in the snapshot. The costs can be reduced by using special analysis algorithms which calculate the change expressions that need to be checked after a change as presented in [7]. Currently, such a mechanism is not present in USE, but could be integrated in the future. For now, we need to ignore state violations of the client objects. Note, that the validation of the correct transitions for the server objects still works.

When using this modified psm all scenarios described above lead to the expected changes of the psm states. Beside the manual execution of observed game situations the presence of computer controlled players in the game can be used as a test driver. As with the manual play all analyzed operations are also used by computer controlled players. We used this to strengthen our PAM.

5 Related Work

In our previous work we focused on the runtime verification of static properties (like multiplicity constraints and invariants) of an application running on a virtual machine [15]. Different approaches for checking information extracted from a running system for certain properties exist. [10] and [2] make a comparison between these approaches. According to [10] most constraint validation techniques for Java are based on the design-by-contract-principle introduced by the Eiffel programming language. In contrast to our approach, the approaches compared to each other in [2] require a full access to the source code of the system under monitoring. The Java Modeling Language (JML) is appropriate both for formal verification and runtime assertion checking [18].

In this paper, we extended our validation engine by support for UML protocol state machines in order to be able to verify and validate dynamic sequences of

operation calls. Our approach applying protocol state machines differentiates from approaches which are based on the usage of regular expressions. Such an approach is presented in [5]. It enables programmers to define parameterized runtime monitors. For this purpose a temporal ordering over breakpoints, which are used for debugging purposes by programmers, is introduced. The temporal ordering is defined by regular expressions. Another approach uses tracematches for runtime verification [6]. As the previous approaches, this one is also based on regular expressions.

A UML protocol state machine as used in our approach is different from regular expressions through the information of transitions: protocol state machines provide the possibility to specify an initial condition (guard) under which an operation can be called. This possibility makes protocol state machines more powerful than regular expressions. The authors of [23] present an approach which applies UML protocol state machines to produce class contracts. For this purpose they define the structure and the semantics of UML protocol state machines.

With 'ocl2j' a tool exists which allows to enforce OCL constraints in Java through translating OCL expressions into Java code [9]. An analog approach is presented e. g. in [14]. From the authors runtime verification approach the tool 'INVCOP' has arised. The Dresden OCL toolkit makes available two distinctive approaches for OCL-based runtime verification [8]. While the 'generative' approach is based on the generation of AspectJ code, the 'interpretative' approach integrates the Dresden OCL2 Interpreter into a runtime environment in order to interpret OCL constraints.

In [4] the monitoring of state machines is focused while the usage of OCL is relinguished. With the 'aspect oriented approach', the 'listener approach' and the 'debugging approach', the authors describe three possibilities to extract runtime models.

To synchronize a running system with a runtime model the authors of [26] use 'synchronizers'. Thus the system can be changed immediately when the model has been updated and the model can be immediately adapted if the system progresses.

Java PathFinder (JPF) is a runtime verification and testing environment for Java developed at NASA Ames Research Center [27]. JPF is based upon a special Java Virtual Machine which is called from a model checking engine included in JPF. The authors of [1] present JPF-SE, a symbolic execution extension to JPF. The framework Polyglot has been integrated with the Java PathFinder [3]. Polyglot enables the execution of multiple variants of statecharts including UML statecharts and the verification of their models against properties. It uses an intermediate representation which is translated from a range of modeling tools. The intermediate representation is used to generate Java code representing the structure of a statechart which is analyzed by applying JPF.

6 Conclusion

We have presented an extension to our approach for monitoring assumed properties in form of OCL constraints for a running Java application. Based on this

approach, which takes advantage of the powerful features of the Java virtual machine, we have added support for protocol state machines to the underlying validation engine. This allows us to specify assumptions not only formulated as class invariants or operation contracts, but also as state invariants. By using a protocol state machine, more knowledge about the history of an object is available because of the recording of states. We have shown that the definition of state invariants is important for our approach in order to determine the correct states of an object when connecting to a running system without the information about previous operation calls. We explained our work by a non-trivial example of an open-source game.

As future work we want to extend the support for protocol state machines within our validation engine. One major improvement would be the support for change events. To be applicable in practice, an efficient implementation is needed which considers only the transitions with an effective change event. A more detailed study of similar approaches, for example, based on aspect-orientation or approaches considering the Java Modeling Language (JML) as a target language, might introduce alternative features and our monitor could be improved in various directions. For example, one could consider abstract model breakpoints, which are configurable by the user or by extended information about elements that are only present within the running system. Last, but not least, comprehensive case studies must give more feedback about the applicability of our work.

References

1. Anand, S., Păsăreanu, C.S., Visser, W.: JPF–SE: A Symbolic Execution Extension to Java PathFinder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 134–138. Springer, Heidelberg (2007)
2. Avila, C., Sarcar, A., Cheon, Y., Yeep, C.: Runtime Constraint Checking Approaches for OCL, A Critical Comparison. In: SEKE (2010)
3. Balasubramanian, D., Pasareanu, C.S., Whalen, M.W., Karsai, G., Lowry, M.R.: Polyglot: modeling and analysis for multiple Statechart formalisms. In: Dwyer, M.B., Tip, F. (eds.) ISSTA, pp. 45–55. ACM (2011)
4. Balz, M., Striewe, M., Goedicke, M.: Monitoring Model Specifications in Program Code Patterns. In: Proc. of the 5th Int. WS Models@run.time, pp. 60–71 (2010)
5. Bodden, E.: Stateful breakpoints: a practical approach to defining parameterized runtime monitors. In: ESEC/FSE 2011. ACM, New York (2011)
6. Bodden, E., Hendren, L.J., Lam, P., Lhoták, O., Naeem, N.A.: Collaborative Runtime Verification with Tracematches. J. Log. Comput. 20(3), 707–723 (2010)
7. Cabot, J., Teniente, E.: Incremental integrity checking of UML/OCL conceptual schemas. Journal of Systems and Software 82(9), 1459–1478 (2009)
8. Demuth, B., Wilke, C.: Model and object verification by using Dresden OCL. In: Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice, Ufa, Russia, pp. 687–690 (2009)
9. Dzidek, W.J., Briand, L.C., Labiche, Y.: Lessons Learned from Developing a Dynamic OCL Constraint Enforcement Tool for Java. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 10–19. Springer, Heidelberg (2006)

10. Froihofer, L., Glos, G., Osrael, J., Goeschka, K.M.: Overview and Evaluation of Constraint Validation Approaches in Java. In: Proc. of ICSE 2007, pp. 313–322. IEEE Computer Society, Washington, DC (2007)
11. 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)
12. Gogolla, M., Hamann, L., Xu, J., Zhang, J.: Exploring (Meta-)Model Snapshots by Combining Visual and Textual Techniques. In: Proc. 10th Int. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2011) (2011)
13. Gogolla, M., Kuhlmann, M., Hamann, L.: Consistency, Independence and Consequences in UML and OCL Models. In: Dubois, C. (ed.) TAP 2009. LNCS, vol. 5668, pp. 90–104. Springer, Heidelberg (2009)
14. Gopinathan, M., Rajamani, S.K.: Runtime Monitoring of Object Invariants with Guarantee. In: Leucker, M. (ed.) RV 2008. LNCS, vol. 5289, pp. 158–172. Springer, Heidelberg (2008)
15. Hamann, L., Gogolla, M., Kuhlmann, M.: OCL-Based Runtime Monitoring of JVM Hosted Applications. In: Proc. WS OCL and Textual Modelling. ECEASST (2011)
16. Hamann, L., Vidács, L., Gogolla, M., Kuhlmann, M.: Abstract Runtime Monitoring with USE. In: Proc. CSMR 2012, pp. 549–552 (2012)
17. Katoen, J.P., Baier, C.: Principles of Model Checking. MIT Press (2008)
18. Leavens, G.T., Cheon, Y., Clifton, C., Ruby, C., Cok, D.R.: How the design of JML accommodates both runtime assertion checking and formal verification. Sci. Comput. Program. 55(1-3), 185–208 (2005)
19. 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)
20. UML Superstructure 2.2. Object Management Group (OMG) (February 2009), http://www.omg.org/spec/UML/2.2/Superstructure/PDF/
21. Object Constraint Language 2.2. Object Management Group (OMG) (February 2010), http://www.omg.org/spec/OCL/2.2/
22. Oracle: Java™ Platform Debugger Architecture - Structure Overview (2011), http://download.oracle.com/javase/6/docs/technotes/guides/jpda/architecture.html
23. Porres, I., Rauf, I.: From Nondeterministic UML Protocol Statemachines to Class Contracts. In: Int. Conf. on Software Testing, Verification, and Validation, pp. 107–116. IEEE Computer Society, Los Alamitos (2010)
24. Shah, S.M.A., Anastasakis, K., Bordbar, B.: From UML to Alloy and Back Again. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 158–171. Springer, Heidelberg (2010)
25. Soeken, M., Wille, R., Drechsler, R.: Encoding OCL Data Types for SAT-Based Verification of UML/OCL Models. In: Gogolla, M., Wolff, B. (eds.) TAP 2011. LNCS, vol. 6706, pp. 152–170. Springer, Heidelberg (2011)
26. Song, H., Huang, G., Chauvel, F., Sun, Y.: Applying MDE Tools at Runtime: Experiments upon Runtime Models. In: Models@run.time, pp. 25–36 (2010)
27. Visser, W., Havelund, K., Brat, G.P., Park, S., Lerda, F.: Model Checking Programs. Autom. Softw. Eng. 10(2), 203–232 (2003)
28. Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modeling with UML, 2nd edn. Addison-Wesley (2003)

On Model Subtyping

Clément Guy[1], Benoît Combemale[1], Steven Derrien[1],
Jim R.H. Steel[2], and Jean-Marc Jézéquel[1]

[1] University of Rennes1, IRISA/INRIA, France
[2] University of Queensland, Australia

Abstract. Various approaches have recently been proposed to ease the manipulation of models for specific purposes (*e.g.,* automatic model adaptation or reuse of model transformations). Such approaches raise the need for a unified theory that would ease their combination, but would also outline the scope of what can be expected in terms of engineering to put model manipulation into action. In this work, we address this problem from the model substitutability point of view, through model typing. We introduce four mechanisms to achieve model substitutability, each formally defined by a subtyping relation. We then discuss how to declare and check these subtyping relations. This work provides a formal reference specification establishing a family of model-oriented type systems. These type systems enable many facilities that are well known at the programming language level. Such facilities range from abstraction, reuse and safety to impact analyses and auto-completion.

Keywords: SLE, Modeling Languages, Model Typing, Model Substitutability.

1 Introduction

The growing use of Model Driven Engineering (MDE) and the increasing number of modeling languages has led software engineers to define more and more operators to manipulate models. These operators are defined in terms of model transformations expressed at the language level, on the corresponding metamodel. However, new modeling languages are still generally designed and tooled from scratch with few possibilities to reuse structure or model manipulations from existing modeling languages.

To address the need for a more systematic *engineering* of model transformations, various approaches have recently been proposed. These approaches include model transformation reuse [1,2,3,4,5,6] and automatic model adaptation [7,8,9,10]. Although these approaches do meet their goals, they remain somewhat disconnected from each other, and lack a unified theory enabling both their combination and comparison. Such a formalization would also help defining the scope of what can be expected (from a engineering point of view) to put model manipulation into action.

In this paper, we tackle the problem from the model substitutability point of view, through model typing. Model typing provides a well-defined theory that considers models as first-class entities, and typed by their respective model types [3]. In addition to the previous work on model typing focusing on the typing relation (*i.e.*, between a model

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 400–415, 2012.

and its model types), we introduce four model subtyping relations. These relations provide model substitutability, that is they enable a model typed by A to be safely used where a model typed by B is expected, A and B being model types.

This work provides a formal reference specification establishing a family of model type systems. These type systems enable many facilities that are well known at the programming language level, ranging from abstraction, reuse and safety to auto-completion.

This paper is structured as follows. We first illustrate the need for a systematic engineering for model manipulation using examples from the optimizing compilation community (Section 2). We then provide some background on MDE and model typing as introduced by Steel *et al.* [3] in Section 3. In Section 4 we formally define four model subtyping relations, based on two criteria: the structure and the considered subset of the involved model types. Section 5 addresses the ways to declare and check these subtyping relations. Section 6 classifies existing approaches providing reuse facilities for model manipulation with respect to the specification of model-oriented type systems provided in sections 4 and 5. Finally, Section 7 concludes the paper and summarizes our ideas for future work.

2 Illustrative Examples

Optimizing compilers have always used abstractions (*i.e.*, models) of the compiled program to apply numerous analyses, which are often tedious to implement. Thus, optimizing compilation seems a good candidate when looking for a domain in which model manipulation engineering facilities would be valuable.

Most analyses performed by optimizing compilers leverage sophisticated algorithms implemented on different types of compilers' intermediate representations (IRs). Implementation of such algorithms is known to be a tedious and error-prone task. As a consequence, providing modularity and reuse is a crucial issue for improving compiler quality but also compiler designer productivity.

Dead Code Elimination (DCE) is an example of such an algorithm. It is a classical compiler optimization which removes unreachable code from the control flow graph of a program (*e.g.*, the `else` branch of a `if` whose condition is always true) [11]. Although the DCE algorithm is relatively straightforward, there exist more complex analyses, such as those leveraging abstract interpretation techniques [12]. Such analyses can infer accurate invariants over the values of the variables of the analyzed program. These invariants can then be used to detect possible program errors (*e.g.*, buffer overflows) or to offer program optimization opportunities.

As with many compiler optimizations, the scope of applicability of these algorithms is wide, including most imperative programming languages. We consider three examples of such languages: two compiler IRs, the GeCoS and the ORCC IRs; and a toy procedural language, Simple. GeCoS[1] is a retargetable C compiler infrastructure targeted at embedded processors. ORCC[2] is a compiler for CAL[3], a dataflow actor language in which actions are described with a standard imperative semantics. Both IRs are

[1] Cf. `http://gecos.gforge.inria.fr`
[2] Cf. `http://orcc.sourceforge.net/`
[3] Cf. `http://ptolemy.eecs.berkeley.edu/papers/03/Cal/index.htm`

metamodel-based: models conforming to these metamodels are used as abstractions of the compiled program. Finally, Simple is a language on which is defined P-Interproc[4], an interprocedural analyzer implementing several abstract interpretation analyses.

Rather than being reimplemented for each targeted languages (*e.g.*, GeCoS IR, ORCC IR and Simple), we would expect DCE and our abstract interpretation analyses to be defined once and then reused across these languages. Of course, this reuse should be done safely and should be transparent for the programmer. More precisely, the only thing the programmer should care about when building his compilation flow is whether his model is eligible (or not) for a given model manipulation.

Facilities such as abstraction (hiding unnecessary details from the programmer), reuse (sharing a model manipulation between different metamodels), and safety (forbidding erroneous reuse) could be provided by a type system specifically targeted at models. Such a model-oriented type system would greatly help in increasing both programmers productivity and model-oriented software quality.

3 Background

In this section, we first present the MOF (Meta-Object Facility) metalanguage, the basis for metamodels, and thus model manipulation operators. We then present model types as introduced by Steel *et al.* [3] on which we base our model subtyping relations. Finally, we discuss the limits of the model subtyping relation proposed by Steel *et al.*

3.1 Model Driven Engineering

The Meta-Object Facility. (MOF) [13] is the OMG's standardized meta-language, *i.e.*, a language to define metamodels. As such it is a common basis for a vast majority of model-oriented languages and tools. A metamodel defines a set of models on which it is possible to apply common operators. Therefore, model substitutability must take into account MOF and the way it expresses metamodels.

Figure 1 displays the structure of EMOF (Essential MOF) which contains the core of the MOF meta-language in the form of a class diagram. EMOF supports the definition of the concepts and relationships of a metamodel using `Classes` and `Propertys`. `Classes` can be abstract (*i.e.*, they cannot be instantiated) and have `Propertys` and `Operations`, which declare respectively attributes and references, and the signatures of methods available from the modeled concept. `Classes` can have several superclasses, from which they inherit all `Propertys` and `Operations`. A `Property` can be composite (an object can only be referenced through one composite `Property` at a given instant), derived (*i.e.*, calculated from other `Propertys`) and read-only (*i.e.*, cannot be modified). A `Property` can also have an opposite `Property` with which it forms a bidirectional association. An `Operation` declares the `Types` of the exceptions it can raise and ordered `Parameters`. `Propertys`, `Operations`, and `Parameters` are `TypedElements`; their type can be either: a `Datatype` (*e.g.*, `Boolean`, `String`, etc.) or a `Class`. `Parameters`, `Propertys` and `Operations` are `Multiplicity Elements`. As such, they have a multiplicity (defined by a lower and an upper bound), as well as orderedness and uniqueness.

[4] Cf. `http://pop-art.inrialpes.fr/interproc/pinterprocweb.cgi`

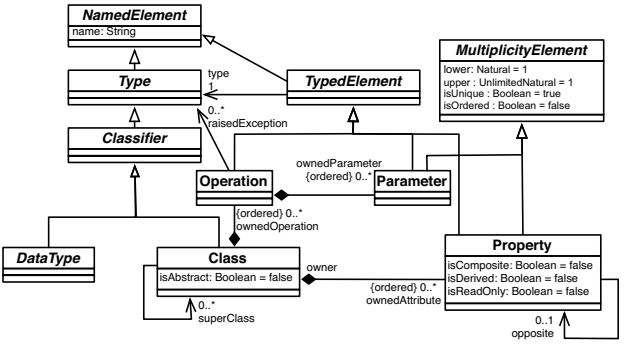


Fig. 1. The EMOF core with class diagram notation

Metamodels can be seen as class diagrams, each of their concepts being instantiable by objects belonging to models. However, metamodel concepts are also instances of MOF elements and thus a metamodel can be drawn as an object diagram where each concept is an instance of one of the MOF elements (*e.g.*, classes `Class` or `Property`).

Because model subtyping takes place at the metamodel level, the latter representation facilitates the definition of model subtyping relations by depicting metamodels and their contained concepts as objects with attributes and properties. Thus, we will use the object diagram representation in preference to the more common class diagram one.

3.2 Model Typing

Model Types were introduced by Steel *et al.* [3], as an extension of object typing to provide abstraction from object types and enable model manipulation reuse.

Definition 1. (Model type) *A type of a model is a set of types of objects which may belong to the model, and their relations.*

MOF classes are closer to types (interfaces) than to object classes, thus a model type is closely related to a metamodel. The difference between model types and metamodels lies in their respective relations with models. A model has one and only one metamodel to which it conforms. This metamodel contains all the types needed to instantiate objects of the model. Conversely, a model can have several model types which are subsets of the model's metamodel.

Because model types and metamodels share the same structure, it is possible to extract the type of a model from its metamodel (we call the model type containing all the types from a model's metamodel the *exact type* of the model). Figure 2 represents a model m_1 which conforms to a metamodel MM_1 and is typed by model types MT_A and MT_B, MT_B being the *exact type* of m_1 extracted from MM_1. Both metamodels and model types conforms themselves to MOF.

MOF delegates the definitions of contracts (*e.g.*, pre and post-conditions or invariants) to other languages (*e.g.*, OCL, the Object Constraint Language [14]). Hence neither the original paper on model typing [3] nor this one considers contracts in subtyping relations, but focuses on features of object types contained by model types.

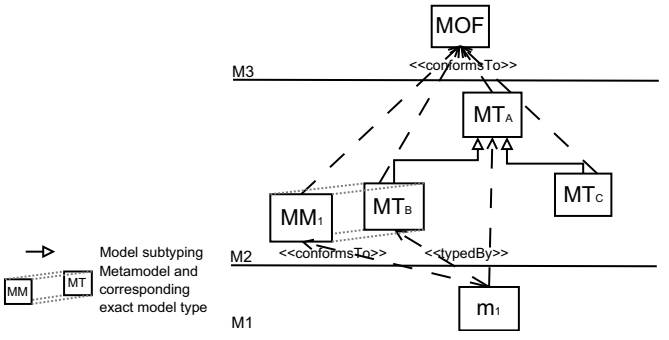


Fig. 2. Conformance, model typing and model subtyping relations

Substitutability is the ability to safely use an object of type *A* where an object of type *B* is expected. Substitutability is supported through subtyping in object-oriented languages. However, object subtyping does not handle type group specialization (*i.e.*, the possibility to specialize relations between several objects and thus groups of types).[5] Such type group specialization have been explored by Kühne in the context of MDE [16]. Kühne defines three model specialization relations (specification import, conceptual containment and subtyping) implying different level of compatibility. We are only interested here in the third one, subtyping, which requires an *uncompromised mutator forward-compatibility*, *e.g.*, substitutability, between instances of model types.

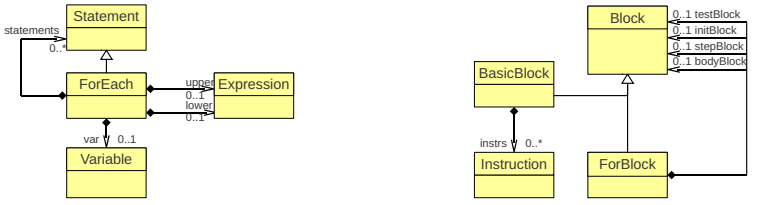
Model Type Matching is a model subtyping relation proposed by Steel *et al.* to enable safe model manipulation reuse in spite of limits of object subtyping. To this end, they use the *object type matching* relation defined by Bruce *et al.* [17], which is more flexible than subtyping. For more details, we refer the reader to Steel's PhD thesis [18].

Definition 2. (Model type matching proposed by Steel *et al.* [3]) *Model Type MT_B matches model type MT_A if for each object type C in MT_A there is a corresponding object type with the same name in MT_B such that every property and operation in $MT_A.C$ also occurs in $MT_B.C$ with exactly the same signature as in $MT_A.C$.*

Limits of Model Type Matching. However, *model type matching* as presented by Steel *et al.* is subject to some shortcomings. First, the type rules they present, and their implementation in Kermeta[6], violate their definition of *type matching* by permitting the relaxation of lower multiplicities, i.e. by allowing a non-mandatory attribute to be matched by a mandatory one, which could potentially lead to an invalid model.

In addition, and more significantly, finding a model type common to several model types (*e.g.*, GeCoS IR, ORCC IR and Simple) is not always possible, even if they share numerous concepts (*e.g.*, concepts used in DCE). This impossibility is due to structural heterogeneities between the metamodels [19]. Figure 3 presents such heterogeneities between excerpts from the GeCoS IR and the ORCC IR metamodels representing `foreach` (from ORCC IR) and `for` loops (from GeCoS IR, and thus C). The

[5] We refer the reader interested in the type group specialization problem to the Ernst's paper [15].
[6] Cf. `http://www.kermeta.org`



(a) `foreach` loop in ORCC IR (b) `for` loop in GeCoS IR

Fig. 3. Extracts of ORCC IR and GeCoS IR metamodels

former (Figure 3(a)) is a simpler loop than the latter (Figure 3(b)), iterating only by steps of one on a given variable between bounds, where a C `for` can have complete code blocks as initialization, step and test.

Thus to reuse a model manipulation (*e.g.*, DCE), a subtyping mechanism should provide for the definition of an adaptation, needed to *bind* different structures to a single one on which the manipulation is defined. In our example, such an adaptation could be the transformation of `foreach` loops into more generic `for` loops, using the variable and the lower bound to produce an initialization block, the variable and the upper bound to produce a test block, and automatically producing a step block with a step of one.

This adaptation should be able to adapt back the result of the manipulation, because this manipulation could modify the model it processes or return a result containing elements of the model. For example, DCE modifies the representation of the program by removing code. Once the optimization has been processed on a common representation, it should be possible to adapt back the structure to impact the result of DCE in the original structure (*i.e.*, an ORCC IR or GeCoS IR model).

Although defining common optimizations on a minimal dedicated structure seems to best fit the need for modularity and reuse, we need to consider the presence of legacy code. For example, DCE is already implemented for the GeCoS IR. Reusing this implementation on ORCC IR would avoid the creation of a generic model type and the reimplementation of the optimization. However, the GeCoS IR does not contain only the concepts required for DCE. More particularly, it contains concepts which do not exist in ORCC IR (*e.g.*, pointers). Therefore, a model subtyping mechanism should be able to accept a subtype which only possesses the concepts of the supertype required for the reuse of a specific model manipulation.

4 Model Subtyping Relations

Object-oriented type systems provide important systematic engineering facilities, including abstraction, reuse and safety. We strongly believe that these facilities can also be provided for model manipulation through a model-oriented type system. However, the existing model subtyping relation has shown some limitations.

For this reason, in this section we review four subtyping relations between model types, based on two criteria: the presence of heterogeneities between the two model

types (Subsections 4.1 and 4.2) and the considered subset of the model types (Subsections 4.3 and 4.4). Such a model subtyping relation is pictured in Figure 2 by the generalization arrow between model types MT_A and MT_B. Through this subtyping relation, models typed by MT_A are substitutable to models typed by MT_B, *i.e.*, model manipulations defined on MT_B can be *safely reused* on model typed by MT_A.

4.1 Isomorphic Model Subtyping

An obvious way to safely reuse on a model typed by MT_B a model manipulation from a model type MT_A is to ensure that MT_B contains substitutable concepts (*e.g.*, classes, properties, operations) for those contained by MT_A. As mentioned in Section 3, it is not possible to achieve type group (or model type) substitutability through object subtyping.

MOF Class Matching. Thus, we use an extended definition of *object type matching* introduced by Bruce *et al.* [17] and used by Steel *et al.* to define their *model type matching* relation. Our *object type matching* relation is similar to, but stricter than the latter, because class names must be the same, as must lower and upper bounds of multiplicity elements. Moreover, every mandatory property in the matching type requires a corresponding property in the matched type, in order to prevent model manipulation from instantiating a type without its mandatory properties.

Definition 3. (MOF class matching) *MOF class T' matches T (written $T' <\# T$) iff:*

1. $T.name = T'.name$
2. $T'.isAbstract \Rightarrow T.isAbstract$
3. $\forall op \in T.ownedOperation, \exists S' \in SuperClasses(T')$ *such that* $\exists op' \in S'.ownedOperation$ *and:*
   - 3.1 $op.name = op'.name$
   - 3.2 $op'.type <\# op.type \vee op.type <: op'.type$
   - 3.3 $\forall p \in op.ownedParameter, \exists p' \in op'.ownedParameter$ *such that:*
     - (a) $\exists U' \in SubClasses(p'.type)$ *such that* $U' <\# p.type \vee p.type <: p'.type$
     - (b) $p.rank = p'.rank$
     - (c) $p.lower = p'.lower$
     - (d) $p.upper = p'.upper$
     - (e) $p.isUnique = p'.isUnique$
   - 3.4 $\forall e' \in op'.raisedException, \exists e \in op.raisedException$ *such that* $e' <\# e \vee e' <: e$
4. $\forall a \in T.ownedAttribute, \exists S' \in SuperClasses(T')$ *such that* $\exists a' \in S'.ownedAttribute$ *such that:*
   - 4.1 $a.name = a'.name$
   - 4.2 $a'.isReadOnly \Rightarrow a.isReadOnly$
   - 4.3 $a.isComposite = a'.isComposite$
   - 4.4 $a'.type <\# a.type \vee (a'.type <: a.type \wedge a.isReadOnly)$
   - 4.5 $a.lower = a'.lower$
   - 4.6 $a.upper = a'.upper$
   - 4.7 $a.opposite \neq void \Rightarrow a'.opposite \neq void \wedge a.opposite.name = a'.opposite.name$
   - 4.8 $a.isUnique = a'.isUnique$

5 $\forall a' \in T'.ownedAttribute, a'.lower > 0 \Rightarrow \exists S \in SuperClasses(T)$ *such that* $\exists a \in S.ownedAttribute \wedge a.name = a'.name$

Where $SuperClasses(T)$, is the set of all superclasses of T, $SubClasses(T)$, the set of all its subclasses, both including T and $<:$ is the object subtyping relation.

Model Type Matching. Given the conditions under which objects may be substitutable in the context of a model type, we can define a *model type matching* relation which ensures the safe type group substitutability. Based on the definition of MOF class matching, we redefine the *model type matching* relation as follows:

Definition 4. (Model type matching) *The* model type matching *relation is a binary relation* $\sqsubseteq$ *on ModelType, the set of all model types, such that* $(MT_B, MT_A) \in \sqsubseteq$ *(also written* $MT_B \sqsubseteq MT_A$*) iff* $\forall T_A \in MT_A, \exists T_B \in MT_B$ *such that* $T_B <\# T_A$.

The *model type matching* relation can be seen as a kind of subgraph isomorphism which takes into account the MOF specificities (*e.g.*, inherited properties and operations). For this reason we call *isomorphic* model subtyping relation a relation which satisfies the *matching* relation.

4.2 Non-isomorphic Model Subtyping

The fact that MT_B does not match MT_A does not mean that it is not appropriate for substitution. Indeed, the condition for safely substituting a model m for another is that m contains all the necessary information expected to be handled safely by the called model manipulation or to access the desired features. But this information can be under another form than expected (*e.g.*, with different class names) in which case m may be substitutable if the expected form of the information is retrieved.

Model Adaptation is the process of retrieving the information from a model in the form expected. It consists in adapting a model m_B into a model m_A which can be handled by the operation or through which it is possible to access to the desired feature. Thus a *model adaptation* is a way to *create* a *model type matching* relation between two model types. A model adaptation is a function defined at the model type level and applied on models. It takes a model m_B typed by MT_B and returns a model m_A with the same information, but in the form defined by MT_A, *i.e.*, a model whose type matches MT_A.

Definition 5. (Model adaptation) *A model adaptation is a function* $adapt_{MT_A}$ *from* MT_B *to* MT_C*, where* MT_A*,* MT_B *and* MT_C *are model types and such that* $MT_C \sqsubseteq MT_A$.

One way to achieve such an adaptation is to implement a model transformation from MT_B to MT_A, in which case $MT_C = MT_A$. Another way is by adding missing types and derived properties from MT_A to MT_B, creating a new model type MT_C with $MT_C \sqsubseteq MT_B$ and $MT_C \sqsubseteq MT_A$. This is the approach followed by Sen *et al.* [5].

Bidirectional Model Adaptation , that is coupled forward adaptation from MT_B to MT_A and backward adaptation from MT_A to MT_B may be needed, depending whether the adaptation is done to reuse an *endogenous* or an *exogenous* model manipulation [20].

If the adaptation to MT_A is done in order to reuse an *endogenous* manipulation, a backward adaptation is necessary in order to reflect changes made to the adapted model on the original model. Conversely, a backward adaptation is not necessary if the reused feature is an *exogenous* manipulation.

The forward and backward adaptation together form a bidirectional adaptation, which enables the adaptation of a model typed by MT_B into a form which fits the expected model type MT_A but also to reflect the result in the original model. Moreover, a roundtrip adaptation, *i.e.*, applying the forward adaptation then the backward adaptation to the result should lead to an unchanged model. To this end, we use here rules defined by Foster *et al.* for well-behaved lenses (*i.e.*, bidirectional transformation operators) [21].

Definition 6. (Bidirectional model adaptation) *A bidirectional model adaptation $adapt_{MT_A}$ between model types MT_B and MT_A comprises a function $adapt_{MT_A} \nearrow$ from MT_B to MT_C and a function $adapt_{MT_A} \searrow$ from $MT_B \times MT_C$ to MT_B, where MT_C is a model type such that $MT_C \sqsubseteq MT_A$ and:*

- $adapt_{MT_A} \nearrow (adapt_{MT_A} \searrow (m_B, m_C)) = m_C, \forall (m_B, m_C) \in MT_B \times MT_C$
- $adapt_{MT_A} \searrow (m_B, adapt_{MT_A} \nearrow (m_B)) = m_B, \forall m_B \in MT_B$

Bidirectional adaptation can be provided through bidirectional transformations. Bidirectional transformations are studied in different disciplines of computer science (*e.g.*, MDE, graph transformations and databases) to synchronize two data structures (a *source* and a *view*) [22,23]. In our case, the *source* is the model typed by MT_B found in a context where a model typed by MT_A (our *view*) is expected.

4.3 Total Model Subtyping

When a model of type MT_B can be used in every context in which a model of type MT_A is expected, we talk about *total* substitutability. Therefore, a subtyping relation which guarantees *total* substitutability is a *total* subtyping relation.

Definition 7. (Total subtype) *MT_B is a total subtype of MT_A if any model typed by MT_B can be safely used everywhere a model typed by MT_A is expected.*

4.4 Partial Model Subtyping

Conversely, a *partial* subtyping relation enable a model typed by MT_B to be used in a given context (*e.g.*, a given model transformation) where a model typed by MT_A is expected. This notion of *usage context* have been introduced by Kühne in order to define in which cases a specialization relation holds, while it does not hold in the general case [16]. Typically, a *partial* subtyping relation enables a model typed by MT_B to be substituted for a model a of type MT_A in the context of the call `m(a)` if MT_B contains the required features for m, even if MT_B is not a *total* subtype of MT_A.

Definition 8. (Partial subtype) *MT_B is a partial subtype to MT_A wrt. f if models typed by MT_B can be safely used where a model typed by MT_A is expected to use the feature f.*

(a) Total subtyping (b) Partial non-isomorphic subtyping

Fig. 4. Two different scenarios of the reuse of DCE between ORCC IR and GeCoS IR

Here, f can be an attribute or an operation from the model or a model manipulation that takes the model as argument. MT_B is a partial subtype to MT_A wrt. f if MT_B is a total subtype of MT_f, where MT_f is a model type which contains only the necessary information to apply or access f safely and such that MT_A is a total subtype of MT_f. We call MT_f the *effective model type* of f.

Definition 9. (Effective model type) *The effective model type MT_f of a feature f extracted from a model type MT_A is the model type which contains all the required features to access or call f and such that $MT_A \sqsubseteq MT_f$.*

This effective model type can be processed using a function which analyzes the model type and extracts its required subset to access a given feature.

Definition 10. (Effective model type extraction) *The effective model type extraction function is a function $extractEffectiveMT(MT_A, f)$, with MT_A a model type and f a required feature belonging to MT_A, and such that $MT_f = extractEffectiveMT(MT_A, f)$ is the effective model type of f extracted from MT_A.*

One possible way to extract this required subset is to use an approach like the one proposed by Sen *et al.* [5]. They compute a metamodel (called the *effective metamodel*) from a larger metamodel using the *footprint* of a model manipulation, *i.e.*, the set of types and features touched by the manipulation. This footprint can be processed statically, by analyzing the code of the model manipulation or dynamically using a trace of the execution of the operation [24]. The dynamic footprint is more accurate because it contains only the types and features of the objects which have been touched by the operation, whereas the static footprint contains all the types and features which may be touched by the operation. However, the dynamic footprint is also costlier and cannot be used for static type checking (cf. Section 5.2).

4.5 Definition of Subtyping Relations for Model Types

From these two criteria (isomorphism of the structures and totality of the subtyping), we define four model subtyping relations to provide model substitutability. In the following MT_A and MT_B are model types and $ModelType$ is the set of all model types.

The first model subtyping relation is the *total isomorphic* subtyping relation, to which the three others refer. MT_B is a *total isomorphic* subtype of MT_A if it contains one

matching object type for every object type of MT_A, *i.e.*, if $MT_B \sqsubseteq MT_A$. For example, such a subtyping relation could hold between GeCoS IR and a model type extracted from GeCoS IR by selecting only the relevant concepts for Dead Code Elimination (DCE). In Figure 4(a), where the DCE arrow represent the DCE model manipulation defined on a dedicated model type, this case is represented by the generalization arrow between GeCoS IR and the DCE dedicated model type.

Definition 11. (Total isomorphic subtyping relation) *The* total isomorphic *subtyping relation is the matching relation, denoted* $MT_B \sqsubseteq MT_A$.

A *partial isomorphic* subtyping relation wrt. feature f holds between MT_B and MT_A if MT_B contains matching object types for every object type belonging to the *effective model type* of f extracted from MT_A, *i.e.*, MT_B is *partial isomorphic* subtype of MT_A wrt. feature f if MT_B is a *total isomorphic* subtype of the *effective model* type of f extracted from MT_A.

Definition 12. (Partial isomorphic subtyping relation) *The* partial isomorphic *subtyping relation wrt. the feature* f *is a binary relation* $\sqsubseteq_f$ *on* $ModelType$ *such that* $(MT_B, MT_A) \in <:_f$ *(also written* $MT_B \sqsubseteq_f MT_A$*) iff* $MT' \sqsubseteq extractEffectiveMT(MT_A, f)$.

MT_B is a *total non-isomorphic* subtype of MT_A if there is a adaptation able to adapt every model typed by MT_B in a model typed by a *total isomorphic* subtype of MT_A. This adaptation must be bidirectional, or it would be impossible to reuse *endogenous* model manipulations from MT_A and the subtyping relation would not be *total*. Figure 4(a) represents such a subtyping relation between ORCC IR and the model type dedicated to DCE mentioned above. Loops from the latter are isomorphic to GeCoS IR ones, thus they cannot be isomorphic to loops from the former. Therefore an adaptation is needed, as the one described earlier (see 3.2).

Definition 13. (Total non-isomorphic subtyping relation) *The* total non-isomorphic *subtyping relation is a binary relation* $\sqsubseteq_{\sim}$ *on* $ModelType$ *such that* $(MT_B, MT_A) \in \sqsubseteq_{\sim}$ *(also written* $MT_B \sqsubseteq_{\sim} MT_A$*) iff* $\exists adapt_{MT_A}$ *a bidirectional adaptation from* MT_B *to* MT_C *such that* $MT_C \sqsubseteq MT_A$.

Finally, model type MT_B is a *partial non-isomorphic* subtype of MT_A wrt. the feature f if there is an adaptation able to adapt a model typed by MT_B in a model typed by a *total isomorphic* subtype of the effective model type of f extracted from MT_A. This adaptation must be bidirectional if f is an *endogenous* feature. Such a *partial non-isomorphic* subtyping relation is pictured in Figure 4(b), where ORCC IR is subtype of GeCoS IR through an adaptation to the *effective model type* of DCE extracted from GeCoS IR.

Definition 14. (Partial non-isomorphic subtyping relation) *The* partial non-isomorphic *subtyping relation wrt. the feature* f *is a binary relation* $\sqsubseteq_{\sim f}$ *on* $ModelType$ *such that* $(MT_B, MT_A) \in \sqsubseteq_{\sim f}$ *(also written* $MT_B \sqsubseteq_{\sim f} MT_A$*) iff* $\exists adapt_{MT_A}$ *an adaptation from* MT_A *to* MT_C *such that* $MT_C \sqsubseteq extractEffectiveMT(MT_A, f)$ *and* $adapt_{MT_A}$ *is a bidirectional adaptation if* f *is an* endogenous *model manipulation.*

5 Putting Subtyping Relations to Work

Defining model subtyping relations is not sufficient to build a type system. Indeed, a type system implements one or more subtyping relations and provides ways to declare and check them. Thus, we discuss here the ways to declare and check subtyping relations and the respective drawbacks and advantages of these approaches for an implementation of a model-oriented type system.

5.1 Declaration of Subtyping Relations

Subtyping relations can be declared in two ways: *explicitly* and *implicitly*. We call a subtyping relation declaration *explicit* when a syntactic construct is used to state the subtyping relation. Conversely, if the type system infers the subtyping relation from the information it can gather about the types or the use which is done from their instances, the declaration of the subtyping relation is *implicit*. In addition, the declaration of the subtyping relation can take place either *at the definition* of a type or *after the definition* of the subtype and the supertype involved in the subtyping relation.

The way to declare model subtyping relations may affect the possibilities that these relations offer through the type system. For example, a *non-isomorphic* model subtyping relation can be declared *implicitly*. To this end, a tool able to infer adaptations is necessary. Such inference can be done through patterns which are known to be safe or using ontologies to find corresponding class or feature names. However, an *implicit* adaptation mechanism will be more limited in terms of possible adaptations than an *explicit* one, which let the user define its adaptation based on its knowledge of the two model types involved. On the other hand, an *explicit* adaptation mechanism needs appropriate syntactic constructs and analyses to ensure that an adaptation is safe.

Declaration of a subtyping relation *at the definition* of a type is a kind of documentation, letting know what are the subtypes or supertypes of the defined type. Conversely, it is not always possible or desirable to add this information in a type, particularly if the subtyping relation is required for a very specific use (*e.g.*, a *partial* subtyping relation for a single model manipulation) or legacy code where existing model types should be modified. In such cases, declaring the subtyping relation *after the definition* of the involved types may be a solution.

Finally, declaration of a subtyping relation *explicitly at the definition* of a model type could allow inheritance. That is, reuse of the structure of the supertype, with the possibility to redefine or modify it in the subtype without breaking model subtyping. Moreover, if *explicit* declaration *at the definition* is the only way to declare model subtyping relations, it prevents from the type system to use subtyping relations which are unknown from the user, and thus prevents from accidental substitutability.

5.2 Checking of Subtyping Relations

Checking of the subtyping relations is the verification that a subtyping relation holds. Regardless of the way the subtyping relation is declared, this check can be processed either at design time, *i.e.*, during the compilation or interpretation process, or at runtime.

Here again, the way to check model subtyping relations can impact the facilities provided for model manipulations. On the one hand, design time (or static) check enables earlier detection (*i.e.*, than runtime check) of type errors and programming mistakes and thus earlier user feedback. It also enables tools to provide more facilities, such as type-based compiler optimizations, auto-completion or impact analyses. Moreover, compared to runtime checking, design time checking needs significantly fewer tests to achieve the same level of runtime safety.

On the other hand, runtime checking can be processed with more precise type information. When the program is running, the actual type of a variable is known rather than its declared type. Although possibly slower because of the process of the check during the execution of the program, dynamic checking enables valid programs which would be forbidden by a static type checker because of a lack of information. In the context of model types, knowing the actual model would enable the extraction of its model type and would possibly enable subtyping relations forbidden by a static type checker.

6 Discussions

Several approaches have been proposed in the last decade to provide engineering facilities for model manipulation reuse. We show in this section how the fthe different model subtyping mechanisms (*i.e.*, total/partial and isomorphic/non-isomorphic model subtyping, declaration and checking) defined in this paper can be used to classify these approaches through a unified theory. Figure 5 summarizes this classification. The question marks indicate the lack of information about the given mechanism.

6.1 Isomorphic vs. Non-isomorphic Subtyping Relations

To the best of our knowledge, the only approach using an isomorphic subtyping relation is the bidirectional subset of the *adaptation* DSL proposed by Babau *et al.* [9,10]. All

	Total / partial	Iso / non-iso	At / after definition	Explicit / implicit	Checking	Legacy tool reuse
Varró *et al.* [1]	Total	Non-iso (Class renaming)	After	Implicit	?	No
Cuccurru *et al.* [2]	Total	Non-iso (Abstract class renaming)	After	Explicit (Genericity and explicit object subtyping)	?	Yes
Steel *et al.* [3]	Total	Non-iso (Class renaming, multiplicities contraction)	After	Implicit	At compile-time, with possible runtime type errors	Yes
Sanchez Cuadrado *et al.* [4]	Total	Non-iso (Class renaming, multiplicities contraction)	After	Explicit (Binding DSL)	?	Yes
Sen *et al.* [5]	Partial (Effective metamodel)	Non-iso (Potentially any adaptation)	After	Explicit (Static introduction)	At compile-time, with possible runtime type errors	Yes
De Lara *et al.* [6,7,8]	Total	Non-iso (Class renaming, navigation and filtering of properties, $n-to-1$ bindings)	After (Binding), At definition (Specialization)	Explicit (Binding DSL)	?	No
Babau *et al.* [9,10]	Total (Bidirectional subset)	Iso (Bidirectional subset)	After	Explicit (Adaptation DSL)	?	Yes

Fig. 5. Classification of different model manipulation reuse approaches

other approaches either let class names vary or go further, enabling adaptations such as $n-to-1$ concepts *binding* or navigation and filtering of features. The latter use different mechanisms to *bind* the subtype to its supertype and express the adaptation (*e.g., adaptation* and *binding* DSLs or static introduction). The rarity of *isomorphic* subtyping relations can be explained by the restrictions such relations impose, restrictions which can be safely relaxed in some cases, *e.g.*, class names modification.

6.2 Total vs. Partial Subtyping Relations

Excepting one approach which allows the extraction of the *effective metamodel* from a model manipulation [5], all existing approaches are *total*. To be *total* a *non-isomorphic* subtyping relation must handle bidirectional adaptation. Bidirectionality is tackled in existing approaches by almost *isomorphic* relations [1,2,3,4,9] or by generating an adapted model manipulation rather than adapting the model [6,7,8].

6.3 Declaration of Subtyping Relations

All the existing approaches declare the subtyping relation or *binding after the definition* of the two model types (or their equivalent). However, de Lara *et al.* authorize specialization of model types (called *concepts* in their terminology) using a mechanism close to inheritance (*i.e.*, *at definition*) [6]. Only two approaches declare subtyping relations *implicitly* [1,3] whereas the others use *explicit* mechanisms mainly through DSLs [4,6,7,8,10,9], with the exception of the approaches from Cuccuru *et al.* [2] and Sen *et al.* [5] which use respectively genericity and static introduction.

6.4 Checking of Subtyping Relations

Little is said about the checking of the subtyping relations, apart from the work of Steel *et al.* [3], in which subtyping relations are checked *at compile time*. De Lara *et al.* [6] mention a notion of *valid* binding, but do not formalize it.

6.5 Legacy Tools Reuse

One group of our examples, abstract interpretation analyses, are implemented in an existing tool (P-Interproc). Among the existing approaches, some need to specifically define the model manipulation to be reused [1], or to process it in order to generate an adapted model manipulation [6,7,8]. By doing so, they prevent from reusing existing model manipulations which have not been defined using their own mechanisms or which sources are not available. The other approaches, which enable a subtyping relation with a legacy tool, are the ones with the fewest possible adaptations [2,3,4,10,9], or without any guarantee on the bidirectionality of such adaptations [5].

7 Conclusion and Perspective

This paper provides a review of the overall scope of model substitutability through model typing. To this end we analyze the subtyping relation between two model types wrt. both their structure, and the context of the need for such a substitutability.

First, to be substitutable a model must be structurally equivalent to the expected one.Such a structural equivalence can be achieved if the structures of the two model types are *isomorphic*, or thanks to an adaptation making the two structures *isomorphic*.

Second, such an isomorphism can be *total*, *i.e.*, achieved between the whole structures of the two model types, allowing a substitution of the corresponding models in every possible context where the substitution is necessary. Otherwise, *partial* model substitutability can be achieved according to a given context. In other words, the isomorphism can be achieved between a model type and the *effective model type* of the feature to be reused, *i.e.*, the subset of the model type used in a given context.

From these distinctions, we define four model subtyping relations providing different kinds of model substitutability: *total isomorphic*, *partial isomorphic*, *total non-isomorphic* and *partial non-isomorphic*. We review existing approaches to model manipulation reuse wrt. these subtyping relations. It appears that few existing approaches use *partial* subtyping relations or enable adaptations to handle complex structural heterogeneities between model types. Moreover some approaches forbid the reuse of legacy tools. More importantly, most of the approaches lack of a way to forbid erroneous reuse.

In addition to the comparison of existing approaches, the subtyping relations introduced in this paper provide a specification of a family of model type systems that can be implanted in a MDE CASE tool to enable *safe reuse* of model manipulations. In this context, we thus discuss ways to declare a subtyping relation, and how to check it. Depending on the chosen subtyping relation, as well as the way to declare and check it, these type systems enable many facilities that are well known at the programming language level, such as type-based compiler optimizations and auto-completion.

As a direct perspective of this work, we plan to refactor the existing model typing in Kermeta to support the subtyping relations identified in this paper. To this end, we plan to integrate the state-of-the-art of the existing approaches and to study how the work of Vignaga *et al.* [25], focusing on the typing of the relations between models as functions, can be combined with our subtyping relations.

Acknowledgement. This work has been partially supported by VaryMDE, a collaboration between Inria and Thales Research and Technology, and by the French ANR BioWIC (ANR-08-SEGI-005). The authors thank the anonymous reviewers for their constructive feedback which helped us to considerably improve the article.

References

1. Varró, D., Pataricza, A.: Generic and Meta-transformations for Model Transformation Engineering. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) UML 2004. LNCS, vol. 3273, pp. 290–304. Springer, Heidelberg (2004)
2. Cuccuru, A., Mraidha, C., Terrier, F., Gérard, S.: Templatable Metamodels for Semantic Variation Points. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA. LNCS, vol. 4530, pp. 68–82. Springer, Heidelberg (2007)
3. Steel, J., Jézéquel, J.M.: On model typing. SoSyM 6(4) (2007)
4. Sánchez Cuadrado, J., García Molina, J.: Approaches for Model Transformation Reuse: Factorization and Composition. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 168–182. Springer, Heidelberg (2008)

5. Sen, S., Moha, N., Mahé, V., Barais, O., Baudry, B., Jézéquel, J.-M.: Reusable model transformations. SoSyM 11(1) (2010)
6. de Lara, J., Guerra, E.: From types to type requirements: genericity for model-driven engineering. SoSyM (2011)
7. Sánchez Cuadrado, J., Guerra, E., de Lara, J.: Generic Model Transformations: *Write Once, Reuse Everywhere*. In: Cabot, J., Visser, E. (eds.) ICMT 2011. LNCS, vol. 6707, pp. 62–77. Springer, Heidelberg (2011)
8. Wimmer, M., Kusel, A., Retschitzegger, W., Schönböck, J., Schwinger, W., Cuadrado, J., Guerra, E., de Lara, J.: Reusing model transformations across heterogeneous metamodels. In: International Workshop on Multi-Paradigm Modeling (2011)
9. Kerboeuf, M., Babau, J.-P.: A DSML for reversible transformations. In: OOPSLA Workshop on Domain-Specific Modeling (2011)
10. Babau, J.-P., Kerboeuf, M.: Domain Specific Language Modeling Facilities. In: MoDELS Workshop on Models and Evolution (2011)
11. Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools, 2nd edn. Addison-Wesley (2006)
12. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
13. OMG: Meta Object Facility (MOF) 2.0 Core Specification (2006)
14. OMG: UML Object Constraint Language (OCL) 2.0 Specification (2003)
15. Ernst, E.: Family Polymorphism. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 303–326. Springer, Heidelberg (2001)
16. Kühne, T.: On model compatibility with referees and contexts. SoSyM (2012)
17. Bruce, K.B., Schuett, A., van Gent, R., Fiech, A.: Polytoil: A type-safe polymorphic object-oriented language. ACM TOPLAS 25(2) (2003)
18. Steel, J.: Typage de modèles. PhD thesis, Université de Rennes 1 (April 2007)
19. Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schoenboeck, J., Schwinger, W.: From the Heterogeneity Jungle to Systematic Benchmarking. In: Dingel, J., Solberg, A. (eds.) MODELS 2010. LNCS, vol. 6627, pp. 150–164. Springer, Heidelberg (2011)
20. Mens, T., Gorp, P.V.: A taxonomy of model transformation. Electronic Notes in Theoretical Computer Science 152 (2006)
21. Foster, J.N., Greenwald, M.B., Moore, J.T., Pierce, B.C., Schmitt, A.: Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. ACM TOPLAS 29(3) (2007)
22. Czarnecki, K., Foster, J.N., Hu, Z., Lämmel, R., Schürr, A., Terwilliger, J.F.: Bidirectional Transformations: A Cross-Discipline Perspective. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 260–283. Springer, Heidelberg (2009)
23. Hu, Z., Schürr, A., Stevens, P., Terwilliger, J.F.: Bidirectional transformation ”bx” (dagstuhl seminar 11031). Dagstuhl Reports 1(1) (2011)
24. Jeanneret, C., Glinz, M., Baudry, B.: Estimating footprints of model operations. In: ICSE (2011)
25. Vignaga, A., Jouault, F., Bastarrica, M., Bruneliére, H.: Typing artifacts in megamodeling. SoSyM (2011)

BOB the Builder: A Fast and Friendly Model-to-PetriNet Transformer

Ulrich Winkler[1], Mathias Fritzsche[2], Wasif Gilani[1], and Alan Marshall[3]

[1] SAP Research, SAP AG,
The Concourse, Belfast, United Kingdom
{ulrich.winkler,wasif.gilani}@sap.com
[2] SAP Modelling and Taxonomy, SAP AG Walldorf, Germany
mathias.fritzsche@sap.com
[3] Queen's Universiy Belfast
Belfast, United Kingdom
a.marshall@ee.qub.ac.uk

Abstract. Petri-Nets are a very expressive modelling concept. However, modelling industrial problems using Petri-Nets is not a trivial task as Petri-Nets do not provide support for constructing large models. Modelling a complete business process, for example, with several activities and associated resources using Petri-Nets becomes a complex task. Model transformations are a promising technology to address this problem. In this paper we present an extended Petri-Net model that supports modelling industrial problems via model transformations. We also introduce a transformation framework that allows to graphically define model transformations by templates.

1 Introduction

Even the simplest form of Petri-Nets (PN), ordinary place/transition nets, are a very expressive modelling concept. Extension, such as inhibitor arcs, time or priorities make Petri-Nets Turing complete.

However, the uptake of Petri-Net based simulations in industry is low as modelling of industrial sized problems using Petri-Nets is not a trivial task. Petri-Nets do not provide support for structuring large models. To model an enterprise scale business process, for example, with several activities and associated resources using Petri-Nets becomes a cumbersome task.

We address this problem by utilising model-driven technologies, such as model-to-model transformations and model tracing. Model transformation allows us to transform an arbitrary model, i.e., a BPMN model, into a Petri-Net model. The Petri-Net model is then used to conduct various analysis, such as simulations and model checking. The results are then transformed back and visualised in the context of the original source model (e.g. the BPMN model). We explain our transformation approach briefly in Section 2.

In our previous work we used ATL [5] to implement model transformations. Although ATL is a very powerful and generic transformation language, we found

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 416–427, 2012.

that developing transformation scripts is time consuming, cumbersome and error prone. Moreover, script based transformation toolkits, such as ATL, tend to be slow.

Our approach uses (a) template based method to define a transformation, (b) graphical editors to model these templates and (c) code generation to transform these templates into the transformation code. Moreover, our framework is deeply integrated into the Eclipse IDE [11] to provide streamlined design, coding, testing, and debugging of transformations.

Our framework comprises:

- BEAM - The Behaviour Analysis Model. BEAM is a modular Petri-Net model with extensions which are useful for model transformations and model tracing. We briefly discuses BEAM in Section 3.
- BOB - The BEAM Orchestration and Builder Framework. BOB is a transformation framework for modularised Petri-Nets. Part of BOB are graphical tools to define transformations by providing templates. Using these user provided templates BOB generates the transformer code. We describe BOB in Section 4.

We discuss related work in Section 5 and conclude this paper with an outlook in Section 6.

2 BOB Transformations at a Glance

Before going into details, let us briefly introduce the basic idea of the transformation procedure with modularised Petri-Nets.

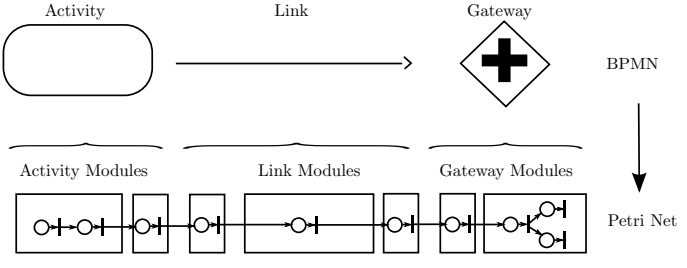


Fig. 1. BOB transformation: the business process model and its transformation into a BEAM instance

In Figure 1 you see a business process modelled in BPMN like notation and its transformation into a Petri-Net. The process comprises three model elements: an *activity*, a *gateway* and an *FlowLink*. The FlowLink connects the activity and the gateway. In bottom half of Figure 1 you see a Petri-Net model which models the behaviour of this business process. Parts of the Petri-Net are grouped by

rectangles, which we call modules. For example, the left side module describes the internal behaviour of a process activity, the right side module the functionality of the gateway. The *FlowLink* module describes how the *FlowLink* connects the *activity* and the *gateway*.

The basic idea behind BOB is, that at design time the simulation developer creates templates of the basic Petri-Net structure for all meta-classes of the source meta-model, in this case the BPMN meta-model. We call these template Petri-Net modules *class modules*. If a meta-class contains a reference to other meta-classes (or to itself) the simulation developer also provides templates how class modules should be connected by *reference modules*. These modules are compiled by the BEAM compiler into Java code. The simulation developer may refine this code as required.

At run-time, the BOB transformation process takes a BPMN process model, the provided transformation code and produces a BEAM Net in three steps:

1. A new, empty BEAM Petri-Net is created.
2. The BOB transformer iterates over all elements in the BPMN model. For each source model element the transformation determines the meta-model class of that element and selects the right BEAM module. If a module is available, the transformer creates an instance of that module and inserts that instance in the BEAM model. We call that step the *class module generation*.
3. After all class patterns have been created, the transformer *fuses* the patterns together; that is the transformer creates instances of *reference modules* and connect places and transitions between different class modules and reference modules.

3 BEAM Behaviour Analysis Model

BEAM is a generalised stochastic Petri-Net which supports read, reset and inhibitor FlowLinks. Places and transitions of a BEAM net are grouped in modules. There is nothing special about BEAM; from a pure Petri-Net point of view BEAM is an ordinary Petri-Net as many others, except that we build the support for transformation into the BEAM meta-model. The general idea how we transform arbitrary ECORE based models into a Petri-Net can be applied to other Petri-Net specification as well.

The BEAM meta-model is shown in Figure 2.

BEAM Module: A BEAM module is the basic building block in the BOB transformation framework. A BEAM Module (which is contained in a BEAM file), groups *nodes* (places and transitions) and *arcs* of the BEAM Petri-Net.

Each module (and node) has two identifiers; the *module identifier* (MID) and the *universal unique identifier* (UUID). The UUID, as the name implies, differs for each element in a BEAM-Net. MIDs are only unique for elements within a module and might be the same in two or more modules. Two different modules with the same MID are treated as the same type of module. Module identifiers are used by the BEAM Synchroniser to synchronise modules across files.

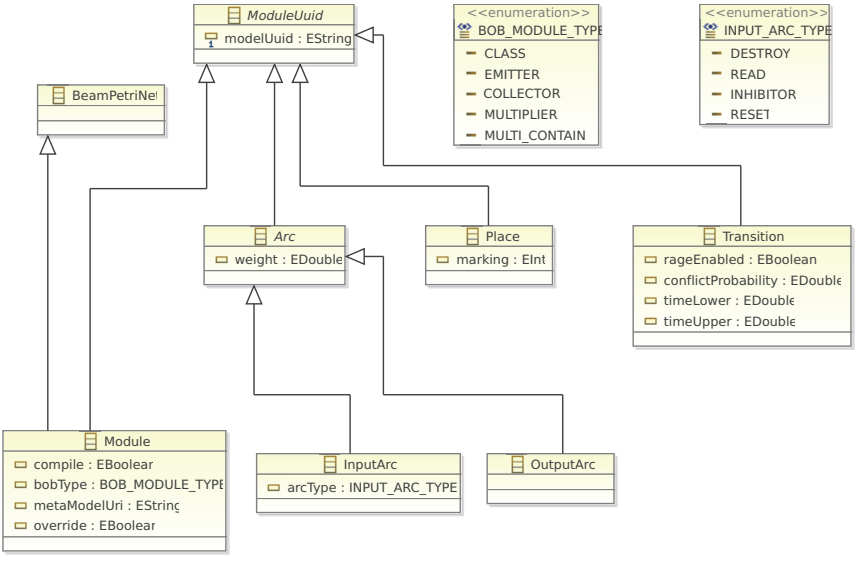


Fig. 2. BEAM meta-model

Each module has a `BOB_MODULE_TYPE`. The `BOB_MODULE_TYPE` is used by the transformer to determine how and when to create and fuse modules. We elaborate on these different module types in Section 4.4 and 4.3.

Furthermore we distinguish between *template modules* and *instance modules*. However, this classification is pure linguistics and not expressed in the BEAM meta-model. A template module is provided by the developer and serves as a basic pattern for the BOB transformer to transform a source model object into its Petri-Net representation. We call that process *instantiation*. Once the BOB transformer creates an instance of a *template module* we call it an *instance module*.

Modules may contain other modules and thus may form a hierarchy. This feature is used by the transformation framework to reflect the internal structure of the source model.

4 BOB - The BEAM Orchestration and Builder Framework

Throughout this paper we use a very simple ECORE based meta-model as an accompanying example. The UML diagram of the meta model is given in Figure 3. This example meta-model comprises three classes A, B and X. Class X extends class B. Class A has a zero-to-many reference to B.

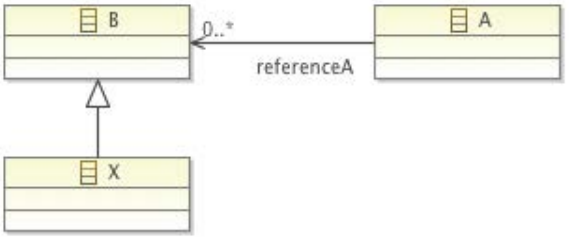


Fig. 3. The meta-model `simple.Example`

4.1 BOB Library Project and BEAM Packages

Let us assume that a simulation developer wants to create an Example-Model-to-BEAM transformation. The simulation developer would start his modelling activities by creating a new BOB library project using the BOB project generator. The BOB project generator loads the source meta-model and initialises the *BOB library project*.

BOB Library Project. A BOB library project is an Eclipse Java project with additional BOB build natures, the BEAM compiler nature and the BEAM synchroniser nature. We elaborate on both natures in Section 4.2. The BOB project generator also initialised the basic project structure and generates *BEAM packages* for all classes of the meta-model.

BEAM package. A BEAM package is a set of *template BEAM modules* and auxiliary Java code that define how a source model object is supposed to be transformed into BEAM modules, orchestrated and composed by the BOB transformer during a transformation. In our example the BOB project generator would produce three BEAM packages for A, B and X. Let us assume that the full qualified name of our meta-model is `simple.Example`. In such a case BOB would create a package `beam.bob.simple.example.a` for class A, `beam.bob.simple.example.b` for B and `beam.bob.simple.example.x` for X.

Once the BOB project generator has initialised the basic project structure and all BEAM packages, the simulation developer starts to fill in the initial empty BEAM module templates and models the basic behaviour using a graphical editors.

4.2 BEAM Module Compiler and Synchroniser

Let us briefly explain what a build-nature does. Every Eclipse project has zero, one or more associated build-natures. Briefly explained, build-natures work like this: the Eclipse runtime-environment observers all files within a project. If a file has been created, changed or deleted all build-natures are notified. Upon a

notification the build-nature decides what to do with that received notification. The Java build-nature, for example, would compile a Java source code file into a Java class code file.

BEAM Module Compiler: The BEAM Module Compiler takes a BEAM module and generates the Java code that would produce that module. As the BEAM Module compiler is a part of the build-nature, the BEAM compiler is invoked whenever the simulation developer or the BEAM synchroniser changes a BEAM module definition.

BEAM Synchroniser: BEAM modules of the same type may be used in different BEAM files in a project. We wanted to free the simulation developer of the burden to keeping all BEAM modules in sync. Therefore we provide the BEAM synchroniser which detects changes made to a BEAM module and applies these changes to all BEAM modules in the same project.

4.3 Classes and Interfaces

The Eclipse Modelling Framework and its ECORE meta-model supports object oriented concepts like *classes*, *interfaces* and *inheritance*.

BEAM does not distinguish between interfaces and classes. The project generator produces a BEAM file with a single BEAM module for all classes and interfaces. In our example the project generator would create a single module in a BEAM file named `A.beam` in the package `bob.simple.example.a`

These modules are used to define the internal behaviour of a source model element, for example the internal behaviour of an activity or gateway.

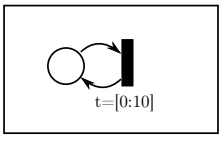


Fig. 4. Class A template

Figure 4 shows the template module of class A which we will use in our example.

4.4 References

In our business process example (Figure 1) the FlowLink object has two references, one reference `source` to the activity object and one reference `targets` to the gateway object. In this example the only objective of the FlowLink modules is to transport tokens from the activity module to the gateway module. However, in other use-case that might be different and the simulation developer has to module a more complex behaviour of a reference.

A reference may be regarded as a communication channel between objects. The token would be the message sent over that channel. The transformation of a reference into a BEAM Petri-Net is defined by three modules: the *emitter*, the *multiplier* and the *collector* module.

The emitter module is used to define which object should receive a message and in which order. The multiplier module is the channel itself and might be used to "delay" a message or to lose a message. The multiplier is used to coordinate how messages are "transmitted" or be used to delay a token on it's way to the receiver. The collector might be used to specify in which order a sender receives messages.

For every reference in a meta-model the BOB project generator generates templates for the developer to fill in and to change according to the required need. Let us explain how the developer would model the reference template. In our example class A has a reference to class B. The BOB project generator would create a BEAM file `A_referenceA_B.beam` in package `bob.simple.A`. This beam file would contain five modules: the template A, the emitter template, a multiplier template, the collector template and the class template B as shown in Figure 5. Let us assume that the developer filled these templates in as depicted in Figure 5 and 7.

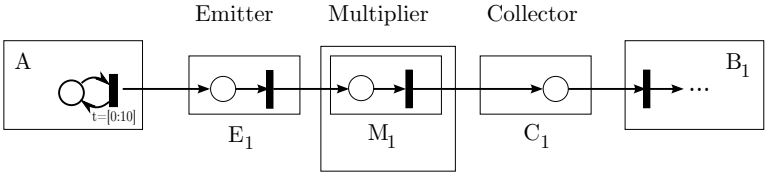


Fig. 5. Template modules for `referenceA`

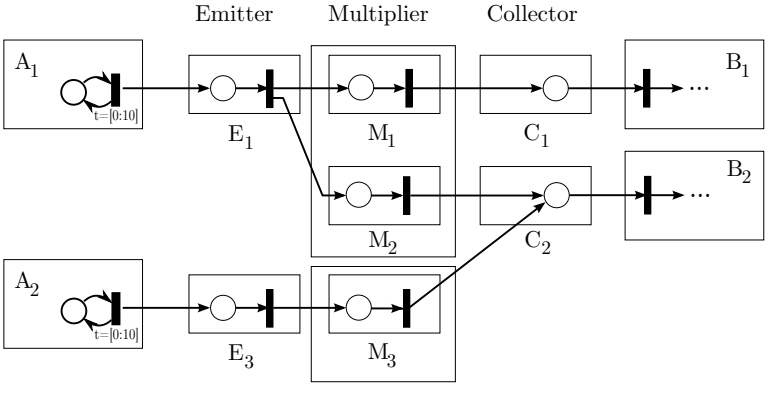


Fig. 6. Transformation result: instance modules (2:2 case)

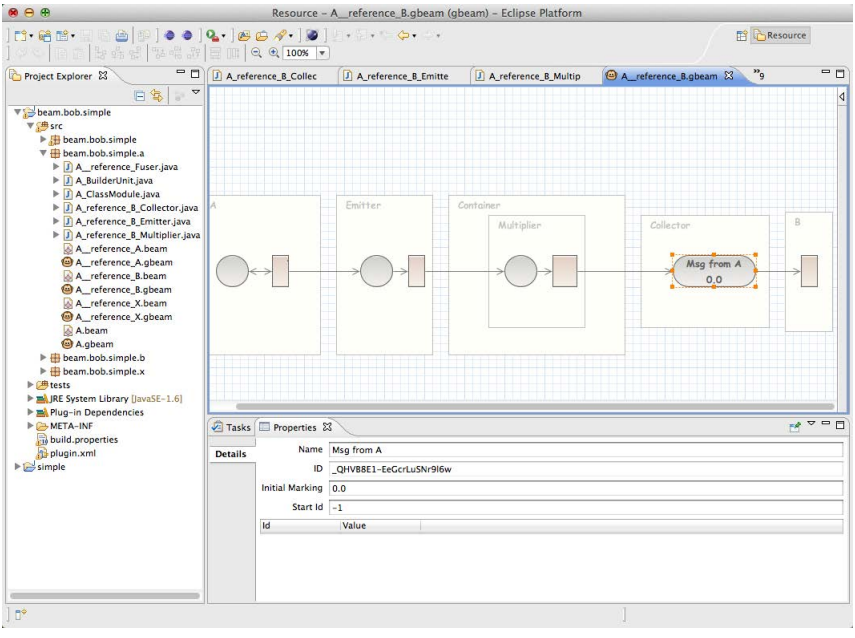


Fig. 7. BOB screen shot. This screen shot shows how the developer uses the graphical BEAM editor (the GBEAM editor) to model the A to B reference templates of the example meta-model. The *Project Explorer View* shows the BOB library project package layout, generated Java code and the BEAM and GBEAM files. BEAM files contain the BEAM model whereas GBEAM contain the graphical details, such as layout, positions and size of elements.

Let us assume that we have a model instance of our example meta-model with four objects; A_1, A_2, B_1 and B_2 as shown in Figure 6. A_1 holds two references to B_1 and B_2 whereas A_3 holds only one reference to B_2. The transformer generates object modules and reference modules as shown in Figure 6 according to the following rules:

1. For each object that holds a reference r the transformer generates an Emitter module. A_1 and A_2 hold a reference to B objects and therefore the transformer produces the Emitter modules E_1 and E_2.
2. For each reference the transformer produces a multiplier module. In our example we have three references and therefore three multiplier modules.
3. For each referenced object the transformer generates a collector pattern. As B_1 and B_2 are referenced objects we have two collectors in our examples.
4. Finally, the BOB transformer creates all arcs between emitter, multiplier and collector.

This schema of modelling references provides a very flexible way to define various scenarios. In most of our use-case we could use this approach to model references.

If it was required to model special behaviour we were able to changed the generated code of the emitter, multiplier or collector.

4.5 Containment References

In some cases we reduce the modelling effort. If, for example a reference is a containment reference then the contained object does not need a collector module, as the containing object is the only object that holds a reference to it. If the reference is a 1:1 reference we can omit the multiplier modules.

4.6 Inheritance

BOB supports inheritance in a very simplistic way. BEAM Modules have the boolean flag `override`. This flag is used to indicate whether the BEAM compiler should compile a template module. The `override` flag is also used by the synchroniser to check whether or not a module should be synchronised with its parent module.

In our simple example, model X inherits from B. The BOB project generator creates the same set of empty template modules as for B. If the simulation developer makes changes in the B module, these changes are applied to the X template module as well unless the `override` flag is set. If the developer change the X module, the `override` flag is set to true. If the `override` flag is set, the synchroniser will no longer synchronise B and X models and the compiler creates the necessary Java code.

If a meta-class has more than one super type this simple mechanism does not work anymore. In this case the BOB project generator disables the inheritance feature by setting the override flag to `true` and the simulation developer has to model this class.

4.7 Module Interfaces and Module Re-use

The BEAM synchroniser uses the BEAM module UMD to synchronise modules across files. This synchronisation mechanism is not limited to modules generated by BOB. The developer may also create modules and assign a unique Universal Module ID to it. This allows the developers to create "libraries" of modules. If they wants to "pull in" a module from a library all they has to do is to create an empty module and give it the same Universal Module ID as the module in the library. The synchroniser then replaces the empty module with the module in the library. With that mechanism we enable the developer to re-use modules and to create interfaces as well.

4.8 Tracing

Model tracing is essential, as tracing relates source model elements to specific places and transitions. Take for example a process activity in Figure 1. The Business Process Expert wants to learn about the queueing length for that particular

activity. The queue is symbolised by a place in the BEAM net; the queuing length would be the number of tokens in that place. Tracing establishes the link between process activity and the activity module instance.

Model tracing produces additional *tracing model* [8,3]. We found that using tracing techniques, as for example used by [2], is cumbersome and not needed for BEAM. We were able to build bi-directional tracing into BEAM using a simple, easy to use, and yet sufficient approach: if the source model element has an unique identifier that unique identifier is stored as an attribute by each BEAM module. If the source model element has no unique identifier, the source model URI and the traversal path to the source model element is stored instead.

To identify specific places or transitions, such as the place that symbolises the queue of an process activity, we are using MIDs as well to tag places, transitions and arcs. In our example, the queue place would have the MID `activity-queue`.

Hence, whenever the process expert selects a activity element in the process modelling environment we are able to identify the instance module for that activity and the queueing place of the instance module by utilising MIDs.

4.9 Module Unit Tests and Transformation Verification

In software engineering Unit testing is a method to determine if a small testable part of an application or library meets design requirements and is ready for use. The smallest testable part of a BOB transformation are modules and the BOB project generator creates and initialises for each class and reference a *Unit Test* case. Unit Tests are used by the developer to verify that each template module meets a set of initial design requirements and specifications. For example, a Unit Test for the gateway model may test if the gateway distributes tokens correctly.

BOB provides tools to simplify Unit testing, for example a BEAM simulator that can be executed step by step thus enabling the developer to observe and verify each state during a Unit-Test run.

5 Related Work

The problem of modelling industrial sized Petri-Net has been addressed by a large body of research. Basically four approaches exists: bottom-up/top-down refinements [13,7,1], higher-level Petri-Nets [4], equivalent models [12,6] and Model-to-Petri-Net transformations [10]. The first three approaches have in common that the end-user is required to manually construct Petri-Nets. However, as pointed by Lohmann et al. non-academic people - for example a business process expert - prefer easy-to-use modelling environments tailored to their specific needs [6]. Refinement strategies and Higher-Level Petri-Net are very academic solutions. Equivalent models, such as YAWL [12] for analysing business processes, aim to provide a modelling environment that is less academic, acceptable for a business process experts in terms of usability, and yet is based on strong Petri-Net semantics. However, most commercial products uses BPMN [9] as BPMN is preferred by business process experts as it offers a larger number of modelling elements which YAWL is missing, such as gateway or complex control-flow constructs.

The Petri-Net Kernel (ePNK) is a tool based on Eclipse. It provides graphical editors and a plug-in mechanism to connect various types of Petri-Nets. The ePNK is developed in model driven way, just like BEAM, using the Eclipse Modelling Framework and ECORE, and thus existing model-transformation frameworks as ATL [5] can be used to transform any kind of models to ePNK models. However ATL is a generic transformation language, but we aim to provide a domain specific transformation environment which is tailored to BEAM.

6 Conclusion and Future Work

In this paper we presented BEAM and BOB. BEAM, the Behaviour Analysis Model, is a modular Petri-Net model with extensions which are useful for model transformations and model tracing. BOB, the BEAM Orchestration and Builder Framework an Eclipse based toolkit that comprises a project generator, a compiler, a synchroniser, and a graphical editor to define transformations by providing templates. Using these user templates BOB generates the transformer code.

We found that BEAM/BOB simplify the way that developer model and implement a model-to-PetriNet transformation. In our previous work we used either ATL [5] based scripts or pure Java functions to transform a source model into a BEAM net. Writing code was cumbersome, time consuming and error-prone. Using BOB we are able to develop a transformation much faster and in a user-friendly way using graphical editors to model and test the transformation.

BOB provides a systematic modelling approach to simulation developers. As the BOB project generator generates a complete set of templates the developer has to fill in. The developers can go through these set of templates one by one and model the transformation. If they finished this activity they can be assured that they modelled a transformation completely.

BOB transformation is compiled code rather than an interpreted script. Compared to other script-based transformation, such as ATL [5], a BOB transformation is very fast. Initial tests on 2.4 GHz Intel Core 2 Duo processor with 8 GB RAM demonstrated that a BOB transformation can be 4000 times faster than a ATL based transformation (we compared a BOB transformation with our previous ATL based work [2]). The main reason is that a BOB transformation is compiled code whereas ATL is a script-based transformation. ATL needs to load and parse the transformation scripts, loads the XML models and so on. These steps are not required by BOB and therefore BOB is much faster. Moreover, BOB executes all transformation solely in memory to avoid any disk IO. Comprehensive performance tests of BOB are considered as future work.

References

1. Fehling, R.: A Concept of Hierarchical Petri Nets with Building Blocks. In: Rozenberg, G. (ed.) APN 1993. LNCS, vol. 674, pp. 148–168. Springer, Heidelberg (1993)
2. Fritzsche, M.: Performance related Decision Support for Process Modelling. Phd, Queen's University Belfast (2010)

3. Fritzsche, M., Johannes, J., Zschaler, S., Zherebtsov, A., Terekhov, A.: Application of Tracing Techniques in Model-Driven Performance Engineering. In: Proceedings of the 4th ECMDA Traceability Workshop (ECMDA-TW), pp. 111–120 (2008)
4. Jensen, K.: Coloured petri nets (1987)
5. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72, 31–39 (2008)
6. Lohmann, N., Verbeek, E., Dijkman, R.: Petri Net Transformations for Business Processes – A Survey. In: Jensen, K., van der Aalst, W.M.P. (eds.) ToPNoC II. LNCS, vol. 5460, pp. 46–63. Springer, Heidelberg (2009)
7. Mu, D.J., DiCesare, F.: A Review of Synthesis Techniques for Petri Nets. In: Proceedings of Rensselaer's Second International Conference on Computer Integrated Manufacturing, Troy, NY, USA, pp. 348–355. IEEE Comput. Soc. Press, Los Alamitos (1990)
8. Object Management Group: MOF QVT Final Adopted Specification (2005)
9. Object Management Group: Business Process Modeling Notation Specification, Final Adopted Specification, Version 1.0 (2006)
10. Raedts, I.G.J., Petkovi, M., Usenko, Y.S., van der Werf, J., Somers, J.F.G.L.: Transformation of BPMN models for behaviour analysis. In: Proceedings 5th International Workshop on Modelling, Simulation, Verification and Validation of Enterprise Information Systems, MSVVEIS 2007, Funchal, Madeira, Portugal, June 12-13, pp. 126–137. INSTICC Press (2007)
11. und Dan Rubel, E.C.: Eclipse. Building Commercial-Quality Plug-Ins. Addison-Wesley (2006)
12. Vanderaalst, W., Terhofstede, A.: YAWL: yet another workflow language. Information Systems (4), 245–275 (2005)
13. Zuberek, W.M.: Petri nets in hierarchical modeling of manufacturing systems. In: Proc. IFAC Conf. on Control Systems Design (CSD 2000), Bratislava, Slovak Republic, June 18-20, pp. 287–292 (2000)

Solving Acquisition Problems Using Model-Driven Engineering

Frank R. Burton[1,2], Richard F. Paige[2], Louis M. Rose[2], Dimitrios S. Kolovos[2], Simon Poulding[2], and Simon Smith[1]

[1] MooD International, York Science Park, YO10 5ZF, UK
{frank.burton,simon.smith}@moodinternational.com
[2] Department of Computer Science, University of York, YO10 5GH, UK
{paige,louis,dkolovos,smp}@cs.york.ac.uk

Abstract. An acquisition problem involves the identification, procurement and management of resources that allow an organisation to achieve goals. Examples include through-life capability management (in the defense domain), and planning for the *next release* of a software system. The latter is representative of the challenges of acquisition, as solving the problem involves the assessment of the very many ways in which the different requirements of multiple heterogeneous customers may be satisfied. We present a novel approach to modelling acquisition problems, based on the use of Model-Driven Engineering principles and practices. The approach includes domain-specific modelling languages for acquisition problems, and uses model transformation to automatically generate potential solutions to the acquisition problem. We outline a prototype tool, built using the Epsilon model management framework. We illustrate the approach and tool on an example of the next release acquisition problem.

1 Introduction

An *acquisition problem* involves the identification, procurement and management of resources to achieve goals. Consider the following scenario.

> The National Lifeboats Institution is considering its next-generation capability. It has a limited budget, and must satisfy numerous stakeholders such as lifeboat operators, funding bodies, charity workers, and the general public. Its goal is to provide an efficient and effective lifeboat service that protects the public and saves lives. It may acquire improved capability through: better training of current operators, better equipment (e.g., newer lifeboats), better education, etc. It wants to find the most cost-effective capability solution.

This scenario is representative of acquisition problems in numerous domains, including defense, aerospace, logistics, software project management and law. Such problems are complex, and are known to be hard [1,2]. They are challenging for a number of reasons:

- they often involve multiple objectives (e.g., saving lives, minimising cost);
- they are heterogeneous, involving tradeoffs between different types of entities (e.g., better training versus better equipment);

A. Vallecillo et al. (Eds.): ECMFA 2012, LNCS 7349, pp. 428–443, 2012.

- they are often dynamic: different solutions may be optimal or near-optimal at different times.
- there is rarely a single optimal solution.

Not only is determining optimal or near-optimal solutions to acquisition problems difficult, even understanding the acquisition problem may be challenging, in part because of the uncertainty and imprecision in the problem definition.

We contribute a general approach, based on Model-Driven Engineering (MDE) concepts and technologies, for modelling acquisition problems and calculating solutions that are optimal (Pareto optimality as defined in Section 2.1). The approach consists of a set of domain-specific languages (DSLs) for modelling acquisition goals and scenarios; these scenarios are then manipulated, by a chain of model transformations and model management operations, to produce solutions that are optimal. The advantages of using DSLs and MDE concepts and technologies for modelling and solving acquisition problems are multi-fold:

- Existing techniques used for solving acquisition problems are predominantly domain *dependent*: they rely on domain-specific models and algorithms for expressing how a capability matches a problem. MDE technologies and concepts offer a general, domain *independent* approach to solving such problem while still enabling a domain specific method of expressing the problem itself.
- In particular, the representation of problem concepts, goals and constraints using a DSL-based approach may be more accessible by domain experts.
- Acquisition problems conceptually reduce to manipulating graphs, where nodes typically represent goals and solutions, and edges represent dependencies. MDE technologies allow us to model and attempt to solve acquisition problems in their full generality.
- MDE principles and technologies allow us to abstract away from the complexity of calculating optimal solutions, and to modularise the calculation process. Model transformation, in particular, simplifies mapping (domain) models of problems to optimal solutions.

To illustrate the modelling approach, its novelties and limitations, we apply it to a representative acquisition problem: the software engineering *Next Release Problem (NRP)*. The acquisition objective is to find the 'best' customer requirements to satisfy in a new software release, while staying within budget. This problem, defined in detail in Section 2, is known to be NP-hard [3]. We use it to demonstrate that the MDE-approach can successfully model such complex problems, calculate optimal solutions, and also handle a more general version of the problem that includes dependencies between requirements. We also demonstrate that we can handle continuous variable requirements [2] and provide feedback and explanations of the results, via visualisations.

Beyond offering a solution method for NRP, the presented approach makes multiple generic contributions: the automatic generation of goal models from a partial goal model decomposition with high level descriptions of possible acquirable systems; the automatic evaluation of the generated goal models; calculation of multiple solutions that satisfy multiple stakeholder objectives and that are Pareto optimal, allowing decision-makers to make more informed acquisition; and solution visualisation via MDE tools, particularly EuGENia [4] and GMF [5].

The paper is structured as follows. In Section 2 we present related work and introduce the foundations of NRP, as well as the MDE techniques and technologies we use for our solution. In Section 3 we present our modelling approach, focusing on the DSLs to be used by engineers in representing acquisition goals and scenarios; we also outline the transformations used to produce optimal outputs, and explain how solutions are judged to be optimal. In Section 4 we apply the approach to an instance of NRP. We conclude in Section 5.

2 Background and Related Work

In this section we review the key previous work on acquisition problems, focusing on the Next Release Problem as a representative example. Acquisition problems are, formally, multi-objective optimisation problems. As described in this section we also discuss related work on NRP in more detail, then review the MDE technology that underpins our approach and tools.

2.1 Multi-objective Optimisation Problems

NRP, like other acquisition problems (e.g., Through-Life Capability Management [1]), is an example of a multi-objective optimisation problem. In contrast to single-objective problems, when multiple competing objectives exist, there is normally not a single 'best' solution. Instead, a solution may be the better at one objective but worse than other solutions at a different objective. A solution, X, is said to be *dominated* by another solution, Y, if Y is strictly better than X on at least one objective and as good as X on the remaining objectives. This leads to the notion of a *Pareto front* consisting of those solutions that are not dominated by any other solution. In other words, a solution on the Pareto front may only be bettered on a particular objective if we are willing to accept a worse score on another objective. These solutions on the Pareto front are considered to have *Pareto optimality*. (Deb in [6] presents a detailed explanation of multi-objective optimisation and the concept of a Pareto front.)

When applying search techniques to multi-objective problems it is necessary to quantify how 'good' all solutions are in terms of Pareto optimality, whether or not they are on the Pareto front. An appropriate method is to assign solutions on the Pareto front a *non-domination rank* of 0, then to temporarily ignore the solutions on this front in order to find a new Pareto front which is assigned rank 1, repeating this process until all solutions are ranked [7].

An example of a Pareto front for two objectives, together with solutions having non-domination ranks of 1 and 2, is shown in Fig. 1. For each solution on the Pareto front, it can be seen that no other solution dominates it, but no one solution is optimal for both objectives. Solutions on the Pareto front are the best representatives of different possible trade-offs beyond the competing objectives, and so when making decisions about a multi-objective optimisation problem, typically only the solutions on the Pareto front need be considered.

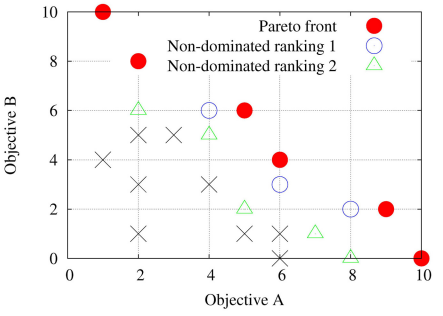


Fig. 1. Example of a Pareto front with the first two non-domination rankings marked. For each objective, a larger value represents a better solution.

2.2 The Next Release Problem

The Next Release Problem involves finding the 'best' set of customer requirements that will be satisfied in the next software release while staying within budget. In the NRP there are multiple customers with numerical weights to indicate their importance to the software company. Each customer requests one or more requirements and each requirement has an associated cost; requirements may also have causal dependencies. The original objective of the NRP was to select which customers should have their requirements satisfied. A consequence of framing the NRP with this objective is that solutions can be unbalanced: a small proportion of customers might have all of their requirements satisfied, and the requirements of some customers might be ignored. The approach to NRP presented by Bagnall et al [3] is limited in this manner, as it is very likely for some customers to have no requirements satisfied at all.

Later work [8–11] overcomes this limitation by having customers place weights of importance on each requirement and hence modifying the NRP so that requirements, instead of customers, are selected for inclusion. The objective in the modified NRP is to select which requirements to satisfy, in order to maximize the total customer satisfaction subject to the provided numerical weights. A further variant on the problem [9] considers the decomposition of a system, where a system is made up of components (typically software features) that work together, and focuses on selecting which new components will be added in the next release.

2.3 The Multi-objective NRP

In the multi-objective Next Release Problem (MONRP) [10], the cost becomes an objective rather than a constraint. The formulation of the problem in terms of customers and requirements is equivalent to variants of NRP described above that consider weights on each requirement indicating the importance of that requirement to the customer. There are now two competing objectives: to maximise the total customer satisfaction, and to minimise the cost to the company (the sum of all the costs of the selected requirements). The solution of the multi-objective NRP problem is therefore a Pareto

front of points, each representing a different combination of requirements to include in the release.

Zhang [10] presents an approach to solving MONRP by calculating maximum customer satisfaction for all possible available resources, and hence demonstrates the effects of varying the budget on customer satisfaction. Zhang shows that it is possible to determine situations where increasing the budget only slightly can have a large effect on customer satisfaction, or that the budget can be significantly reduced with only minimal effect on customer satisfaction. However, this approach does not consider dependencies between requirements. In follow-up work, Zhang [2] identifies multiple challenges for MONRP, two of which are: providing sensible feedback and explanation of results; and handling continuous variable requirements (i.e. requirements that can be partially fulfilled, such as increasing the responsiveness of a system). Besides handling requirements dependencies, the modelling approach and tool we present in this paper addresses the latter completely, and the former partially (via visualisations). MDE technologies enable both of these contributions, as we discuss in more detail in the following section.

Various optimisation techniques have been applied to the MONRP, including greedy search, hill climbing, simulated annealing, genetic algorithms, NSGA-II, MoCell and Ant-based search [3,8,11,12]. Since NSGA-II [7] has demonstrated acceptable results in a number of empirical studies on MONRP [10,11,13], we choose it as the basis of solution calculation.

2.4 MDE and Model Management

Our modelling approach and supporting tools are based on MDE principles [14] and technologies. The tools we present in later sections exploit Eclipse's EMF/Ecore for defining models and DSLs, and use the Epsilon platform [15] for automatically generating solutions to acquisition problems. Epsilon is a framework of task-specific languages for model management, and comprises a core language (EOL) upon which other task-specific languages are defined. These include a model-to-model transformation language (ETL) [16], a model migration language (Flock) [17], a model-to-text language (EGL), and a tool for generating GMF editors (EuGENia) [4].

The tool that supports our acquisition modelling approach is implemented as a set of DSLs, general model management code and model-to-model (M2M) transformations. Graphical editor support for the tool is provided via EuGENiA and GMF [5].

3 Modelling Approach

As part of a collaboration between MooD International and the University of York, we have developed a general-purpose modelling approach and tool designed to support acquisition problems. The approach is based on a set of DSLs that are used to model an acquisition problem as well as mechanisms and resources that can contribute towards fulfilling the problem's goals. We first describe the modelling approach, and then explain how the tool calculates solutions, using MDE techniques.

3.1 Concept and DSLs

The modelling approach is based on notions of goal modelling [18]. Goal models are normally applied to acquisition problems by first defining the top-level goals of what is required; these top-level goals are then decomposed into sub-goals until it is possible to identify a system or process that enables the goal to be fulfilled. In some cases, it is possible to determine whether a system or process *satisfies* a goal; in others, a looser notion, *satisficing*, is used to indicate that a system or process goes some way to meeting a goal.

Most approaches to calculating solutions to goal models terminate after identifying a complete goal model, i.e., one in which all goals are fulfilled. However, this is insufficient for solving acquisition problems in their full generality. For example, consider Through-Life Capability Management (TLCM) [1]: there are multiple ways for the same goals to be met by completely different supporting systems. Moreover, different acquisition strategies meet the goals to different degrees and have different costs.

Our modelling approach separates the modelling of the goals from the modelling of the systems and processes and how they interact. By underpinning the goal model and the system and process models (component models) with domain specific modelling languages, we can use MDE techniques to freely manipulate them. The use of MDE allows us to manipulate these models to automatically compose completed goals models. Additionally, with the aid of a multi-objective search technique, we can search for a Pareto front of completed goal models representing the various acquisition trade-offs to aid the acquisition decision makers. Goal models are a generic acquisition technique. This means that we can perform trade-off analysis on a very general class of acquisition problem. Being able to manipulate goal models using MDE techniques is the main enabling contribution that enables this. The application of our approach to NRP is purely illustrative.

The modelling of the high level goals and how they decompose is contained in the *Scenario Model*, which is captured in the Scenario DSL, illustrated in Fig. 2.

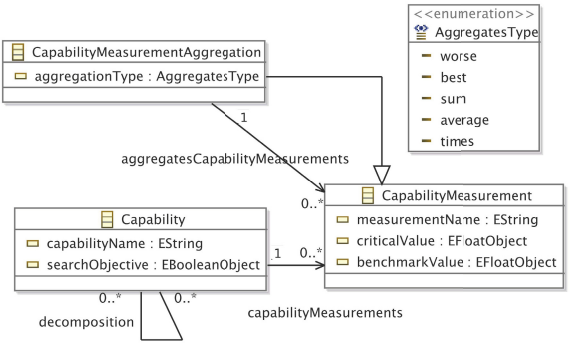


Fig. 2. Scenario Metamodel in Ecore

The main element in the Scenario DSL is the concept of a *Capability*, which is a TLCM term for a Requirement or Goal [1]. Capabilities can be decomposed into sub-capabilities and can have associated quantitative measures. Capability measurements are either fulfilled directly by a *CapabilityProvision* (in a Component Model, see Fig. 3), or indirectly through the *CapabilityMeasurementAggregation* model element. This states how the measure is derived from other capability measurements. Capabilities can either be not fulfilled (typically by the critical value for the measurement not being reached), partially fulfilled (by the critical measurement being met), or completely fulfilled by the benchmark value being met.

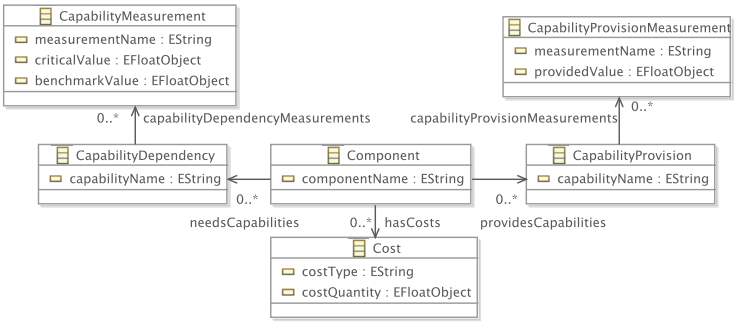


Fig. 3. Simplified Component Metamodel in Ecore

The modelling of the systems/people/processes that can satisfy the goals in the *Scenario Model* are captured in *Component Models*; the abstract syntax is illustrated in Figure 3. The main element in the DSL is *Component*, which represents some system or process that can be acquired. Each component has a name, can provide multiple capabilities, can itself depend on multiple different capabilities, and can have multiple costs (this last element is essential, because some components will cost more in different contexts, e.g., a component used in a safety critical project may require more review by experts, and hence will be more expensive). There is an essential notion of coherency and completeness with components: if a dependency associated with a component is unsatisfied, that component cannot satisfy any other component with a provision.

An end-user models their goals (using the Scenario DSL) and the components (using the Component DSL). We then use MDE techniques to implement search-based algorithms to calculate optimal solutions; this is described in the next section.

3.2 Calculating Solutions to the Acquisition Problem

After specifying a set of goals and available components that can contribute to solutions, we want to calculate solutions to the acquisition problem. This is a search problem. However, it is a non-trivial search problem for reasons discussed earlier: components can contribute to multiple goals; a goal may have multiple possible components that

can fulfil it; and, ultimately, the problem may have an arbitrary number of solutions. As such, we need to define a notion of what constitutes an *optimal* solution for such multi-objective problems. Our modelling approach thus calculates a Pareto front. The objectives for the Pareto front are set within the Scenario Model.

The goal model is constructed via a chain of M2M transformations that connect *Capability* model elements from the Scenario DSL with matching *CapabilityProvision* model elements from the Component DSL, thus creating a relationship between them in the Satisfied Scenario DSL. The *CapabilityDependency* in the Component DSL will also be matched with a *CapabilityProvision* in another Component. During this process, when a *Capability* or *CapabilityDependency* model element is connected to a *CapabilityProvision* model element, the provided values from the *CapabilityProvisionMeasurement* model elements attached to the *CapabilityProvision* are compared with the critical and benchmark values from the *CapabilityMeasurements* model elements attached to the *Capability* or *CapabilityDependency* model element to determine how well the *Capability* or *CapabilityDependency* is satisfied. There is normally more than one *Component* which can satisfy a *Capability* and in some cases it may be preferable for a *Capability* to remain unfulfilled (e.g. to reduce costs). Consequently, the transformations are *stochastic*.

The overall approach is illustrated in Fig. 4. An end-user provides a single scenario model, multiple component models and a user interface model. The user interface model merely provides the settings for the search algorithm. The scenario model and component models are transformed via a M2M transformation into an intermediate DSL which express a correspondence [19] (Correspondence model). The correspondence model only holds information on *how* the models involved in the search can be *structurally connected* to each other. The next M2M transformation is stochastic and produces a completed correspondence model that captures *exactly one way in which* the models can be connected. This M2M transformation is executed multiple times to generate an initial population for the search method (described below). The completed correspondence model does not capture details such as measurements, costs, etc. A final M2M transformation is used to produce satisfied scenario models, which are the completed goal models that can be evaluated against their objectives. The transformation takes in the structural information from the completed correspondence model and the details from the original scenario and component models. The satisfied scenario metamodel is a composition of the scenario and component meta models with additional relationship for connecting them together.

To derive a Pareto front of solutions, we utilise a search method based on the multi-objective optimisation algorithm NSGA-II [7]. The starting point is the initial population, termed the 'parent' population, of different completed correspondence models produced by the stochastic process described above, and the search proceeds as a sequence of iterations, typically called 'generations', as follows. An 'offspring' population is created by applying a single-point crossover operator that swaps parts of two randomly-chosen models from the parent population to produce a new completed correspondence model. Each new model generated in this way is evaluated against the

objectives using a M2M transformation to create a corresponding satisfied scenario model. The parent and offspring populations are then combined and the solutions ordered by non-domination rank. (The concepts of domination and non-domination rank are explained in Section 2.1) The better half of the combined population – the solutions with lowest non-domination rank – are chosen to form a new parent population. The algorithm continues in this way until a specified number of generations is reached. The output is a Pareto front of satisfied scenario models representing the different possible trade-offs between the objectives.

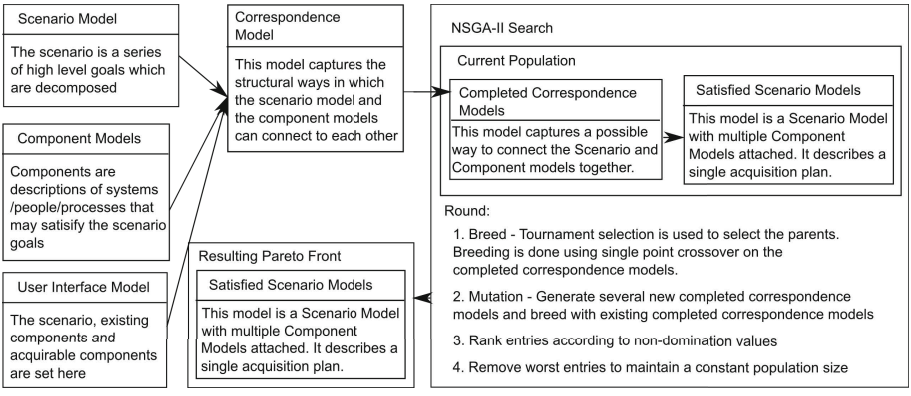


Fig. 4. Solution architecture

The approach offers a number of novelties, including the following.

- *Automatic construction and evaluation of goal models.* The tool uses model management to take a partially decomposed goal model along with components models (representing systems/processes/people) and uses them to automatically generate multiple completed goal models, which represent different viable acquisition plans. The completed goal model is automatically evaluated by using the measures contained in the goal model and the component models used in its construction.
- *Trade-off support between stakeholder goals.* The approach has the ability to find the Pareto front between different stakeholder goals and the different involved costs when generating solutions to the acquisition problem of interest. This enables decision makers to consider trade-offs in the acquisition plans which is useful in cases where there are limited resources and multiple ways to achieve the same goals such as in Through Life Capability Management.
- *Solution visualisation.* The tool generates solutions to acquisition problems using the satisfied scenario DSL. Since a DSL is used, the model, containing the acquisition plan, can be visualised using MDE tools such as EuGENia [4] and GMF [5]. The solution visualisation is illustrated briefly in the next section.

4 Application to the Next Release Problem

We illustrate the modelling approach on instances of the Multi-Objective Next Release Problem (MONRP). First, by using a small representative example, we will demonstrate the modelling approach (and supporting tools, implemented using the Epsilon framework), the calculation of solutions, show how the approach can support problems that are not normally supported by NRP tools. Second, to explore scalability, we apply the approach to a much larger randomly generated dataset for a different instance of the MONRP.

4.1 Stock Control System Example

Our first example is based on the following scenario. A small shop is considering upgrading their existing stock control system. There are two main stakeholders: the shop manager (paying for the upgrade) and shop clerk (who uses the system). The shop manager has three requirements for the upgraded system: monthly reports on the shop's stock flow, email notifications for when stock needs to be reordered, and an automatic ordering system for new stock. The shop clerk also has three requirements: means to track stock (other than manual stock tracking), a better user interface for the system, and a requirement shared with the manager: that of automated stock ordering.

Producing email notifications and automatically ordering stock both depend on common code for determining when stock of an item is likely to run out. There are two different potential improved stock control systems: a cheaper barcode system and a more expensive RFID tag system which is easier to use and better at tracking stock. Such scenarios, where two different component satisfy the same requirement to different degrees, are not supported by existing NRP methods. Additionally, the manager's requirements are deemed more important than the clerk's. Moreover, the manager does not have enough money available to pay for all the upgrades.

4.2 Scenario Model

The first step is to model the acquisition problem in the Scenario DSL (Fig. 5). The Scenario DSL captures both the structure of the MONRP itself, the details of the two customers (manager, clerk) and their requirements. The objective is to calculate the weighted sum of the customer satisfactions. The capability *Next Release Problem* is the search objective and its measure is the sum of the shop manager's and shop clerk's satisfaction, which are weighted. The shop manager and shop clerk are modelled as capabilities which decompose into their requirements, also modelled as capabilities. A requirement whose satisfaction can be measured by a real value (as opposed to a requirement that is either fully met or not met at all) is called a continuous variable requirement. Continuous variable requirements are supported by the approach by giving each requirement a measure with a critical value of 0 and a benchmark value of 1. A

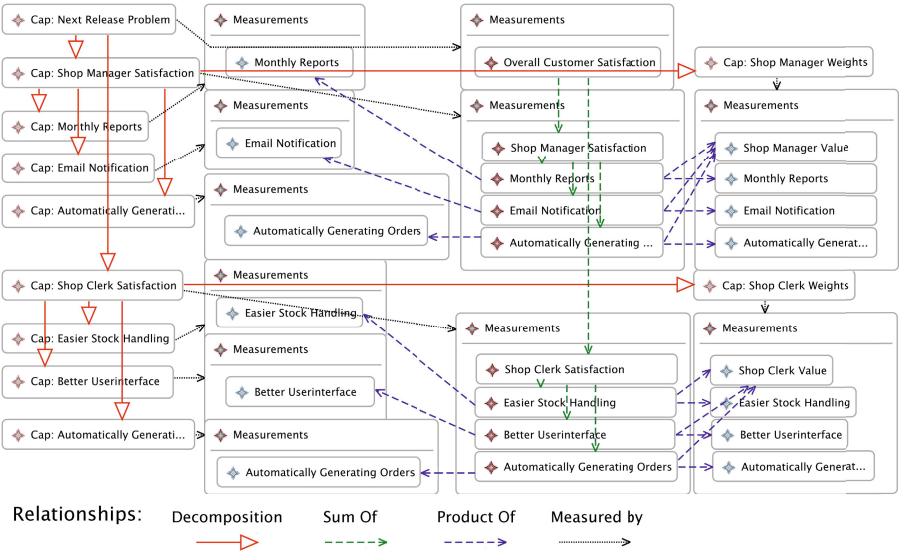


Fig. 5. Shop Case Study Scenario Model

slight modelling quirk is that because the Scenario DSL cannot hold numerical values, the weights have to be modelled in the Component DSL and therefore there are capability decompositions for the shop manager and the shop clerk which take into account the numerical weights for each [1]. The two weight components are set in the User Interface model as pre-existing components and so are always included in the generated goal models. The customer satisfactions have attached measures that are numerical and match the MONRP definition by using the multiplication and sum aggregations relationships.

4.3 Component Models

Examples of the component models for this acquisition problem are shown in Fig. 6. These examples demonstrate, for instance, that the *Barcode Scanning System* is dependent on the *Stock Management System*, will cost £120 to implement, and will provide *Easier Stock Handling*.

Based on these component models, our approach will generate a Pareto front of goal models using the Satisfied Scenario DSL. The search space in this illustrative problem is reasonability small so the tool quickly finds the optimal Pareto front of solutions. The Pareto front has been extracted from the result set using a model-to-text transformation in Epsilon; it is presented in Fig. 8. In this example, suppose that the shop manager has a budget of £250 and initially selects the solution that costs £230. This solution is

[1] A dedicated model element for this situation as now been introduced in the tool.

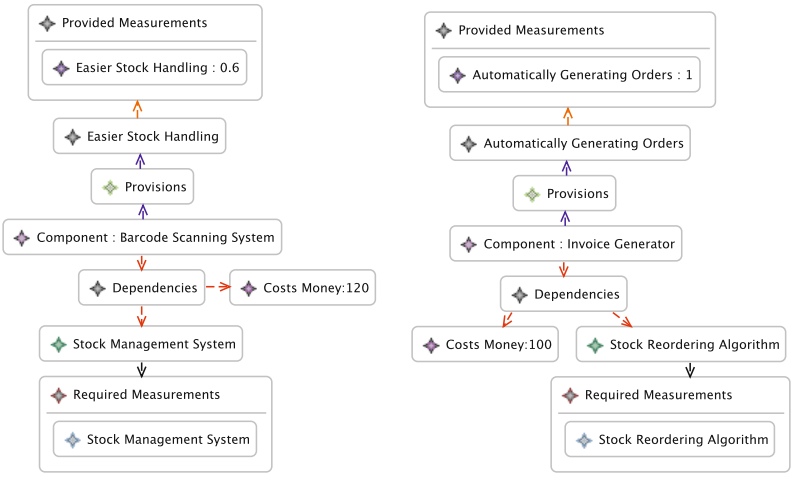


Fig. 6. Shop Case Study Example Component Models

visualised in Fig. 7, which demonstrates how the components are related to the capabilities in order to solve the problem. In the figure, *Capabilities* begin with "Cap:" and *Components* begin with "Comp:".

The shop clerk, seeing that the solution in Fig. 7 only satisfies one of his requirements, is unhappy. To resolve this, the search objectives in the Scenario Model are changed from the Next Release Problem to the two customer satisfactions. This generates a 3-dimensional Pareto front between the shop manager, shop clerk and the cost (Fig. 4.4). After some discussion between the shop manager and the shop clerk, they decide to choose a balanced solution, which gives them both a satisfaction of 0.7 for the cost of £275. The shop clerk agreed to have the extra £25 deducted from his pay.

This illustrates an advantage of an explicitly multi-objective approach to the problem. By deriving a set of potential solutions on the Pareto front, the stakeholders are presented with detailed information on which to base a discussion of trade-offs among the competing objectives.

4.4 Scalability Example

The previous example is small and illustrative; to better assess scalability, we used our approach and tool on a much larger, randomly generated MONRP problem with 5 customers, 50 requirements and 50 components. The customer and requirement weights are all randomly generated and normalised.

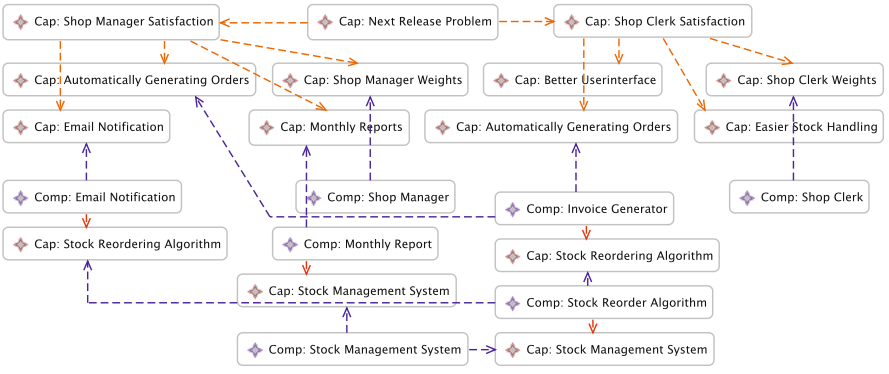


Fig. 7. Shop Case Study Example Solution

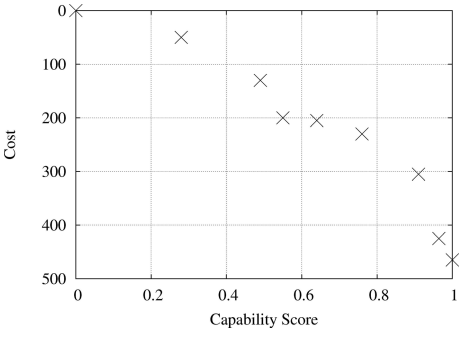


Fig. 8. Shop Case Study Pareto Front

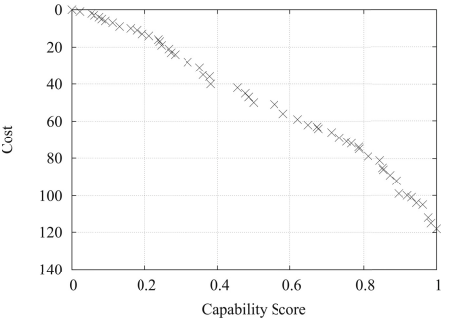


Fig. 9. Scalability Example Pareto Front

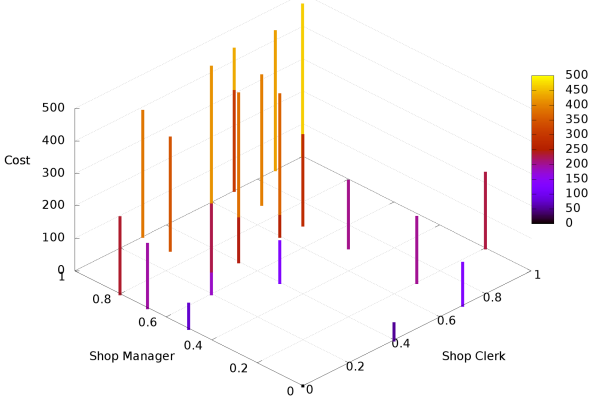


Fig. 10. Shop Case Study 3D Pareto Front

For each requirement a component is generated that provides that requirement. Additionally, there is a 50% chance the component provides another requirement and a 50% chance the component depends on a randomly chosen requirement. The generated Pareto front (shown in Fig. 9) shows that the relationship between customer satisfaction and cost begins near linear, as the software developer can start by selecting software features that are cheap to implement and are highly desirable to their customers. However, as more of the software features are implemented by the software developer, the remaining software features for the software developer to select from contains progressively less desirable and more expensive ones. The result of this is a slight curve in the Pareto front with the last 20% of the customer satisfaction begin twice as expensive to gain as the first 20% of the customer satisfaction.

For the search, a typical population size of 100 and generation count of 100 has being used. The runtime of our prototype on such a problem is approximately 5.5 hours. The randomly generated test data is larger then the largest real world problem in the NRP research field, the Motorola Problem [13]. Our tool is, however, several orders of magnitude slower than dedicated tools for MONRP, which can run in under 1 second [13]. Our focus up until now has not been on performance, but on genericity: our approach supports general acquisition problems and is not specifically tailored for MONRP. Some of the speed reduction we see is however inherent in using general-purpose modelling tools instead of bit strings. A significant reason for lower performance is that the M2M transformation languages are interpreted rather than compiled. In future work, the core M2M transformations may be reimplemented in a compiled language which should significantly reduce the execution time.

4.5 Contributions to the Next Release Problem

Our modelling approach and prototypical tool contribute several specific novelties to the Next Release Problem:

- *Trade-offs in software architecture.* By applying a principled modelling approach, we are able to represent the problem domain concept of customer requirements and the solution domain concept of components separately. In earlier work [9,10], the customer requirements and components were treated identically. As a result, our approach permits situations in which multiple different components can fulfil the same customer requirement. This corresponds to trade-offs in software architecture and these are not supported by previous work on the NRP.
- *Continuous variable requirements.* A proposed research challenge for the NRP is the handling of continuous variable requirements [2], i.e. requirements whose satisfaction may be partial. For example, a requirement on a web server's response time. It could be a critical that the response time (the continuous variable) is under 300ms and desirable for the response time to be under 100ms. This is easily supported by our tool since it tackles in the problem in a conceptually cleaner way, whereas in previous work [3,8,10,11] on the NRP problem, a requirement can only be satisfied or not satisfied.
- *Visualisation of solutions.* Another research challenge for the NRP is explaining to the stakeholder why solutions are good [2]. By supporting the visualisation of

solutions to show how customer requirements are fulfilled by different components, our tool partially addresses this concern.
- *Tool Flexibility.* Our tool is generic and allows the end-user to change the problem definition. Here, for example, we have used this flexibility to produce a 3-dimensional Pareto Front of two stakeholder's satisfaction against the cost.

5 Conclusions and Further Work

The paper contributes a general approach, based on Model-Driven Engineering (MDE) concepts and technologies, for modelling acquisition problems and calculating optimal solutions. By doing so, the approach supports engineers in carrying out trade-offs between different acquisition strategies, some of which may never have occurred to the engineers, because they were unable to exhaustively identify all the potential solutions.

The paper discussed the typical challenges present in acquisition problems, using the multi-objective Next Release Problem as an example. In particular, we have demonstrated some of the typical problems associated with calculating solutions and presenting them to the end-user. We have presented our modelling approach, which is based on a number of domain-specific languages and is inspired by goal modelling, and explained how the languages are used to specify acquisition scenarios and potential capabilities, and algorithms for matching solutions against acquisition scenarios, in order to present solutions that are Pareto optimal. A prototype tool was presented, which automatically generates solution models. Both the automatic generation of goal models from the problem description, and the descriptions of the available systems, people and processes that can be acquired, as well as the generation of multiple instead of single goal models to show tradeoffs, are novelties of this work.

We then applied the modelling approach to concrete instances of the Next Release Problem, demonstrating that the approach and supporting tool can be used on both significant examples of the NRP, as well as more general NRP problems than those handled by other approaches (e.g., with dependencies between requirements). Overall, the approach that we have taken contributes a more generic capability for handling NRP problems; the price that we pay comes in the form of performance: because our tools are based on general-purpose MDE tools (i.e., Eclipse EMF, Epsilon) instead of dedicated NRP tools, our approach is less efficient; however, the approach does scale, as our example in the previous section demonstrates.

In future work, the tool will be applied to more general acquisition scenarios and on real world case studies. Since the work is primary motivated by the Through Life Capability Management (TLCM), we will enhance the tool to support multiple releases, similar to earlier work by Greer et al [8]. Additionally, we will manage the additional complexities that arise through TLCM, most notably the larger time scales involved (i.e., acquisition processes that take decades). We also intend to investigate mechanisms for performing sensitivity analysis, to check for robustness of acquisition plans.

Acknowledgements. This work is being sponsored by MooD International and the EPRSC (EP/F501374/1) under the Large Scale Complex IT System research programme.

References

1. McKane, T.: Enabling acquisition change - an examination of the Ministry of Defence's ability to undertake Through Life Capability Management. Technical report (June 2006)
2. Zhang, Y.-Y., Finkelstein, A., Harman, M.: Search Based Requirements Optimisation: Existing Work and Challenges. In: Rolland, C. (ed.) REFSQ 2008. LNCS, vol. 5025, pp. 88–94. Springer, Heidelberg (2008)
3. Bagnall, A.J., Rayward-Smith, V.J., Whittley, I.M.: The Next Release Problem. Information and Software Technology 43(14), 883–890 (2001)
4. 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. LNCS, vol. 6394, pp. 211–225. Springer, Heidelberg (2010)
5. Eclipse GMF - Graphical Modeling Framework, http://www.eclipse.org/gmf
6. Deb, K.: Multi-objective optimization. In: Burke, E.K., Kendall, G. (eds.) Search Methodologies, pp. 273–316. Springer, US (2005)
7. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Transactions on Evolutionary Computation 6(2), 182–197 (2002)
8. Greer, D., Ruhe, G.: Software release planning: an evolutionary and iterative approach. Information and Software Technology 46(4), 243–253 (2004)
9. Baker, P., Harman, M., Steinhofel, K., Skaliotis, A.: Search based approaches to component selection and prioritization for the next release problem. In: 22nd IEEE International Conference on Software Maintenance, ICSM 2006, pp. 176–185 (2006)
10. Zhang, Y., Harman, M., Mansouri, S.A.: The multi-objective next release problem. In: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, pp. 1129–1137 (2007)
11. Durillo, J.J., Zhang, Y.Y., Alba, E., Nebro, A.J.: A study of the multi-objective next release problem. In: 1st International Symposium on Search Based Software Engineering, pp. 49–58 (2009)
12. del Sagrado, J., del Águila, I.M., Orellana, F.J.: Ant colony optimization for the next release problem: A comparative study. In: Second International Symposium on Search Based Software Engineering, pp. 67–76 (2010)
13. Durillo, J.J., Zhang, Y., Alba, E., Harman, M., Nebro, A.J.: A study of the bi-objective next release problem. In: Empirical Software Engineering, pp. 1–32 (2011)
14. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. Computer 39, 25–31 (2006)
15. Kolovos, D.S.: An Extensible Platform for Specification of Integrated Languages for Model Management. PhD thesis, University of York (2008)
16. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon Transformation Language. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 46–60. Springer, Heidelberg (2008)
17. 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)
18. Lamsweerde, A.V., Dardenne, A., Delcourt, B., Dubisy, F.: The KAOS Project: Knowledge acquisition in automated specifications of software. In: Proceeding AAAI Spring Symposium Series, Track: Design of Composite Systems (1991)
19. Bézivin, J., Bouzitouna, S., Del Fabro, M., Gervais, M.P., Jouault, F., Kolovos, D., Kurtev, I., Paige, R.: A canonical scheme for model composition. In: Model Driven Architecture–Foundations and Applications, pp. 346–360 (2006)

Author Index