Pieter Van Gorp
Tom Ritter
Louis M. Rose (Eds.)

LNCS 7949

Modelling Foundations and Applications

9th European Conference, ECMFA 2013
Montpellier, France, July 2013
Proceedings

Springer

Lecture Notes in Computer Science 7949

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

Editorial Board

David Hutchison
Lancaster University, UK
Takeo Kanade
Carnegie Mellon University, Pittsburgh, PA, USA
Josef Kittler
University of Surrey, Guildford, UK
Jon M. Kleinberg
Cornell University, Ithaca, NY, USA
Alfred Kobsa
University of California, Irvine, CA, USA
Friedemann Mattern
ETH Zurich, Switzerland
John C. Mitchell
Stanford University, CA, USA
Moni Naor
Weizmann Institute of Science, Rehovot, Israel
Oscar Nierstrasz
University of Bern, Switzerland
C. Pandu Rangan
Indian Institute of Technology, Madras, India
Bernhard Steffen
TU Dortmund University, Germany
Madhu Sudan
Microsoft Research, Cambridge, MA, USA
Demetri Terzopoulos
University of California, Los Angeles, CA, USA
Doug Tygar
University of California, Berkeley, CA, USA
Gerhard Weikum
Max Planck Institute for Informatics, Saarbruecken, Germany

Pieter Van Gorp Tom Ritter
Louis M. Rose (Eds.)

Modelling Foundations and Applications

9th European Conference, ECMFA 2013
Montpellier, France, July 1-5, 2013
Proceedings

Springer

Volume Editors

Pieter Van Gorp
Eindhoven University of Technology
School of Industrial Engineering
P.O. Box 513, 5600 MB, Eindhoven, The Netherlands
E-mail: p.m.e.v.gorp@tue.nl

Tom Ritter
Fraunhofer FOKUS
Kaiserin-Augusta-Allee 31, 10589 Berlin, Germany
E-mail: tom.ritter@fokus.fraunhofer.de

Louis M. Rose
University of York
Department of Computer Science
Deramore Lane, Heslington, York, YO10 5GH, UK
E-mail: louis.rose@york.ac.uk

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-39012-8 e-ISBN 978-3-642-39013-5
DOI 10.1007/978-3-642-39013-5
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013940513

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

© Springer-Verlag Berlin Heidelberg 2013
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

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 2013 European Conference on Modelling Foundations and Applications (ECMFA 2013) was organized by the Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier (LIRMM), CNRS, University Montpellier 2, France, and held during July 1–5, 2013.

ECMFA is the key European conference aiming at advancing the techniques and furthering the underlying knowledge related to 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 8 years, ECMFA has provided an ideal venue for interaction among researchers interested in MDE both from academia and industry. The ninth edition of the conference covered major advances in foundational research and industrial applications of MDE.

In 2013, the Program Committee received 76 abstracts and 51 full paper submissions. From these, 9 Foundations Track papers and 6 Applications Track papers were accepted for presentation at the conference and publication in these proceedings, resulting in an acceptance rate of 23% for the Foundations Track, and an overall acceptance rate of 29.4%. Papers on all aspects of MDE were received, including topics such as model querying, consistency checking, model transformation, model-based systems engineering, and domain-specific modeling. 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 2013 keynote speakers were Martin Gogolla (University of Bremen) and Dierk Steinbach (Cassidian, EADS Deutschland GmbH). 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 and all authors who submitted papers to ECMFA 2013.

July 2013

Pieter Van Gorp
Tom Ritter
Louis M. Rose

Organization

ECMFA 2013 was organized by the Laboratoire d'Informatique de Robotique et de Microélectronique de Montpellier (LIRMM), CNRS, University Montpellier 2, France.

Program Committee

Jan Aagedal
Vasco Amaral
Terry Bailey
Marco Bambrilla
Reda Bendraou
Xavier Blanc
Behzad Bordbar
Jordi Cabot
Tony Clark
Benoit Combemale
Zhen Ru Dai
Zinovy Diskin
Gregor Engels
Anne Etien
Robert France
Mathias Fritzsche
Jesus Garcia Molina
Sebastien Gerard
Marie-Pierre Gervais
Holger Giese
Martin Gogolla
Pieter Van Gorp
Jeff Gray
Esther Guerra
Michael R. Hansen
Reiko Heckel
Markus Heller
Andreas Hoffmann
Muhammad Zohaib Iqbal
Teemu Kanstren
Jochen Kuester
Thomas Kühne
Vinay Kulkarni
Ivan Kurtev
Roberto Lopez-Herrejon
Dragan Milicev
Claus Pahl
Alfonso Pierantonio
Ivan Porres
Juan Antonio de la Puente
Olli-Pekka Puolitaival
Arend Rensink
Laurent Rioux
Tom Ritter
Louis Rose
Ella Roubtsova
Julia Rubin
Bernhard Rumpe
Andrey Sadovykh
Houari Sahraoui
Bernhard Schätz
Andy Schürr
Bran Selic
Ali Shaukat
Marten van Sinderen
Renuka Sindghatta
John Slaby
James Steel
Alin Stefanescu
Harald Störrle
Ragnhild Van Der Straeten
Eugene Syriani
Gabriele Taentzer
Francois Terrier
Juha-Pekka Tolvanen
Salvador Trujillo

Andreas Ulrich
Hans Vangheluwe
Daniel Varró
Cristina Vicente-Chicote
Markus Voelter
Edward D. Willink
Manuel Wimmer
Tao Yue
Gefei Zhang
Steffen Zschaler

Additional Reviewers

Mathieu Acher
Mustafa Al-Lail
Bruno Barroca
Phillipa Bennett
Steven Bosems
Benjamin Braatz
Moises Branco
Jacome Cunha
Juan de Lara
Joachim Denil
Regina Hebig
Stephan Hildebrandt
Juan F. Ingles-Romero
Ludovico Iovino
Niels Joncheere
Udo Kelter
Mirco Kuhlmann
Philip Langer
Florian Mantz
David Mendez
Dmytro Plotnikov
Alexander Roth
Oscar Sanchez-Ramon
Concepción Sanz-Pineda
Christoph Schulze
Mathias Soeken

Sponsoring Institutions

Gold: Oracle
Silver: Bouygues Telecom
Bronze: IBM Research, Microsoft Research, Typesafe
Academic: GDR GPL, Region LR, Montpellier-Agglomration

Table of Contents

Foundations

Employing the Object Constraint Language in Model-Based Engineering 1
Martin Gogolla

MOCQL: A Declarative Language for Ad-Hoc Model Querying 3
Harald Störrle

Supporting Different Process Views through a Shared Process Model ... 20
Jochen Küster, Hagen Völzer, Cédric Favre, Moises Castelo Branco, and Krzysztof Czarnecki

Characterization of Adaptable Interpreted-DSML 37
Eric Cariou, Olivier Le Goaer, Franck Barbier, and Samson Pierre

Transformation as Search 54
Mathias Kleiner, Marcos Didonet Del Fabro, and Davi De Queiroz Santos

Model-Based Generation of Run-Time Monitors for AUTOSAR 70
Lars Patzina, Sven Patzina, Thorsten Piper, and Paul Manns

End-User Support for Debugging Demonstration-Based Model Transformation Execution 86
Yu Sun and Jeff Gray

DPMP: A Software Pattern for Real-Time Tasks Merge 101
Rania Mzid, Chokri Mraidha, Asma Mehiaoui, Sara Tucci-Piergiovanni, Jean-Philippe Babau, and Mohamed Abid

Using Model Types to Support Contract-Aware Model Substitutability 118
Wuliang Sun, Benoit Combemale, Steven Derrien, and Robert B. France

Applying a Def-Use Approach on Signal Exchange to Implement SysML Model-Based Testing 134
Fabrice Ambert, Fabrice Bouquet, Jonathan Lasalle, Bruno Legeard, and Fabien Peureux

Applications

A Network-Centric BPMN Model for Business Network Management . . . 152
Daniel Ritter

Design Management: A Collaborative Design Solution 165
Maged Elaasar and James Conallen

Umbra Designer: Graphical Modelling for Telephony Services 179
Nicolás Buezas, Esther Guerra, Juan de Lara, Javier Martín, Miguel Monforte, Fiorella Mori, Eva Ogallar, Oscar Pérez, and Jesús Sánchez Cuadrado

Experience with Industrial Adoption of Business Process Models for User Acceptance Testing 192
Deepali Kholkar, Pooja Yelure, Harshit Tiwari, Ajay Deshpande, and Aditya Shetye

A Case Study in Evidence-Based DSL Evolution 207
Jeroen van den Bos and Tijs van der Storm

Model Driven Software Development: A Practitioner Takes Stock and Looks into Future 220
Vinay Kulkarni

Author Index 237

Employing the Object Constraint Language in Model-Based Engineering

Martin Gogolla

Database Systems Group, University of Bremen, Germany
gogolla@informatik.uni-bremen.de

Abstract. MBE (Model-Based Engineering) proposes to develop software by taking advantage of models, in contrast to traditional code-centric development approaches. If models play a central role in development, model properties must be formulated and checked early on the modeling level, not late on the implementation level. We discuss how to validate and verify model properties in the context of modeling languages like the UML (Unified Modeling Language) combined with textual restrictions formulated in the OCL (Object Constraint Language).

Typical modeling and transformation languages like UML (Unified Modeling Language), EMF (Eclipse Modeling Framework), QVT (Queries, Views, and Transformations) or ATL (Atlan Transformation Language) are complemented by the textual OCL (Object Constraint Language) enriching graphical or textual models with necessary details. Models allow the developer to formulate essential system properties in an implementation- and platform-independent way.

Precise object-oriented development must take into account system structure and system behavior. The system structure is often captured by class diagrams and can be instantiated in terms of prototypical exemplars by object diagrams. The system behavior can be determined by statechart diagrams, and system execution traces can be demonstrated by sequence diagrams. OCL restricts the possible system states and transitions through the use of class invariants, operation pre- and postconditions, state invariants, and transition pre- and postconditions. OCL can also be used during model development as a query language.

Modeling features and their analysis through validation and verification must be supported by tools like, for example, the tool USE (UML-based Specification Environment) [1]. Within USE, UML class, object, statechart, and sequence diagrams extended with OCL are available [2]. OCL has been extended with programming language features in SOIL (Simple Ocl-like Imperative Language) which allows the developer to build operation realizations on the modeling level without having to dig into implementation level details [3]. Thus models in USE are executable, but a prescriptive SOIL model for operations can be checked against descriptive plain OCL pre- and postconditions.

Tools like USE assist the developer in order to validate and to verify model characteristics. Validation and verification can be realized, in USE for example, by employing a so-called model validator based on relational logic and SMT

P. Van Gorp, T. Ritter, and L.M. Rose (Eds.): ECMFA 2013, LNCS 7949, pp. 1–2, 2013.
© Springer-Verlag Berlin Heidelberg 2013

solvers [4]. Model properties to be inspected include consistency, redundancy freeness, checking consequences from stated constraints, and reachability [5]. These properties are handled on the conceptual modeling level, not on an implementation level. Employing these instruments, central and crucial model and transformation model characteristics can be successfully analyzed and checked.

Transformation models [6] provide an approach that formulates model transformations in a descriptive, not prescriptive way by stating desired properties of transformations in terms of input and output models and their relationship. From transformation models, tests [7] can be derived that are independent from the employed transformation language. Modeling and model transformations with USE have been successfully applied in a number of projects, for example, in the context of public authority data interchange [8].

Acknowledgement. The force and energy of Mark Richters, Jörn Bohling, Fabian Büttner, Mirco Kuhlmann and Lars Hamann formed USE as it stands today. Thanks!

References

1. 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)
2. Hamann, L., Hofrichter, O., Gogolla, M.: Towards Integrated Structure and Behavior Modeling with OCL. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 235–251. Springer, Heidelberg (2012)
3. Büttner, F., Gogolla, M.: Modular Embedding of the Object Constraint Language into a Programming Language. In: Simao, A., Morgan, C. (eds.) SBMF 2011. LNCS, vol. 7021, pp. 124–139. Springer, Heidelberg (2011)
4. Kuhlmann, M., Gogolla, M.: From UML and OCL to Relational Logic and Back. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 415–431. Springer, Heidelberg (2012)
5. 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)
6. Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A.: Model Transformations? Transformation Models! In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 440–453. Springer, Heidelberg (2006)
7. Gogolla, M., Vallecillo, A.: *Tract*able Model Transformation Testing. In: France, R.B., Kuester, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 221–235. Springer, Heidelberg (2011)
8. Büttner, F., Bartels, U., Hamann, L., Hofrichter, O., Kuhlmann, M., Gogolla, M., Rabe, L., Steimke, F., Rabenstein, Y., Stosiek, A.: Model-Driven Standardization of Public Authority Data Interchange. Science of Computer Programming (2013)

MOCQL: A Declarative Language for Ad-Hoc Model Querying

Harald Störrle

Institute of Applied Mathematics and Computer Science
Technical University of Denmark
hsto@dtu.dk

Abstract. This paper starts from the observation that existing model query facilities are not easy to use, and are thus not suitable for users without substantial IT/Computer Science background. In an attempt to highlight this issue and explore alternatives, we have created the Model Constraint and Query Language (MOCQL), an experimental declarative textual language to express queries (and constraints) on models. We introduce MOCQL by examples and its grammar, evaluate its usability by means of controlled experiments, and find that modelers perform better and experience less cognitive load when working with MOCQL than when working with OCL. While MOCQL is currently only implemented and validated for the different notations defined by UML, its concepts should be universally applicable.

Keywords: OCL, UML, model querying, empirical software engineering, Prolog.

1 Introduction

1.1 Motivation

Many software development approaches today use models instead of or alongside with code for different purposes, e. g., model based and model driven development, Domain-Specific Languages (DSLs), and Business process management. As a consequence, tasks such as version and configuration management, consistency checking, transformations, and querying of models are much more common today than they used to be. Unfortunately, these and other tasks are not well covered in current CASE tools. But from practical experience we have learned that modelers dearly need, among others, an ad-hoc query facility covering more than just full text search and a fixed set of predefined queries. The natural choice of language when it comes to selecting a powerful general-purpose model querying language is the Object Constraint Language (OCL [10]), at least for UML and similar languages. However, OCL is often perceived as too complex for many modelers, let alone domain experts without formal training in Computer Science. So, the goal of this paper is to try and come up with better solutions to this problem. In order to achieve high levels of accessibility, we are prepared to even sacrifice a certain degree of theoretical expressiveness, as long as the practically relevant cases are covered. Aspects other than usability are beyond the scope of this paper.

P. Van Gorp, T. Ritter, and L.M. Rose (Eds.): ECMFA 2013, LNCS 7949, pp. 3–19, 2013.
© Springer-Verlag Berlin Heidelberg 2013

1.2 Approach

Our first approach to improving the usability of model query languages was based on the commonly held assumption that visual languages are generally easier to understand than textual languages. Since OCL is a purely textual language, a visual language might perform better. Following this assumption, we defined a predominantly visual alternative to OCL, the Visual Model Query Language (VMQL [15]). We could indeed demonstrate that VMQL is easier to use than OCL [17], while being applicable to the same kinds of tasks OCL is targeted at (i.e., both for querying and expressing constraints, see [7] for the latter).

During these research projects, however, we also found evidence that the visual nature of VMQL is only one of several factors contributing to its usability, and quite possibly not the largest one. In order to pursue this lead, and building on the lessons learned in the design of VMQL, we invented a new textual language on the fly. To our surprise, this improvised language was even more effective than VMQL, and was preferred by users when given a choice. Thus, we refined and elaborated this language which has now evolved into the Model Constraint and Query Language (abbreviated to MOCQL, pronounce as "mockle").

The design of MOCQL is informed by the lessons learned during the design of VMQL, and they share conceptual and implementation element, yet MOCQL is a genuinely new language. Our working hypothesis is that MOCQL offers better usability than both OCL and VMQL. MOCQL is not conceptually restricted to UML, but the current implementation and validation have so far only covered this notation. Thus, while we believe that MOCQL is suitable as a *universal* model query language, no such claim will be raised here. Also, we envision using MOCQL on model repositories such as CDO, EMFStore, Morsa, and ModelBus.[1] However, this has not yet been attempted.

In the remainder of this paper, we will first introduce MOCQL by example, showing how it may be used to query models in a concise and modeler-friendly way. We provide a (simplified) grammar for MOCQL, informally describe its semantics, and briefly report on its implementation. Then, we report on two controlled experiments to assess the relative effectiveness of OCL, VMQL, and MOCQL from a user's point of view. We conclude by comparing MOCQL with existing approaches, highlighting the contributions, and outlining ongoing research.

2 Introducing MOCQL

We will now show some examples of MOCQL queries. In order to explain their meaning, we also present OCL queries with the same effect. Note that no formal relationship between MOCQL and OCL exists, in particular, there is no automatic translation between these languages. All queries are assumed to be executed on the models shown in Fig. 1. For simplicity, we shall assume that

[1] See `www.eclipse.org/cdo`, `www.emfstore.org`, `www.modelum.es/morsa`, and `www.modelbus.org`, respectively.

Fig. 1 shows two models M1 and M2, the former of which is completely covered by diagram M1, and the latter is completely covered by the diagrams M2a and M2b.

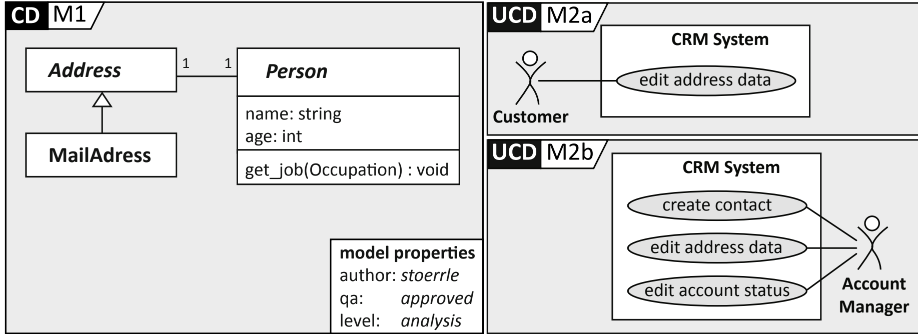


Fig. 1. The sample models used as the base model for all queries described in this paper

Suppose, a modeler is looking for all classes whose name ends with "Address" in model M1. On the given sample model, the result would be the set of the two classes named "Address" and "MailAddress". Such a simple query would probably be best answered by using whatever search facility any given modeling tool provides; most tools allow specifying the meta-class of the search target and pattern matching for names. Using OCL, the query might look something like the following.

```
(1a)   Class.allInstances()
          -> select(c | c.name.contains_substring("Address")).
```

Here, we assume an OCL-function `contains_substring`. Such a function is not part of the OCL or UML standards [10,9], but it could probably be programmed and thus be available in some hypothetical OCL Query Library. In MOCQL, this query can be expressed in a very natural way:

```
(1b)   find all classes $X named like "*Address" in M1.
```

Understanding this query requires much less knowledge about the UML meta model than the corresponding OCL expression, which makes it more readily understood by modelers (as we shall show below). Also, this expression is easy to modify and extend, e.g., we want to add the constraint that the classes we look for are abstract. In OCL, this could be expressed by query (2a).

```
(2a)   Class.allInstances()
          -> select(c | c.name.contains_substring("Address")
                        && c.isAbstract = true).
```

Compare this with the corresponding MOCQL query (2b), which we believe is significantly clearer, and easier to understand.

```
(2b)   find all abstract classes $X named like "*Address" in M1.
```

Suppose we were to look for abstract classes that do not have subclasses. The typical built-in search facilities of most modeling tools would not support such a query. In OCL, we would have to write something like the following.

```
(3a)   Class.allInstances()
           ->select(c | c.isAbstract=true).
                   intersection(c | c.general->isEmpty())
```

Even more understanding of fine details of the UML meta model are required here (e.g., the property "general"), and knowledge of the different navigational operators in OCL (i.e., the dot vs. the arrow), which at least many students struggle with. In MOCQL, we can instead write

```
(3b)    find all abstract classes $X in M1 where
              there are no classes $Y such that $X generalizes $Y
```

which was created from the first MOCQL query. This expression just requires to know the name "Generalization" for the relevant UML relationship. Thus, less knowledge about the UML meta model is required when using MOCQL as compared to when using OCL. Together with its user friendly syntax, MOCQL also allows domain experts to query models in a straightforward way.

Let us now turn to model M2. Assume, a modeler wants to find out all the actors involved in a given use case named "edit address data". In MOCQL, the following query would achieve this goal, returning the actors "Customer" and "Account Manager".

```
(4)    find all actors $X where $X is associated to $Y and
              there is a useCase $Y named "edit address data".
```

The way associations are represented in the UML meta-model would make this a rather complex query were we to express it in OCL. Observe, that elements of the UML meta model (i.e., meta classes and meta attributes) are not part of the MOCQL syntax. They are treated as strings and passed on "as is" to the query execution procedures.

MOCQL offers capabilities for all kinds of models occurring in UML, including use case models, state machine models, and activities. Suppose, for instance, we are looking for activities that contain Actions unconnected to the initial node. In MOCQL, this could be expressed by query (5).

```
(5)   find all actions $Unconnected in M3 such that
             there is no initialNode $Initial such that
                   $Initial precedes $Unconnected transitively.
```

Finally, suppose we want to query two models simultaneously, to find elements that have the same name. In MOCQL, this can easily be achieved by query (6).

```
(6)   find all $X1 named $NAME in M1 such that
               there is $X2 named $NAME in M2.
```

Relaxing the query to check for similar names rather than exact matches only requires to add `like` before the second occurrence of `$NAME`. An EBNF grammar of MOCQL is shown in Fig. 2. This grammar has been simplified for purposes of presentation.

3 Semantics

The semantics of MOCQL consists of two parts. On the one hand, there is a particular model representation that facilitates the operations involved in the query execution, storing models as Prolog knowledge bases. On the other hand, there is a mechanical translation of MOCQL queries into Prolog programs that are then executed on the knowledge base, utilizing a set of predefined Prolog predicates.

3.1 Model Representation

The model representation we use for MOCQL has been used for implementing VMLQ [17], and a set of other advanced operations on models such as clone detection [16], or difference computation [18]. In this representation, models are looked at as knowledge bases, and individual model elements are considered facts that are stored in a Prolog data base. Queries and other operations on models are implemented as Prolog predicates over this knowledge base, i.e., queries are translated into Prolog predicates by a definite clause grammar (DCG, a kind of Prolog program). Those query predicates are then simply executed, calling a small library of predefined search functions. This greatly simplifies the implementation, while making it easy to extend and experiment with, which is the main design objective at this stage of the development of MOCQL.

First, the user creates source models, exports them to an XMI file, and transforms it into a Prolog database. Each model element is transformed to one Prolog clause of the predicate `me`, see Fig. 3 for an example (edited for improved readability). The first argument of each `me`-fact is a pair of type and internal identifier (usually an integer). The second argument is a property list of tags for meta-attributes and their values. References to identifiers are marked with an `id` or `ids`-term.

This conversion has several advantages over XMI. On the one hand, it is much more compact than XMI, which also speeds up processing of models. In particular, with the given representation, we are able to keep even very large models in-memory all the time, which is not always the case for the typical data

```
COMMAND      ::= find SET_SPEC
               | count SET
               | save  SET as NAME
               | load NAME
               | assign SET (u | n) SET to VARIABLE
               | clear VARIABLE

SET          ::= SET_SPEC | VAR
VAR          ::= $A | $B | $C | ...
SET_SPEC     ::= (A_QUANTIFIER | E_QUANTIFIER) ELEMENT_SPEC

A_QUANTIFIER ::= forall | each | every | all
E_QUANTIFIER ::= [there (is | are)] [ARTICLE]
ARTICLE      ::= a | an | the | those | some | no

ELEMENT_SPEC ::= [abstract | concrete] TYPE [VAR] [NPROP] [MPROP]
                 (where | such that) PROPS

TYPE         ::= activity | activities | class | classes | useCase | ...
               | element
               | element VAR
               | element VAR with id STRING
ATTR         ::= name
               | isAbstract
               | ownedMember
               | ...

PROPS        ::= PROP
               | not PROP
               | PROP LOG_OP PROPS
PROP         ::= ATTR CMP_OP VAL
               | VAR REL_OP VAR [directly | transitively]
               | SET_SPEC

VAL          ::= defined | bool | int | float | string | ...
MPROP        ::= in MODELNAMES
MODELNAMES   ::= MODELNAME
               | MODELNAME & MODELNAMES

NPROP        ::= named STRING
               | named like PATTERN

LOG_OP       ::= and | or
CMP_OP       ::= is | are | = | has | have | is like | < | > | is not | ...

REL_OP       ::= REL_OP_AKT ARTICLE | (is | are) (ASSOC_REL | REL_OP_PAS by)
REL_OP_AKT   ::= generalizes | specializes | includes   | extends
               | follows     | precedes    | succeeds   | owns     | ...

ASSOC_REL    ::= associated to | part of

REL_OP_PAS   ::= generalized | specialized | included   | extended
               | followed    | preceeded   | succeedes  |   ...
```

Fig. 2. Simplified EBNF grammar of MOCQL

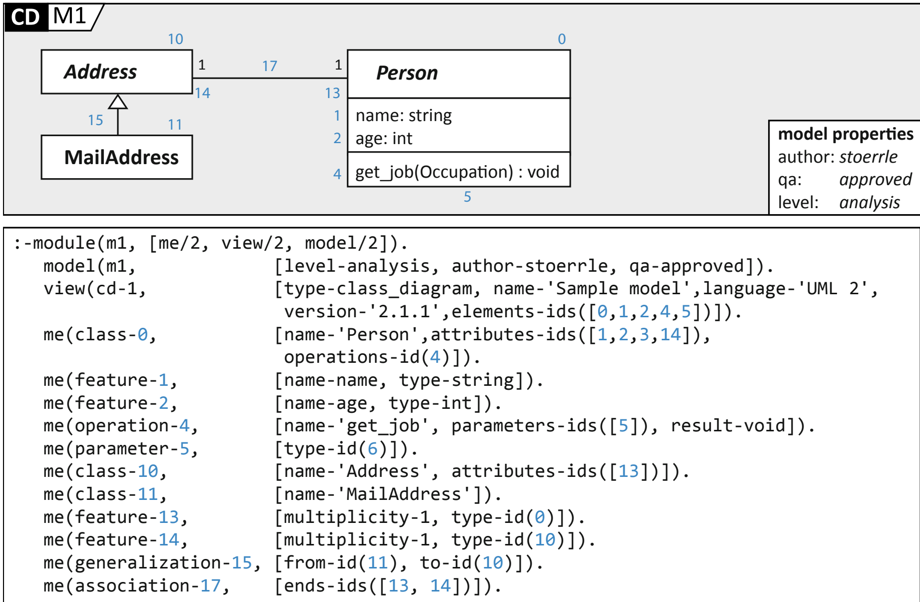


Fig. 3. Example for the Prolog representation of sample model M1

structures for processing XML-files. On the other hand, it is very easy to access models represented in this way by using the Prolog command line interface, or to exchange code operating on models while keeping the model loaded, which is a tremendous help during development. Finally, the representation is generic enough to allow for all kinds of models, including those that do not have a MOF-like meta meta model.

Moreover, observe that the conversion takes only a few milliseconds, and is fully reversible: It neither adds nor removes anything, it merely changes the model representation. The conversion is triggered automatically when trying to access an XMI file in a MOCQL query, so it is completely transparent to the user.

3.2 Query Translation

The second part of the semantics is the translation of queries into executable Prolog code. Executing the queries amounts to executing this code. Let us again consider the introductory examples from Section 2 in order to see how these are interpreted. In the first step, the query is parsed, creating an abstract syntax tree. This step reduces the syntactic sugar, i.e., it reduces plural expressions to singular, and converts syntactic alternatives into a single expression. For instance, the expression

```
(1b)   find all classes $X named like "*Address" in M1.
```

results in the parse tree shown in Fig. 4. This expression is then translated into the following sequence of Prolog predicates.

```
all( 'M1', [X], [type(class), is_like(name, '*Address')]),
show('M1', [X]).
```

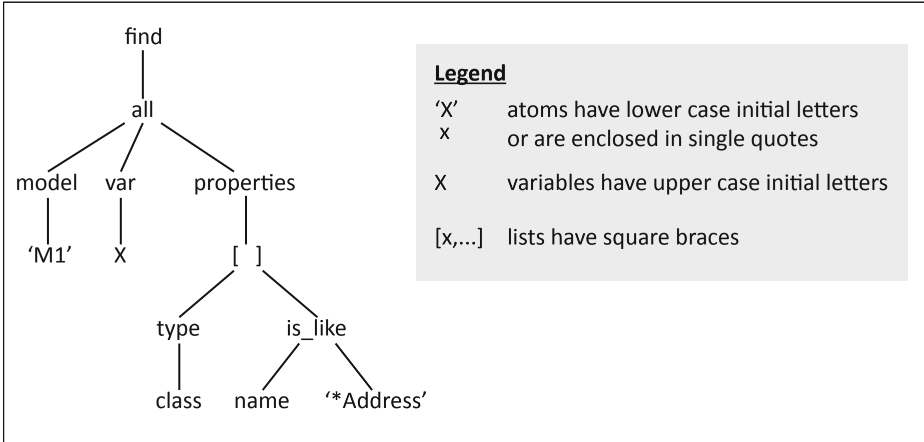


Fig. 4. The abstract syntax tree resulting from parsing and simplifying query (1b)

The predicates `all` and `show` are defined accordingly, so that executing this Prolog program actually executes the query. The atomic value `M1` refers to the name of the model to be queried, the list `[X]` is the list of all variables occurring in the expression.

Similarly, the third query example from Section 2 is translated, so that

```
(3b)   find all abstract classes $X in M1 such that
                there are no classes $Y where $X generalizes $Y
```

becomes

```
all( 'M1', [X],    [type(class), is(isAbstract,true) ]),
none('M1', [X, Y], [type(class), generalizes(X,Y)    ]),
show('M1', [X]).
```

As before, `none` is a predicate defined as part of MOCQL, so that executing this program simply executes the query. Observe that Prolog variables (i.e., `X` and `Y`) are *logical* variables, that is, they are unified rather than containers with assigned values. Thus, the sequence of the first two clauses of this program does not change the result. It does change the execution behavior, though, in particular the required computational resources.

4 Implementation

Fig. 5 shows the overall architecture of MOCQL. Processing queries is done in three steps. First, the base model is transformed from an XMI file to the Prolog transformation described in in Section 3.1 above. Observe that this transformation is purely syntactical and typically too fast for the user to notice. Then, query expressions are transformed into Prolog predicates as shown in Section 3.2, which refer to the predicates defined in the Model Querying Library. Finally, this predicate is then executed.

MOCQL shares the Model Querying Library with previous research including VMQL, but is otherwise independent. Future extensions of MOCQL to allow querying of other modeling languages such as BPMN would require some changes to this implementation. In particular, a new translator from the source model format into the Prolog format would be required, and some amendments the MOCQL grammar to cover new language concepts, i.e., the non-terminals `TYPE`, `ATTR`, and `REL_OP_AKT`. Whether amendments to the Model Querying Library would be necessary, is unclear. Thus, extending the scope of MOCQL to other modeling languages beyond UML is closer to porting a programming language to a new processor architecture than creating a new programming language.

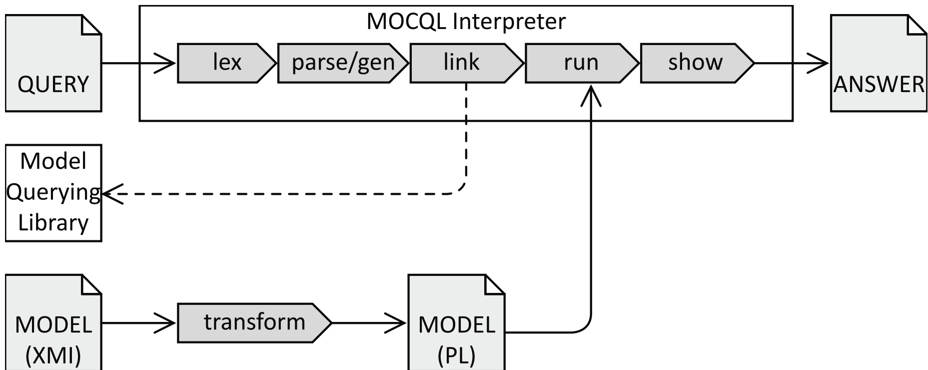


Fig. 5. Architecture of MOCQL: the Model Querying Library is shared with previous research, including VMQL

5 Usability Evaluation

In this section we evaluate MOCQL. We focus on usability, reporting two controlled experiments that study and compare the usability of VMQL, OCL, and MOCQL, respectively. At the end of this section, we briefly discuss other qualities.

5.1 Experiment Design

Both experiments used the same randomized block design, but tested different sets of languages: OCL vs. MOCQL, and VMQL vs. MOCQL, respectively. The experimental setup consisted of four parts asking for (A) demographic data, (B) finding query expressions matching an English description, (C) checking the match between a given query expression and its English description, and (D) asking for subjective judgments regarding the languages tested. Task B contained 28 subtasks while Task C contained 12 subtasks.

In each of the experiments, 17 subjects participated, most of them being graduate students, but also including six IT professionals and a researcher. Different sequences of tasks were randomly assigned to them in order to control learning effects and bias. Our study was blinded by naming the languages A, B, and C, respectively. Going by the self-assessment in the demographic part of the questionnaire (Task A), the participants had little knowledge of either of the tested languages. The participants of the second experiment were recruited from the "Elite SE study line", an educational program that admits only students of very high aptitude.

We controlled the variables language (OCL, VMQL, MOCQL), query expression, and task, and recorded the correctness of the answers, the time taken, and the subjective assessment. The latter was divided into three different measures asking for preference, effort, and confidence in the result. The experiment was run as a pen-and-paper exercise. Participants were offered to talk about the experience or comment on the questionnaire, an opportunity some of them took.

5.2 Observations

We first discuss the objective measure of the number of correct answers given by the subjects. We have normalized the absolute numbers to percentages. A perfect score would reveal the same frequency for each language. Table 1 shows the observations, Fig. 6 visualizes them.

Clearly, subjects perform better on both tasks under the treatment MOCQL than under the treatment OCL. In fact, several subjects complained about OCL in follow-up interviews or comments on the questionnaire margins. In the second experiment, we see that subjects perform better using MOCQL than VMQL with what appears to be a smaller margin. Observe that there is a variation in the scores for MOCQL between the two experiments, which we explain by variations in the subject populations, i.e., participants of Experiment 2 can be expected to have a far beyond average general intelligence. The relative difficulty between the two tasks is consistent across both experiments, further confirming the validity of our findings.

Let us now turn to the subjective assessments. Participants were asked to record their subjective assessment on a 5-point Likert scale which we normed to the interval 0..10 for easier presentation. Since these are subjective measures anyway, we combined the results from both experiments in this presentation. Table 2 shows the observations, Fig. 7 visualizes them.

Table 1. Performance of subjects in tasks B and C

Language	Experiment 1 Task B	Experiment 1 Task C	Experiment 2 Task B	Experiment 2 Task C
MOCQL	82.1%	58.7%	83.9%	62.7%
VMQL	-	-	74.2%	49.0%
OCL	54.8%	38.1%	-	-

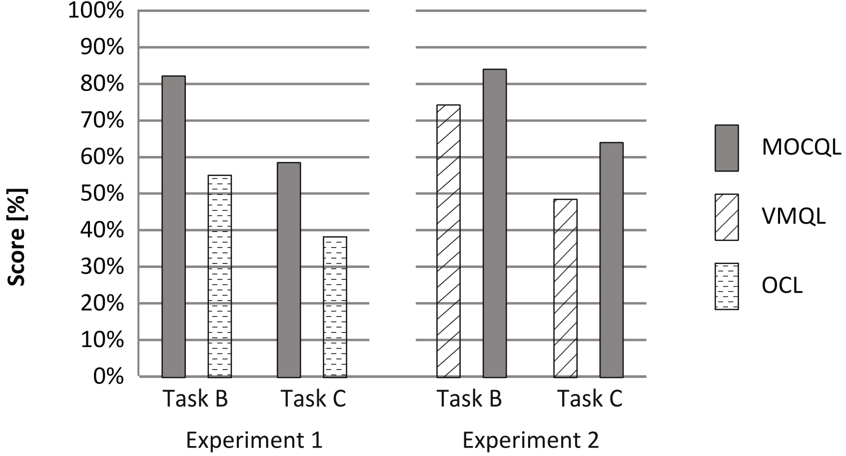


Fig. 6. Performance of subjects in tasks B and C: visualization of the data in Table 1

All three measures consistently show the same trend of OCL scoring lower than VMQL, which in turn scores lower than MOCQL. As before, there is a larger difference between OCL and VMQL, than there is between MOCQL and VMQL. We see that the ratings for "Understandability" and "Confidence" are particularly low for OCL, which is consistent with the post-experimental remarks by participants.

5.3 Validity

With 34 participants, three different tasks, and 28/12/3 different measurements within each task, we have a fairly large sample size. Due to the study design, we can safely exclude bias through learning effects or variations in the subject population. All results are consistent with each other, with only the minor fluctuations between experiments that are to be expected in any kind of human factor study.

Obviously, the task presentation would influence the outcome: since MOCQL tries to imitate a natural language in its concrete syntax, there is high degree of proximity to the task description, that is provided in written English, biasing the result in favor of MOCQL. However, we stipulate that describing a query in plain English is exactly what a modeler does when faced with a search task.

Table 2. Subjective assessment of cognitive load

Language	Understandability		Effort		Confidence	
	μ	σ	μ	σ	μ	σ
MOCQL	8.0	1.88	6.2	3.38	8.5	2.20
VMQL	7.0	2.28	7.5	3.30	7.7	2.90
OCL	3.8	1.80	8.8	1.78	3.3	1.55

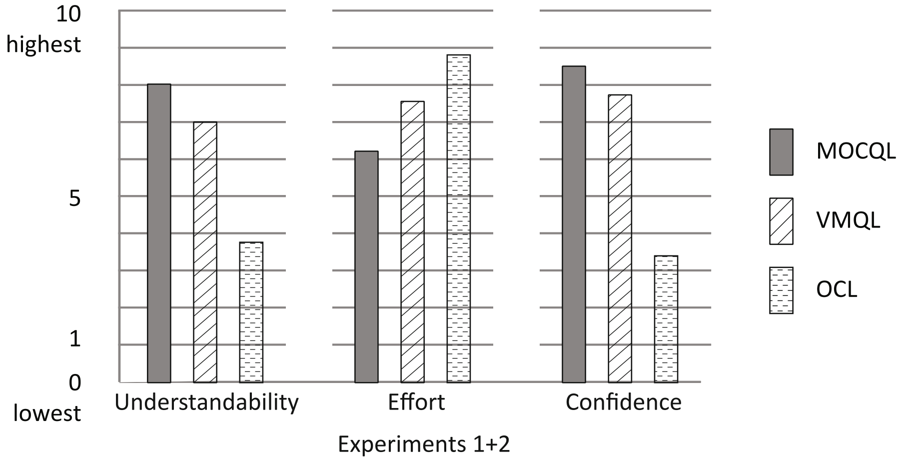


Fig. 7. Subjective assessment of cognitive load (averages across all subjects, normalized to the interval 0 (lowest) to 10 (highest)

Allowing him or her to express queries that way is, thus, not an undue influence on the experiment. Quite contrary, that is exactly the point of MOCQL.

A potential threat to validity is the fact that we did not test the respective whole languages, that is, there are parts of OCL, VMQL, and MOCQL that have not been subjected to experimental validation of their usability. In that sense, the validity of inferences regarding the languages as such is limited. Due to the conceptual differences between the languages, however, it would be difficult to completely compare them.

One might also object that the subjects—students—are not representative for the audience MOCQL is targeted at, i.e., people with little or no UML knowledge. In fact, the participants of Experiment 2 have been tested after they had just completed a one-term intensive course on UML, MDA, and OCL. However, even that degree of UML/OCL knowledge and additional cues and auxiliary high-level functions did not lead to an improved performance on the OCL tasks.

Finally, we have used different measurements to capture aspects of cognitive load (cf. [11]), yielding consistent results. Subjective assessments of cognitive load have been found to be very reliable indicators of the objective difficulty of a task (cf. [8]).

Table 3. Testing Hypothesis with the Binomial test

Hypothesis	Task	Significance
Experiment 1:	B	$p < 10^{-7}$ ***
Subjects perform better using OCL than MOCQL	C	$p < 10^{-5}$ ***
Experiment 2	B	$p = 0.0033$ *
Subjects perform better using VMQL than MOCQL	C	$p = 0.059$ ·

5.4 Inferences

We tested the hypothesis that subjects performed better using OCL than MOCQL in Experiment 1. Using a binomial test in the R environment [12], we can reject this hypothesis with high certainty ($p < 10^{-7}$ for task B, and $p < 10^{-5}$ for task C). Similarly, we can reject the hypothesis that subjects performed better using VMQL than MOCQL in Experiment 2, though there is not necessarily a significant result for task C ($p = 0.0033$ for task B, and $p = 0.059$ for task C).

5.5 Interpretation and Conclusions

It is obvious that OCL offers little to modelers when it comes to querying models, and our investigation establishes the consequences as a fact. In previous research [17], we have shown how users performs better using VMQL than OCL, over a range of tasks. The current results show that user perform better using MOCQL than both OCL and VMQL. Both in our previous research and the current results, users also show a much higher acceptance (and thus, motivation), for MOCQL and VMQL than they exhibit for OCL, consistently across different task types, many different queries, and different measurements, which all consistently point in the same direction. Our results are mostly significant, some of them to the extreme. Our study exhibits a high degree of validity.

We have thus provided substantial evidence in support of our initial working hypothesis that MOCQL offers better usability than both OCL and VMQL, as outlined in Section 1.2. We believe it is safe to assume, that these results are generalizable to other contexts, such as different subject populations or different queries. Also, we expect these findings to carry over to extensions of MOCQL that have not yet been tested.

6 Related Work

There are essentially three kinds of query facilities. First, there are basic tools like full text search and sets of predefined queries. These sacrifice expressiveness for usability, leaving modelers with little leverage. On the other end of the spectrum, there are application programming interfaces of modeling tools, which offer maximum expressiveness to the modeler, but require substantial expertise which only few modelers possess. Certainly, domain experts, which are in the focal point of our work, lack this capability.

Between these two extremes, there are model query languages varying along different dimensions. On the one hand, there is of course OCL [10] as the most widely used model query language. OCL also seems to be the only generically applicable textual model query language (disregarding non-semantic facilities such SQL, XPath, and similar). As our studies clearly show, OCL is not suited for ad-hoc querying by domain experts. In fact, even highly trained professionals and top-notch students with substantial training in OCL have serious trouble using it.

On the other hand, there are the visual model query languages like QM [14,13], BP-QL [3], BPMQ [2,1], Visual OCL [4,5], and VMQL [15,17,7] (see [17] for a detailed comparison). These come with the explicit or implicit promise of higher usability, exploiting the fact that most modeling notations are also visual, and it is intuitively appealing to express queries the same way as base models. However, little evidence has been published to support this intuition; only VMQL seems to have been evaluated from this angle. From the results presented above and in previous studies, respectively, it is clear that OCL performs poorly, and that both MOCQL and VMQL perform better than OCL. Surprisingly, though, MOCQL even surpasses VMQL with respect to usability. This contradicts the common intuition about textual vs. visual notations and demands further inquiry.

We expect Visual OCL to perform similar to OCL since it is just a visualization of OCL; the other visual model query languages should yield results similar to those of VMQL since they are based on a similar paradigm and in some cases offer similar solutions (e.g., the treatment of transitive edges in BP-QL and QM, or negation in BP-QL).

Most model query languages are restricted to express queries on a single notation or a small set of related notations. For instance BPMN-Q addresses BPMN and (to some degree) EPCs, QM address a subset of UML class and sequence diagrams, and CM address only elementary class diagrams. On the other hand, OCL and Visual OCL apply to all MOF-based notations; VMQL and MOCQL even go beyond that requirement.

There are large differences with respect to the tool support a modeler might obtain for the model query languages mentioned. Only for OCL is there a choice of quality tools from different sources. Most of the other tools have been implemented as academic prototypes only, or not even that (e.g., CD and QM).

OCL (and, potentially, Visual OCL) offer maximum expressiveness through defining recursive functions. Most other model query languages mentioned above seem to have been analyzed from this perspective. VMQL does do not offer user-defined recursive functions, and is thus less expressive than OCL, though the exact degree of expressiveness is currently unknown. Similarly, MOCQL does not allow the definition of recursive functions, but it should be not too difficult to add such a feature. Observe also, that MOCQL provides features that are relevant for practical model querying, but currently missing in OCL, such as using wild-card expressions, executing queries across several models, type variables, or access to model element identifiers.

7 Discussion

7.1 Summary

In this paper we have introduced the Model Query and Constraint Language (MOCQL), by means of example and a (simplified) grammar. We report on user studies comparing OCL, VMQL, and MOCQL, and find strong evidence that MOCQL offers higher usability than both OCL and VMQL in a number of ways. This is particularly true when comparing MOCQL and OCL. At the same time, MOCQL offers a high degree of expressiveness. MOCQL can be applied to the whole range of modeling notations present in the UML, not just structural models or meta-models. In fact, MOCQL is not conceptually restricted to UML: we believe it is applicable to any modeling language that has a meta-model or where a meta-model can be constructed, including BPMN, EPCs, and DSLs.

7.2 Contributions

The contribution of this paper is to provide evidence for two observations. Firstly, we maintain that usability is an important concern when it comes to model query languages, but has been largely ignored in existing languages, most notably OCL. Thus, it is relatively easy at this point to achieve substantial improvements over the state of the art. Secondly, it is not so much the concrete syntax that contributes to usability, but the abstract syntax, that is, the conceptual constructs of the query language. In this paper, we show that a textual concrete syntax can actually perform better than a visual concrete syntax, which is somewhat in contradiction with the commonly held belief of visual notations generally being "better" than textual ones.

7.3 Limitations

In its current state, MOCQL has several shortcomings. Firstly, it lacks a formal semantics. Given the time and difficulty it took to arrive at a formal semantics for OCL, we consider this more of a challenge and future work than a lasting deficit.

Secondly, MOCQL currently lacks the capability to define recursive functions, and thus complete expressiveness. MOCQL was designed with the practical modeler in mind, thus, many of the functions that modelers have to define themselves in OCL are built into MOCQL, thus reducing the need for such a feature.

Thirdly, MOCQL allows many expressions that are either hard to process, or may be confusing. For instance, MOCQL allows to express queries with double negation. Clearly, this is computationally inefficient, and since we use the regular negation-as-failure semantics of Prolog, the result might not be what the user expects. Moreover, since double negation is inherently cognitively difficult, using it will be a challenge. We currently lack empirical evidence on the actual usage of MOCQL in the field.

7.4 Future Work

Clearly, the current limitations of MOCQL are some threads of our ongoing and future work. In particular, it would be interesting to apply MOCQL to other modeling languages, such as BPMN and EPCs and see whether the current MOCQL is up to this task, or requires extensions and amendments. Also, parts of the implementation would have to be adapted to accommodate for different model representations.

Then, performance is obviously an issue for practical model querying, in particular for using MOCQL for interactive operations on large models. We generally have very good experience with the performance of the technology underlying our approach in comparison with current OCL implementations (see also [6]), and experience so far indicate that MOCQL might in fact be dramatically faster than existing OCL tools. Still, we will have study and document the run-time performance of MOCQL.

Moreover, our initial research hypothesis is based on the intuition, that the major improvement in usability would derive from using a visual rather than a textual concrete syntax for querying. Thus, one would expect a similar effect for, say, the Visual OCL [4,5]. Doing pairwise comparisons of OCL, Visual OCL, VMQL, and MOCQL, respectively, and studying the factors impacting modeler understanding with qualitative methods such as think aloud protocols might allow us to develop a theory about how queries are being processed by modelers. This, in turn, could be valuable in informing future language design practice.

References

1. Awad, A.: A Compliance Management Framework for Business Process Models. PhD thesis, Hasso Plattner Institute, Univ. of Potsdam (2010)
2. Awad, A., Decker, G., Weske, M.: Efficient Compliance Checking Using BPMN-Q and Temporal Logic. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 326–341. Springer, Heidelberg (2008)
3. Beeri, C., Eyal, A., Kamenkovich, S., Milo, T.: Querying Business Processes. In: Proc. 32nd Intl. Conf. Very Large Data Bases (VLDB), pp. 343–354. VLDB Endowment (2006)
4. Bottoni, P., Koch, M., Parisi-Presicce, F., Taentzer, G.: Consistency Checking and Visualization of OCL Constraints. In: Evans, A., Caskurlu, B., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 294–308. Springer, Heidelberg (2000)
5. Bottoni, P., Koch, M., Parisi-Presicce, F., Taentzer, G.: A Visualisation of OCL using Collaborations. In: Gogolla, M., Kobryn, C. (eds.) UML 2001. LNCS, vol. 2185, pp. 257–271. Springer, Heidelberg (2001)
6. Chimiak-Opoka, J., Felderer, M., Lenz, C., Lange, C.: Querying UML Models using OCL and Prolog: A Performance Study. In: Faivre, A., Ghosh, S., Pretschner, A. (eds.) Ws. Model Driven Engineering, Verification, and Validation (MoDeVVa 2008), pp. 81–89 (2008)
7. Costagliola, G., et al. (eds.): Expressing Model Constraints Visually with VMQL. IEEE Computer Society (2011)
8. Gopher, D., Braune, R.: On the Psychophysics of Workload: Why Bother with Subjective Measures? Human Factors 26(5), 519–532 (1984)

9. OMG. OMG Unified Modeling Language (OMG UML), Superstructure, V2.4 (ptc/2010-12-06). Technical report, Object Management Group (January 2011)
10. OMG. OCL Specification v2.3.1 (formal/2012-01-01). Technical report, Object Management Group (January 2012)
11. Paas, F., Tuovinen, J.E., Tabbers, H., Van Gerven, P.W.M.: Cognitive Load Measurement as a Means to Advance Cognitive Load Theory. Educational Psychologist 38(1), 63–71 (2003)
12. R Development Core Team. R: A Language and Environment for Statistical Computing. R Foundation for Statistical Computing, Vienna, Austria (2011)
13. Stein, D., Hanenberg, S., Unland, R.: A Graphical Notation to Specify Model Queries for MDA Transformations on UML Models. In: Aßmann, U., Akşit, M., Rensink, A. (eds.) MDAFA 2003. LNCS, vol. 3599, pp. 77–92. Springer, Heidelberg (2005), available at `www.ida.liu.se/~henla/mdafa2004`
14. Stein, D., Hanenberg, S., Unland, R.: Query Models. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) UML 2004. LNCS, vol. 3273, pp. 98–112. Springer, Heidelberg (2004)
15. Störrle, H.: VMQL: A Generic Visual Model Query Language. In: Erwig, M., DeLine, R., Minas, M. (eds.) Proc. IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2009), pp. 199–206. IEEE Computer Society (2009)
16. Störrle, H.: Towards Clone Detection in UML Domain Models. J. Software and Systems Modeling (2011) (in print)
17. Störrle, H.: VMQL: A Visual Language for Ad-Hoc Model Querying. J. Visual Languages and Computing 22(1) (February 2011)
18. Störrle, H.: Making Sense to Modelers - Presenting UML Class Model Differences in Prose. In: Filipe, J., das Neves, R.C., Hammoudi, S., Pires, L.F. (eds.) Proc. 1st Intl. Conf. Model-Driven Engineering and Software Development, pp. 39–48. SCITEPRESS (2013)

Supporting Different Process Views through a Shared Process Model

Jochen Küster[1], Hagen Völzer[1], Cédric Favre[1], Moises Castelo Branco[2], and Krzysztof Czarnecki[2]

[1] IBM Research — Zurich, Switzerland
[2] Generative Software Development Laboratory, University of Waterloo, Canada

Abstract. Different stakeholders in the Business Process Management (BPM) life cycle benefit from having different views onto a particular process model. Each view can show, and offer to change, the details relevant to the particular stakeholder, leaving out the irrelevant ones. However, introducing different views on a process model entails the problem to synchronize changes in case that one view evolves. This problem is especially relevant and challenging for views at different abstraction levels. In this paper, we propose a *Shared Process Model* that provides different stakeholder views at different abstraction levels and that synchronizes changes made to any view. We present detailed requirements and a solution design for the Shared Process Model in this paper. Moreover, we also present an overview of our prototypical implementation to demonstrate the feasibility of the approach.

1 Introduction

A central point in the value proposition of BPM suites is that a business process model can be used by different stakeholders for different purposes in the BPM life cycle. It can be used by a business analyst to document, analyze or communicate a process. Technical architects and developers can use a process model to implement the business process on a particular process engine. These are perhaps the two most prominent uses of a process model, but a process model can also be used by a business analyst to visualize monitoring data from the live system, or by an end user of the system, i.e., a process participant, to understand the context of his or her participation in the process.

These different stakeholders would ideally share a single process model to collaborate and to communicate to each other their interests regarding a particular business process. For example, a business analyst and a technical architect could negotiate process changes through the shared model. The business analyst could initiate process changes motivated by new business requirements, which can then be immediately seen by the technical architect and forms the basis for him to evaluate and implement the necessary changes to the IT system. The technical architect may revise the change because it is not implementable in the proposed form on the existing architecture. Vice versa, a technical architect can also initiate and communicate process changes motivated from technical requirements, e.g., new security regulations, revised performance requirements, etc. In this way, a truly shared process model can increase the agility of the enterprise.

P. Van Gorp, T. Ritter, and L.M. Rose (Eds.): ECMFA 2013, LNCS 7949, pp. 20–36, 2013.
© Springer-Verlag Berlin Heidelberg 2013

This appealing vision of a single process model that is shared between stakeholders is difficult to achieve in practice. One practical problem is that, in some enterprises, different stakeholders use different metamodels and/or different tools to maintain their version of the process model. This problem makes it technically difficult to conceptually share 'the' process model between the stakeholders (the BPMN -BPEL *roundtripping problem* is a known example). This technical problem disappears with modern BPM suites and the introduction of BPMN 2, as this single notation now supports modeling both business and IT-level concerns.

However, there is also an essential conceptual problem. We argue that different stakeholders intrinsically want different *views* onto the same process because of their different concerns and their different levels of abstraction. This is even true for parts that all stakeholders are interested in, e.g., the main behavior of the process. Therefore we argue that we need separate, stakeholder-specific views of the process that are kept *consistent* with respect to each other. Current tools do not address this problem. Either different stakeholders use different models of the same process, which then quickly become inconsistent, or they use the same process model, which then cannot reflect the needs of all stakeholders.

This problem is a variation of the coupled evolution problem [11] and the model synchronization problem [10]. Coupled evolution has been studied between metamodels and models but not for process models at different abstraction levels and in the area of model synchronization various techniques have been proposed. Put into this context, our research question is how process views at different abstraction levels can be kept consistent and changes can be propagated in both directions automatically in a way that is aligned with existing studies of requirements from practice. In this paper, we address this problem, present detailed requirements and a design to synchronize process views on different abstraction levels. The challenge for a solution arises from an interplay of a variety of possible interdependent process model changes and their translation between the abstraction levels. We also report on an implementation to substantiate that a solution is indeed technically feasible. An extended version of this paper is available as a technical report [14].

2 The Business-IT Gap Problem

In this section, we motivate our Shared Process Model concept. First we argue why we think that a single process model view is often not adequate for different stakeholders and we discuss how different views differ. We illustrate this issue by example of two prominent stakeholder views of a process: the business analysts view used for documentation, analysis and communicating requirements to IT and the IT view of a process that is used directly for execution. Then, we briefly argue that, with multiple views, we need a dedicated effort to keep them consistent.

2.1 Why We Want Different Views

Since BPMN 2 can be used for both documentation and execution, why can't we use a single BPMN 2 model that is shared between business and IT? To study this question, we

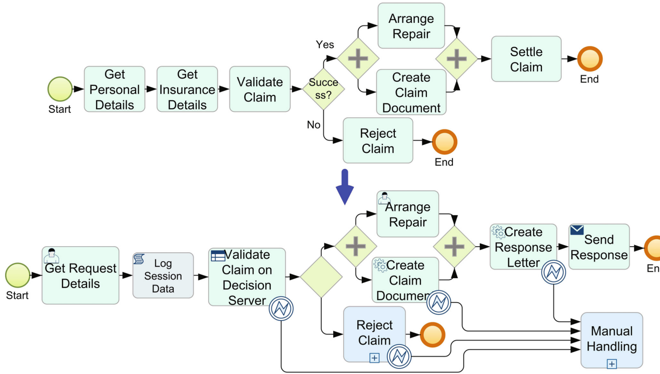


Fig. 1. Illustration of some refinements often made going from the business to the IT model

analyzed the range of differences between a process model created by a business analyst and the corresponding process model that was finally used to drive the execution on a BPM execution engine. We built on our earlier study *et al.* [2], which analyzed more than 70 model pairs from the financial domain, and we also investigated additional model pairs from other domains. Additionally we talked to BPM architects from companies using process models to collect further differences. We summarize our findings here.

We identified the following categories of changes that were applied in producing an execution model from a business model. Fig. 1 illustrates some of these changes in a simplified claim handling process. We refer to our earlier study by Branco *et al.* [2] for a larger, more realistic example. Note that the following categorization of changes, based on a larger study, is a new contribution of this paper.

- *Complementary implementation detail.* Detail that is needed for execution is merely added to the business model, i.e., the part of the model that was specified by the business analyst does not change. Such details include data flow and -transformation, service interfaces and communication detail. For example, to specify the data input for an activity in BPMN 2, one sets a specific attribute of the activity that was previously undefined. The activity itself, its containment in a subprocess hierarchy and its connection with sequence flow does not change.
- *Formalization and renaming.* Some parts of the model need to be formalized further to be interpreted by an execution engine, including routing conditions, specialization of tasks (into service task, human task etc., see Fig. 1) and typing subprocesses (transaction, call) and typing events. Furthermore, activities are sometimes renamed by IT to better reflect some technical aspects of the activity. These are local, non-structural changes to existing model elements that do not alter the flow.
- *Behavioral refinement and refactoring.* The flow of the process is changed in a way that does not essentially change the behavior. This includes
  - *Hierarchical refinement/subsumption.* A high-level activity is refined into a sequence of low-level activities or more generally, into a subprocess with the same input/output behavior. For example, ‘Settle Claim’ in Fig. 1 is refined into ‘Create Response Letter’ and ‘Send Response’. The refining subprocess may or may not be explicitly enclosed in a separate scope (subprocess or call activity). If it is not enclosed in a separate scope, it

is represented as a subgraph which has, in most cases, a single entry and a single exit of sequence flow. We call such a subgraph a *fragment* in this paper.
On the other hand, multiple tasks on the business level may be subsumed in a single service call or a single human task to map the required business steps to the existing services and sub-engines (human task, business rules). For example, in Fig. 1, 'Get Personal Details' and 'Get Insurance Details' got subsumed into a single call 'Get Request Details' of the human task engine.
  - *Hierarchical refactoring.* Existing process parts are separated into a subprocess or call activity or they may be outsourced into a separate process that is called by a message or event. Besides better readability and reuse, there are several other IT-architectural reasons motivating such changes. For example, performance, dependability and security requirements may require executing certain process parts in a separate environment. In particular, long-running processes are often significantly refactored under performance constraints. A long-running process creates more load on the engine than a short running process because each change need to be persisted. Therefore, short-running parts of long-running process are extracted to make the long-running process leaner.
  - *Task removal and addition.* Sometimes, a business task is not implemented on the BPM engine. It may be not subject to the automation or it may already be partly automated outside the BPM system. On the other hand, some tasks are added on the IT level, that are not considered to be a part of an implementation of a specific business task. For example, a script task retrieving, transforming or persisting data or a task that is merely used for debugging purposes (e.g. 'Log Session Data' in Fig. 1).
- *Additional behavior.* Business-level process models are often incomplete in the sense that they do not specify all possible behavior. Apart from exceptions on the business-level that may have been forgotten, there are usually many technical exceptions that may occur that require error handling or compensation. This error handling creates additional behavior on the process execution level. In Fig. 1, some fault handling has been added to the IT model to catch failing service calls.
- *Correction and revision of the flow.* Some business-level process models would not pass syntactical and semantical validation checks on the engine. They may contain modeling errors in the control- or data flow that need to be corrected before execution. Sometimes activties also need to be reordered to take previously unconsidered data and service dependencies into account. These changes generally alter the behavior of the process. A special case is the possible parallelization of activities through IT, which may or may not be considered a behavioral change.

Different changes that occur in the IT implementation phase relate differently to the shared process model idea. Complementary detail could be easily handled by a single model through a *progressive disclosure* of the process model, i.e., showing one graphical layer to business and two layers to IT stakeholders.

However, the decision which model elements are 'business relevant' depends on the project and should not be statically fixed (as in the BPMN 2 conformance classes). Therefore, an implementation of progressive disclosure requires extensions that specify which element belongs to which layer. Additional behavior can be handled through progressive disclosure in a similar way as long as there are no dependencies to the business layer. For example, according to the BPMN 2 metamodel, if we add an error boundary event to a task with subsequent sequence flow specifying the error handling, then this creates no syntactical dependencies from the business elements to this addition. However, if we merge the error handling back to the normal flow through a new gateway or if we

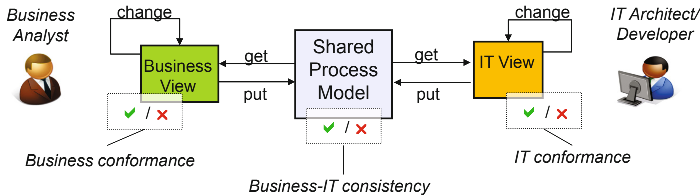


Fig. 2. Process view synchronization via a Shared Process Model

branch off the additional behavior by a new gateway in the first place, then the business elements need to be changed, which would substantially complicate any implementation of a progressive disclosure. In this case, it would be easier to maintain two separate views. Also the changes in the categories *behavioral refinement and refactoring* as well as *formalization and renaming* clearly suggest to maintain two separate views.

Why different views need to be synchronized. In fact, many organizations today keep multiple versions of a process model to reflect the different views of the stakeholder (cf., e.g., [2,18,19]). However, because today's tools do not have any support for synchronizing them, they typically become inconsistent over time. That is, they disagree about which business tasks are executed and in which order. This can lead to costly business disruptions or to audit failures [2]. In the technical report version of this paper [14], we elaborate this point in more detail.

3 Requirements for a Shared Process Model

3.1 The Shared Process Model Concept

The *Shared Process Model*, which we now present, has the capability to synchronize process model views that reside on different abstraction levels. The concept is illustrated by Fig. 2. The Shared Process Model provides two different views, a business view and an IT view, and maintains the consistency between them. A current view can be obtained at any time by the corresponding stakeholder by the 'get' operation. A view may also be changed by the corresponding stakeholder. With a 'put' operation, the changed view can be checked into the Shared Proess Model, which synchronizes the changed view with the other view.

Each view change can be either designated as a *public* or a *private* change. A *public* change is a change that needs to be reflected in the other view whereas a *private* change is one that does not need to be reflected. For example, if an IT architect realizes, while he is working on the refinement of the IT model, that the model is missing an important business activity, he can insert that activity in the IT model. He can then check the change into the Shared Process Model, designating it as a public change to express that the activity should be inserted in the business view as well. The Shared Process Model then inserts the new activity in the business view automatically at the right position, i.e., every new business view henceforth obtained from the Shared Process Model will contain the new activity. If the IT architect designated the activity insertion as a private

change, then the business view will not be updated and the new activity will henceforth be treated by the Shared Process Model as an 'IT-only' activity.

Fig. 2 also illustrates the main three status conditions of a Shared Process Model: *business conformance, IT conformance and Business-IT consistency*. The business view is *business conformant* if it is approved by the business analyst, i.e., if it reflects the business requirements. This should include that the business view passes basic validity checks of the business modeling tool. The IT view is *IT conformant* if it is approved by the IT architect, i.e., if it meets the IT requirements. This should include that the IT view passes all validity checks of the IT modeling tool and the execution engine. *Business-IT consistency* means that the business view faithfully reflects the IT view, or equivalently, that the IT model faithfully implements the business view.

In the remainder of this section, we discuss the requirements and capabilities of the Shared Process Model in more detail.

3.2 Usage Scenarios and Requirements

We distinguish the following usage scenarios for the Shared Process Model. In the *presentation* scenario, either the business or IT stakeholder can, at any time, obtain a current state of his view with the 'get' operation. The view must reflect all previous updates, which may have been caused by either stakeholder.

The Shared Process Model is *initialized* with a single process model (the initial business view), i.e., business and IT views are initially identical. Henceforth, both views may evolve differently through *view change scenarios*, which are discussed below. For simplicity, we assume here that changes to different views do not happen concurrently. Concurrent updates can be handled on top of the Shared Process Model using known concurrency control techniques. That is, either a pessimistic approach is chosen and a locking mechanism prevents concurrent updates, which, we believe, is sufficient in most situations. Or an optimistic approach is chosen and different updates to the Shared Model may occur concurrently—but atomically, i.e., each update creates a separate new consistent version of the Shared Model. Parallel versions of the Shared Model must then be reconciled through a horizontal compare/merge technique on the Shared Model. Such a horizontal technique would be orthogonal to the vertical synchronization we consider here and out of scope of this paper.

In the *view change* scenario, one view is changed by a stakeholder and checked into the Shared Process Model with the 'put' operation to update the other view. A view change may contain many separate individual changes such as insertions, deletions, mutations or rearrangement of modeling elements. Each individual change must be designated as either private or public. We envision that often a new view is checked into the Shared Process Model which contains either only private or only public individual changes. These special cases simplify the designation of the changes. For example, during the initial IT implementation phase, most changes are private IT changes.

A private change only takes effect in one view while the other remains unchanged. Any public change on one view must be propagated to the other view in an automated way. We describe in more detail in Sect. 4, in what way a particular public change in one view is supposed to affect the other view. An appropriate translation of the change

is needed in general. User intervention should only be requested when absolutely necessary for disambiguation in the translation process. We will present an example of such a case in Sect. 4.

The designation of whether a change is private or public is in principle a deliberate choice of the stakeholder that changes his view. However, we imagine that governance rules are implemented that disallow certain changes to be private. For example, private changes should not introduce inconsistencies between the views, e.g., IT should not change the order of two tasks and hide that as a private change. Therefore, the business-IT consistency status need to be checked upon such changes.

The key function of the Shared Process Model is to maintain the consistency between business and IT view. Business-IT consistency can be thought of as a Boolean condition (consistent or inconsistent) or a measure representing a degree of inconsistency. According to our earlier study [2], the most important aspect is *coverage*, which means that (i) every element (e.g. activities and events) in the business view should be implemented by the IT view, and (ii) only the elements in the business view are implemented by the IT view.

The second important aspect of business-IT consistency is *preservation of behavior*. The activities and events should be executed in the order specified by the business view. The concrete selection of a consistency notion and its enforcement policy should be configurable on a per-project basis. A concrete notion should be defined in a way that users can easily understand, to make it as easy as possible for them to fix consistency violations. Common IT refinements as discussed in Sect. 2.1 should be compatible with the consistency notion, i.e., should not introduce inconsistencies, wheras changes that cannot be considered refinements should create consistency violations. Checking consistency should be efficient in order to be able to detect violations immediately after a change.

On top of the previous scenarios, support for change management is desirable to facilitate collaboration between different stakeholders through the Shared Process Model. The change management should support approving or rejecting public changes. In particular, public changes made by IT should be subject to approval by business. Only a subset of the proposed public changes may be approved. The tool supporting the approval of individual changes should make sure that the set of approved changes that is finally applied to the Shared Process Model leads to a valid model. The Shared Process Model should be updated automatically to reflect only the approved changes. The change management requires that one party can see all the changes done by the other party in a consumable way. In particular, it should be possible for an IT stakeholder to understand the necessary implementation steps that arise from a business view change.

If a process is in production, all three conditions, business conformance, IT conformance and business-IT consistency, should be met. Upon a public change of the IT view, the business view changes and hence the Shared Process Model must show that the current business view is not approved. Conversely, a public change on the business view changes the IT view and the Shared Process Model must indicate that the current IT view is not approved by IT. Note that a change of the IT view that was induced by a public change of the business view is likely to affect the validity of the IT view with respect to executability on a BPM engine.

4 A Technical Realization of the Shared Process Model

In this section, we present parts of a technical realization of the concepts and requirements from Sect. 3 as we have designed and implemented them.

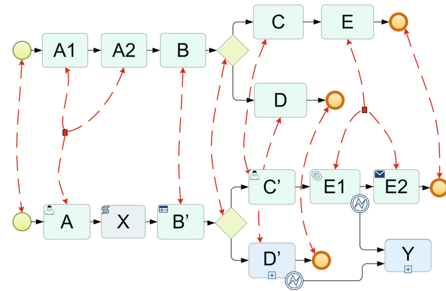


Fig. 3. The Shared Process Model as a combination of two individual models, coupled by correspondences

4.1 Basic Solution Design

We represent the Shared Process Model by maintaining two process models, one for each view, together with *correspondences* between their model elements, as illustrated by Fig. 3. In the upper part, the process model for business is shown, in the lower part the process model for IT. A *correspondence*, shown by red dashed lines, is a bidirectional relation between one or more elements of one model and one or more elements of the other model.

For example, in Fig. 3, task B of the business layer corresponds to task B' of the IT layer which is an example for a one-to-one correspondence. Similarly, task D of the business layer corresponds to subprocess D' of the IT layer and tasks A_1 and A_2 correspond to the (human) task A of the IT layer which is an example for a many-to-one correspondence. Many-to-many correspondences are technically possible but we haven't found a need for them so far. We only relate the main flow elements of the model, i.e., activities, events and gateways, but sequence flow is not linked. Each element is contained in at most one correspondence. An element that is contained in a correspondence is called a *shared* element, otherwise it is a *private* element.

Alternatively, we could have chosen to represent the Shared Process Model differently by merging the business and IT views into one common model with overlapping parts being represented only once. This ultimately results in an equivalent representation, but we felt that we stay more flexible with our decision above in order to be able to easily adapt the precise relationship between business and IT views during further development.

Furthermore, with our realization of the Shared Process Model we can easily support the following:

- Import/export to/from the Shared Process Model: From the Shared Process Model, a process model must be created (e.g. business view) that can be shown by an editor. This is straight-forward in our representation. We use BPMN 2 internally in the Shared Process Model, which can be easily consumed outside by existing editors. Likewise, other tools working on BPMN 2 can be leveraged for the Shared Process Model prototype easily.
- Generalization to a Shared Process Model with more than two process models is easier to realize with correspondences rather than with a merged metamodel. This

includes generalization to three or more stakeholder views, but also when one business model is implemented by a composition of multiple models (see Sect. 2.1) or when a business model should be traced to multiple alternative implementations.

The technical challenges occur in our realization if one of the process models evolves under changes because then the other process model and the correspondences have to be updated in an appropriate way.

4.2 Establishing and Maintaining Correspondences

A possible initialization of the Shared Process Model is with a single process model, which can be thought of the initial business view. This model is then internally duplicated to serve as initially identical business and IT models. This creates one-to-one correspondences between all main elements of the process models for business and IT. The creation of such correspondences is completely automatic because in this case a correspondence is created between elements with the same universal identifier during the duplication process. Another possible initialization is with a pair of initial business and IT views where the two views are not identical, e.g. they might be taken from an existing project situation where the processes at different abstraction levels already exist. In such a case, the user would need to specify the correspondences manually or process matching techniques can be applied to achieve a higher degree of automation [1].

A one-to-many or many-to-one correspondence can be introduced through an editing wizard. For example, if an IT architect decides that one business activity is implemented by a series of IT activities, he uses a dedicated wizard to specify this refinement. The wizard forces the user to specify which activity is replaced with which set of activities, hence the wizard can establish the one-to-many correspondence.

The Shared Model evolves either through such wizards, in which case the wizard takes care of the correspondences, or through free-hand editing operations, such as deletion and insertion of tasks. When such changes are checked into the Shared Model as public changes, the correspondences need to be updated accordingly. For example, if an IT architect introduces several new activities that are business-relevant and therefore designated as public changes, the propagation to the business level must also include the introduction of new one-to-one correspondences. Similarly, if an IT architect deletes a shared element on the IT level, a correspondence connected to this shared element must be removed when propagating this change.

4.3 Business-IT Consistency

As described in Sect. 3.2, we distinguish coverage and preservation of behavior. Coverage can be easily checked by help of the correspondences. Every private element, i.e., every element that is not contained in a correspondence must be accounted for. For example, all private business tasks, if any, could be marked once by the business analyst and all private IT tasks by the IT architect. The Shared Process Model then remembers these designations. A governance rule implemented on top may restrict who can do these designations. All private tasks that are not accounted for violate coverage.

For preservation of behavior, we distinguish *strong and weak consistency* according to the IT refinement patterns discussed in Sect. 2.1. If business and IT views are strongly consistent, then they generate the same behavior. If they are weakly consistent, then every behavior of the IT view is a behavior of the business view, but the IT view may have additional behavior, for example, to capture additional exceptional behavior. As with coverage, additional behavior in the IT view should be explicitly reviewed to check that it is indeed considered technical exception behavior and not-considered 'business-relevant'.

We use the following concretizations of strong and weak consistency here. At this stage, we only consider behavior generated by the abstract control flow, i.e., we do not yet take into account how data influences behavior.

- We define the Shared Process Model to be *strongly consistent* if the IT view can be derived from the business view by applying only operations from the first three categories in Sect. 2.1: complementary implementation detail, formalization and renaming, and behavioral refinement and refactoring. Private tasks in either view are compatible with consistency only if they are connected to shared elements by a path of sequence flow. The operations from the first three categories all preserve the behavior. The Shared Process Model in Fig. 3 is not strongly consistent because the IT view contains private boundary events. Without the boundary events and without activity Y, the model would be strongly consistent. Fig. 4 shows examples for violating strong consistency.
  An initial Shared Process Model with two identical views is strongly consistent. To preserve strong consistency, all flow rearrangements on one view, i.e., moving activities, rearranging sequence flow or gateways must be propagated to the other view as public changes.
- For *weak consistency*, we currently additionally allow only IT-private error boundary events leading to IT private exception handling. Technically we could also allow additional IT-private gateways and additional branches on shared gateways here, but we haven't yet seen a strong need for them. The Shared Process Model in Fig. 3 is weakly consistent. The examples in Fig. 4 also violate weak consistency.

We have used the simplest notion(s) of consistency such that all the refinement patterns we have encountered so far can be dealt with. We haven't yet seen, within our usage scenarios, the need for more complex notions based on behavioral equivalences such as trace equivalence or bisimulation.

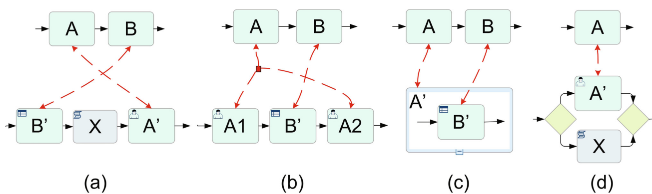


Fig. 4. Examples of inconsistencies

Strong and weak consistency can be efficiently checked but the necessary algorithms and also the formalization of these consistency notions are beyond the scope of this paper[1]. The automatic propagation of public changes, which we will describe in the following sections, rests on the Shared Process Model being at least weakly consistent.

4.4 Computing Changes between Process Models

If the Shared Process Model evolves by changes on the business or IT view, then such changes must be potentially propagated from one view to the other. As a basis for our technical realization of the Shared Process Model, an approach for *compare and merge* of process models is used [15]. We use these compound operations because they minimize the number of changes and represent changes on a higher level of abstraction. This is in contrast to other approaches for comparing and merging models which focus on computing changes on each model element.

Figure 5 shows the change operations that we use for computing changes. InsertActivity, DeleteActivity and MoveActivity insert, delete and move activities or other elements such as events and subprocesses. InsertFragment, DeleteFragment and MoveFragment is used for inserting, deleting and moving fragments which represent control structures. The computation of a *change script* consisting of such compound operations is based on comparing two process models and their Process Structure Trees. For more details of the comparison algorithm, the reader is referred to [15].

Change Operation op	Effects on Process Model V
InsertActivity(x, a, b)	Insertion of a new activity x between two succeeding elements a and b in process model V and reconnection of control flow.
DeleteActivity(x)	Deletion of activity x and reconnection of control flow.
MoveActivity(x, a, b)	Movement of activity x from its old position into its new position between two succeeding elements a and b in process model V and reconnection of control flow.
InsertFragment(f, a, b)	Insertion of a new fragment f between two succeeding elements a and b in process model V and reconnection of control flow.
MoveFragement(f, a, b)	Movement of a fragment f from its old position to its new position.
DeleteFragment(f, c, d)	Deletion of fragment f between c and d from process model V and reconnection of control flow.

Fig. 5. Change operations according to [15]

As an example for an evolution scenario of the Shared Process Model, consider Figure 6. The left hand side shows a part of the initial state of the Shared Process Model in our scenario, which contains a 2-to-1 correspondence and a private IT task. So, some IT refinements have been done already. Assume now, that during IT refinement, the IT

[1] For strong consistency, one has to essentially check that the correspondences define a *continuous* mapping between the graphs as known in graph theory.

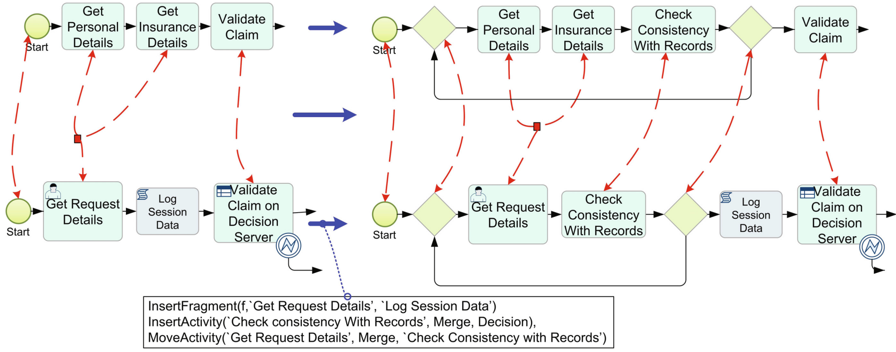


Fig. 6. Example of a change script on the IT level that is propagated to the business level

architect realizes that, in a similar process that he has implemented previously, there was an additional activity that checks the provided customer details against existing records. He is wondering why this is not done in this process and checks that with the business analyst, who in turn confirms that this was just forgotten. Consequently, the IT architect now adds this activity together with a new loop to the IT view, resulting in a new IT view shown in the lower right quadrant of Fig. 6. Upon checking this into the Shared Process Model as a public change, the business view should be automatically updated to the model shown in the upper right quadrant of Fig. 6.

To propagate the changes, one key step is to compute change operations between process models in order to obtain a *change script* as illustrated in Fig. 6. In the particular example, we compute three compound change operations: the insertion of a new empty fragment containing the two XOR gateways and the loop ($InsertFragment$), the insertion of a new activity ($InsertActivity$) and the move of an activity ($MoveActivity$), illustrated by the change script in Figure 6. In the next section, we explain how we use our approach to realize the evolution of the Shared Process Model.

4.5 Evolution of the Shared Process Model

For private changes, only the model in which the private changes occured is updated. In the following, we explain how public changes are propagated from IT to business, the case from business to IT is analogous.

When a new IT view is checked into the Shared Process Model, we first compute all changes between the old model IT and the new model IT', giving rise to a change script $Delta_{IT}$, see Figure 7 (a). The change script is expressed in terms of the change operations introduced above, i.e., $Delta_{IT} = \langle op_1, ..., op_n\rangle$ where each op_i is a change operation. In order to propagate the changes to the business level, $Delta_{IT}$ is translated into a change script $Delta_B$ for the business-level. This is done by translating each individual change operation op_i into an operation op_i^T and then applying it to the business-level. Likewise, we also apply each change operation on the IT-level to produce intermediate

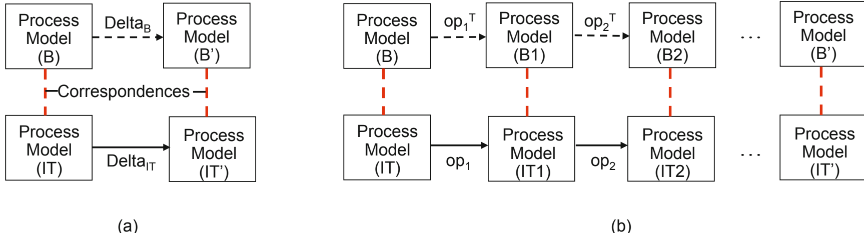


Fig. 7. Delta computation for propagating changes

process models for the IT level. Overall, we thereby achieve a synchronous evolution of the two process models, illustrated in Figure 7 (b).

Algorithm 1. Translation of a compound operation op from process model IT to Business model B

Step 1: compute corresponding parameters of the operation op
Step 2: replace parameters of op **with corresponding parameters to obtain** op^T
Step 3: apply op^T **to** B**, apply** op **to** IT
Step 4: update correspondences between B **and** IT

Algorithm 1 describes in pseudo-code the algorithm for translating a compound operation from IT to business. The algorithm for translation from business to IT can be obtained by swapping business and IT. Overall, one key step is replacing parameters of the operation from the IT model by parameters of the business model according to the correspondences. For example, for translating a change $InsertActivity(x,a,b)$, the parameters a and b are replaced according to their corresponding ones, following the correspondences in the Shared Process Model. In case that a and b are private elements, this replacement of elements requires forward/backward search in the IT model until one reaches the nearest shared element (Step 1 of the algorithm). Similarly, for translating an $InsertFragment(f,a,b)$, the parameters a and b are replaced in the same way. An operation $DeleteActivity(x)$ is translated into $DeleteActivity(x')$ (assuming here that x is related to x' by a one-to-one correspondence). After each translation, in Step 3 the change operation as well as the translated change operation are applied to produce new models B_i and IT_i, as illustrated in Figure 7 (b). Afterwards, Step 4 updates the correspondences between the business and IT model. For example, If x is the source or target of a one-to-many/many-to-one correspondence, then all elements connected to it must be removed.

For the example in Figure 6, the change script $Delta_{IT}$ is translated iteratively and applied as follows:

- The operation *InsertFragment(f, 'Get Request Details', 'Log Session Data')* is translated into *InsertFragment(f, 'Get Insurance Details', 'Validate Claim')*. The operation as well as the translated operation are applied to the IT and business model, respectively, to produce the models IT_1 and B_1, and also the correspondences are updated. In this particular case, new correspondences are created e.g. between the control structures of the inserted fragments.

- The operation *InsertActivity('Check Consistency with Records', Merge, Decision)* is translated into *InsertActivity('Check Consistency with Records', Merge, Decision)*, where the new parameters now refer to elements of the business model. These operations are then also applied, in this case to IT_1 and B_1, and correspondences are updated.
- The operation *MoveActivity('Get Request Details', Merge, 'Check Consistency with Records')* is translated into *MoveActivity('Get Request Details', Merge, 'Check Consistency with Records')*, where the new parameters now refer to elements of the business model. Again, as in the previous steps, the operations are applied and produce the new Shared Process Model consisting of B' and IT'.

In general, when propagating a change operation, it can occur that the insertion point in the other model cannot be uniquely determined. For example, if a business user inserts a new task between the activity *'Get Insurance Details'* and *'Validate Claim'* in Fig. 6, then this activity cannot be propagated to the IT view automatically without user intervention. In this particular case, the user needs to intervene to determine whether the new activity should be inserted before or after the activity *'Log Session Data'*.

In addition to computing changes and propagating them automatically, in many scenarios it is required that before changes are propagated, they are approved from the stakeholders. In order to support this, changes can first be shown to the stakeholders and the stakeholders can approve/disapprove the changes. Only approved changes will then be applied. Disapproved changes are handed back to the other stakeholder. They will then have to be handled on an individual basis. Such a change management can be realized on top of our change propagation.

4.6 Implementation

As proof of concept, we have implemented a prototype as an extension to the IBM Business Process Manager and as an extension to an open source BPMN editor. A recorded demo of our prototype is publically available [8]. Our current prototype implements initialization of a Shared Process Model from a BPMN process model, check-in of private and public changes to either view and change propagation between both views. Furthermore, we have implemented a check for strong consistency, which can be triggered when checking in private changes. We currently assume that the changes between two subsequent IT views (or business views respectively) are either all public or all private.

With an additional component, this assumption can be removed. Then, the change script is presented to the user who can then mark the public changes individually. For this scenario, the compare/merge component needs to meet the following two requirements: (i) the change script must be consumable by a human user and (ii) individual change operations presented to the user must be as independent as possible. Note that the change operations in a change script are in general interdependent, which restricts the ability to apply only an arbitrary subset of operations to a model. Therefore, a compare/merge component may not support to separate all public from all private changes.

In fact, we first experimented with a generic compare/merge component from the EMF Compare Framework, which could be used to generate a change script for two process model based on the process metamodel, i.e., BPMN 2. The change operations were so fine-grained, e.g. 'a sequence flow reference was deleted from the list of incoming sequence flows of a task', such that the change script was very long and not

meaningful to a human user without further postprocessing. Furthermore, the BPMN 2 metamodel generates very strong dependencies across the different parts of the model so that separate changes were likely to be dependent in the EMF Compare change script.

For these reasons, we switched to a different approach with compound changes as described above. Note that the change approval scenarios described in Sect. 3.2 generate the same requirements for the compare/merge component: human consumability of the change script and separability of approved changes from rejected changes.

5 Related Work

We used prior work [15] on comparing and merging process models on the same abstraction level. Our work deals with changes of models on different abstraction level and distinguishes between public and private changes.

Synchronizing a pair of models connected by a correspondence relation is an instance of a symmetric delta lens [6]. In a symmetric delta lens, both models share some information, but also have some information private to them. Deltas are propagated by translation, which has to take the original and the updated source including the relation between them and original target model including the correspondence to the original source as a parameter. Symmetric delta lenses generalize the state-based symmetric lenses by Pierce et al. [12]. In recent years, various techniques have been developed for synchronization of models. Popular approaches are based on graph grammars (e.g. Giese et al. [10]). In contrast to these approaches, our idea of explicitly marking private elements is novel.

In the area of model-driven engineering, the problem of a coupled evolution of a meta-model and models is related to our problem. Coupled evolution has recently been studied extensively (compare Herrmannsdoerfer et al. [11] and Cicchetti et al. [5,4]). The problem of coupled evolution of a meta-model and models has similarities to our problem where two or more models at a different abstraction level evolve. One key difference is that in our application domain we hide private changes and that we allow changes on both levels to occur which then need to be propagated. In contrast to Herrmannsdoerfer et al., we aim at complete automation of the evolution. Due to the application domain, we focus on compound operations and also translate the parameters according to the correspondences. Overall, one could say that our solution tries to solve the problem in a concrete application domain whereas other work puts more emphasis on generic solutions which can be applied to different application domains.

On an even more general level, (in)consistency management of different views has been extensively studied in recent years by many different authors (e.g. Finkelstein et al. [9], Egyed et al. [7]). The goal of these works is to define and manage consistency of different views where views can be diverse software artefacts including models. As indicated in the paper, our problem can be viewed as one instance of a consistency problem. In contrast, we focus on providing a practical solution for a specific application domain which puts specific requirements into place such as usability and hiding of private changes.

In the area of process modeling, Weidlich et al. [20] have studied vertical alignment of process models, which brings models to the same level of abstraction. They also

discuss an approach for automatic identification of correspondences between process models. Buchwald et al. [3] study the Business and IT Gap problem in the context of process models and introduce the Business IT Mapping Model (BIMM), which is very similar to our correspondences. However, they do not describe how this BIMM can be automatically maintained during evolution. Tran et al. [18] focus on integration of modeling languages at different abstraction levels in the context of SOA Models but they do not focus on the closing the business IT gap as we do. Werth et al. [21] propose a business service concept in order to bridge the gap between the process layer and the technical layer, however, they do not introduce two abstraction layers of process models. Thomas et al. [17] on the other hand distinguish between different abstraction layers of process models and also recognize the need of synchronizing the layers but they do not provide techniques for achieving the synchronization.

Various authors have proposed different forms of abstractions from a process model, called a *process view*, e.g. [16]. A process view can be recomputed whenever the underlying process model changes. Recently, Kolb et al. [13] have taken the idea further to allow changes on the process view that can be propagated back to the original process model, which can be considered as a model synchronization. They restrict to hierarchical abstractions of control flow in well-formed process models.

6 Conclusion

Different process model views are important to reflect different concerns of different process stakeholders. Because their concerns overlap, a change in one view must be synchronized with all other overlapping views in order to facilitate stakeholder collaboration.

In this paper, we have presented detailed requirements for process model view synchronization between business and IT views that pose a significant technical challenge for its realization. These requirements were derived from a larger industrial case study [2] and additional interviews with BPM practicioners. A central intermediate step was the systematic categorization of changes from business to IT level given in Sect. 2.1. We have also presented our solution design and reported first results of its implementation to demonstrate the feasibility of our approach.

We are currently working on the further elaboration and implementation of the change management scenarios described above, and we are preparing an experimental validation with users in order to further demonstrate the value of our approach. Also, not all elements of the BPMN metamodel are currently synchronized but only the main ones. In particular, the synchronization of the layout information of the models was not yet addressed and requires further study.

References

1. Castelo Branco, M., Troya, J., Czarnecki, K., Küster, J., Völzer, H.: Matching business process workflows across abstraction levels. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 626–641. Springer, Heidelberg (2012)
2. Castelo Branco, M., Xiong, Y., Czarnecki, K., Küster, J., Völzer, H.: A case study on consistency management of business and IT process models in banking. In: Software and Systems Modeling (March 2013)

3. Buchwald, S., Bauer, T., Reichert, M.: Bridging the gap between business process models and service composition specifications. In: Int'l Handbook on Service Life Cycle Tools and Technologies: Methods, Trends and Advances (2011)
4. Cicchetti, A., Ciccozzi, F., Leveque, T.: A solution for concurrent versioning of metamodels and models. Journal of Object Technology 11(3), 1: 1–32 (2012)
5. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: Managing dependent changes in coupled evolution. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 35–51. Springer, Heidelberg (2009)
6. Diskin, Z., Xiong, Y., Czarnecki, K., Ehrig, H., Hermann, F., Orejas, F.: From state- to delta-based bidirectional model transformations: The symmetric case. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 304–318. Springer, Heidelberg (2011)
7. Egyed, A.: Instant consistency checking for the UML. In: ICSE, pp. 381–390. ACM (2006)
8. Favre, C., Küster, J., Völzer, H.: Recorded demo of shared process model prototype, http://researcher.ibm.com/view_project.php?id=3210
9. Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., Nuseibeh, B.: Inconsistency handling in multi-perspective specifications. In: Sommerville, I., Paul, M. (eds.) ESEC 1993. LNCS, vol. 717, pp. 84–99. Springer, Heidelberg (1993)
10. Giese, H., Wagner, R.: From model transformation to incremental bidirectional model synchronization. Software and System Modeling 8(1), 21–43 (2009)
11. Herrmannsdoerfer, M., Benz, S., Juergens, E.: Cope - automating coupled evolution of metamodels and models. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 52–76. Springer, Heidelberg (2009)
12. Hofmann, M., Pierce, B., Wagner, D.: Symmetric lenses. In: POPL, pp. 371–384. ACM (2011)
13. Kolb, J., Kammerer, K., Reichert, M.: Updatable process views for user-centered adaption of large process models. In: Liu, C., Ludwig, H., Toumani, F., Yu, Q. (eds.) ICSOC 2012. LNCS, vol. 7636, pp. 484–498. Springer, Heidelberg (2012)
14. Küster, J., Völzer, H., Favre, C., Castelo Branco, M., Czarnecki, K.: Supporting different process views through a shared process model. Technical Report RZ3823, IBM (2013)
15. Küster, J.M., Gerth, C., Förster, A., Engels, G.: Detecting and Resolving Process Model Differences in the Absence of a Change Log. In: Dumas, M., Reichert, M., Shan, M.-C. (eds.) BPM 2008. LNCS, vol. 5240, pp. 244–260. Springer, Heidelberg (2008)
16. Schumm, D., Leymann, F., Streule, A.: Process viewing patterns. In: EDOC, pp. 89–98. IEEE Computer Society (2010)
17. Thomas, O., Leyking, K., Dreifus, F.: Using process models for the design of service-oriented architectures: Methodology and e-commerce case study. In: HICSS, p. 109. IEEE Computer Society (2008)
18. Tran, H., Zdun, U., Dustdar, S.: View-based Integration of Process-driven SOA Models at Various Abstraction Levels. In: Kutsche, R.-D., Milanovic, N. (eds.) MBSDI 2008. CCIS, vol. 8, pp. 55–66. Springer, Heidelberg (2008)
19. Weidlich, M., Barros, A., Mendling, J., Weske, M.: Vertical alignment of process models - how can we get there? In: Halpin, T., Krogstie, J., Nurcan, S., Proper, E., Schmidt, R., Soffer, P., Ukor, R. (eds.) BPMDS 2009. LNBIP, vol. 29, pp. 71–84. Springer, Heidelberg (2009)
20. Weidlich, M., Dijkman, R., Mendling, J.: The ICoP framework: Identification of correspondences between process models. In: Pernici, B. (ed.) CAiSE 2010. LNCS, vol. 6051, pp. 483–498. Springer, Heidelberg (2010)
21. Werth, D., Leyking, K., Dreifus, F., Ziemann, J., Martin, A.: Managing SOA through business services – A business-oriented approach to service-oriented architectures. In: Georgakopoulos, D., Ritter, N., Benatallah, B., Zirpins, C., Feuerlicht, G., Schoenherr, M., Motahari-Nezhad, H.R. (eds.) ICSOC 2006. LNCS, vol. 4652, pp. 3–13. Springer, Heidelberg (2007)

Characterization of Adaptable Interpreted-DSML

Eric Cariou, Olivier Le Goaer, Franck Barbier, and Samson Pierre

Université de Pau / LIUPPA, PauWare Research Group, BP 1155,
F-64013 PAU CEDEX, France
{firstname.name}@univ-pau.fr
http://www.pauware.com

Abstract. One of the main goals of model-driven engineering (MDE) is the manipulation of models as exclusive software artifacts. Model execution is in particular a means to substitute models for code. More precisely, as models of a dedicated domain-specific modeling language (DSML) are interpreted through an execution engine, such a DSML is called interpreted-DSML (i-DSML for short). On another way, MDE is a promising discipline for building adaptable systems based on models at runtime. When the model is directly executed, the system becomes the model: This is the model that is adapted. In this paper, we propose a characterization of adaptable i-DSML where a single model is executed and directly adapted at runtime. If model execution only modifies the dynamical elements of the model, we show that the adaptation can modify each part of the model and that the execution and adaptation semantics can be changed at runtime.

Keywords: model execution, adaptation, i-DSML, models at runtime.

1 Problem of Interest

As programming languages, domain-specific modeling languages (DSML) can be compiled or interpreted. This distinction was early noticed by Mernik *et al.* [14] when comes the time to choose the most suitable implementation approach for executable DSML:

- Compiled DSML: DSML constructs are translated to base language constructs and library calls. People are mostly talking about code generation when pointing at this approach;
- Interpreted DSML: DSML constructs are recognized and interpreted using an operational semantics processed by an execution engine. With this approach, no transformation takes place, the model is directly executable.

With interpreted domain-specific modeling languages (the term i-DSML is coined in [7]), the ability to run a model prior to its implementation is a time-saving and henceforth cost-saving approach for at least two reasons: (a) It becomes possible to detect and fix problems in the early stages of the software development cycle

P. Van Gorp, T. Ritter, and L.M. Rose (Eds.): ECMFA 2013, LNCS 7949, pp. 37–53, 2013.
© Springer-Verlag Berlin Heidelberg 2013

and (b) ultimately the implementation stage may be skipped. One slogan associated to i-DSML should be "*what you model is what you get*" (WYMIWYG). Meanwhile, software adaptation and self-adaptive software [16] have gained more and more interest. Consequently, when building such software based on i-DSML, the model has to be adaptable at runtime thus requiring to define adaptable i-DSML.

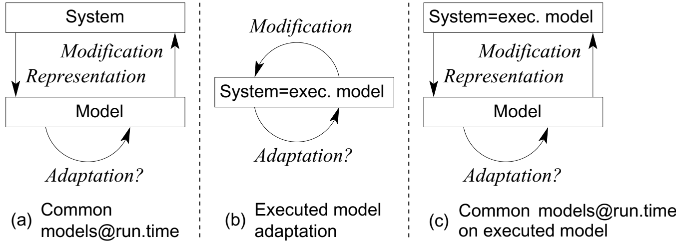


Fig. 1. Adaptation loops

The runtime adaptation problem is commonly tackled as a 2-stages adaptation loop (analyze–modification). In the MDE (Model-Driven Engineering) field, one of the most prominent way to implement this loop is *models@run.time* [2], where models are embedded within the system during its execution and acting primarily as a reasoning support (case (a) in figure 1). The main disadvantage of *models@run.time* deals with maintaining a consistent and causal connection between the system and the model for the model being a valid representation of the system at runtime. The i-DSML approach naturally circumvents this disadvantage since it suppresses the gap between the system and the model: The system is the model being executed. The reasoning is still made on the model but the required modifications are then directly enacted on the model without needing a representation of the system. Case (b) in figure 1 represents the adaptation loop in this new context. However, one major requirement for adaptable i-DSML is that the executed model contains all information necessary for both its execution and its adaptation. This can sometimes lead to an increasing complexity of the model and to a difficulty for managing the adaptation when mixing together adaptation and execution. In this case, one can abstract all the required information into a dedicated model for applying common models@run.time techniques on a model execution system. This solution is depicted by figure 1, case (c). The only difference with the case (a) is simply that the system is of a special kind: A model execution system. The problem of this solution is that it reintroduces the causal link that we precisely try to avoid here. So, both approaches (case (b) and (c)) are complementary and have pros and cons. Depending on the context, one approach will be more suited than the other one.

In this paper, we focus on the direct adaptation of an executed model (case (b) of figure 1). [5,6] are first investigations on this direct adaptation of executed

models, that is, on adaptable i-DSML. They establish what model execution adaptation is and how to express, through a contract-based approach, that a model is consistent with an execution environment (if not, the model has to be adapted). Based on the same example of basic state machines and a train example, the contribution of this paper consists in proposing a conceptual characterization of adaptable i-DSML. The next section recalls what is model execution and its well-known conceptual characterization. This characterization is then extended in section 3 for describing what an adaptable i-DSML contains and is based on. If model execution only modifies the dynamical elements of the model, we show that the adaptation can modify each part of the model and that the execution and adaptation semantics can be changed at runtime. Finally, related work is discussed before concluding.

2 Characterization of i-DSML

Defining executable models is not really a novel idea. Several papers have already studied model execution, such as [3,4,7,8,9,10,13,15]. All these works establish a consensus about what the i-DSML approach assumes:

– Executing the model makes sense. This is much more empirical evidence that shows us that some kinds of model are executable, others are not;
– An engine is responsible for the execution of the model, that is, its evolution over time;
– The model comes with all the information necessary for its execution through an engine: It is self-contained.

Before characterizing precisely what an i-DSML contains, we give a better understanding of these three assumptions.

2.1 Executable Nature of Models

It exists a general classification of models that may help us to identify models which have the ability to be executed or not: The product–process duality. Indeed, models (and meta-models thereof) can either express products or processes, regardless of the system studied. By essence, only process models enable executability of their content since they embody concepts closely related to the world of runtime: Startpoint, endpoint, time (past/current/future), evolution step, etc.

Applied to the field of software development standards, we can cite SPEM as a process modeling language and CMW as a product modeling language. As another OMG's prominent example, UML itself provides three categories of diagrams, namely structure diagrams (Class, Package, . . .), behavior diagrams (State Machines, Activity, . . .) and interaction diagrams (Communication, Sequence, . . .). Logically, only behavior and interaction diagrams may be executed. Beyond these specific examples, when designing a DSML, it is important to keep in mind its potential executable nature.

2.2 Execution Engines

An i-DSML is more than just a meta-model (abstract syntax and well-formedness rules). A language definition also contains a concrete syntax and semantics. The semantics of the language are captured in the transformation rules in the case of compiled DSML or in the execution engines in the case of interpreted DSML. An execution engine is dedicated to a single i-DSML (UML state machines, SPEM, Petri nets, etc.) and can execute any model conforming to this i-DSML.

The purpose of any execution engine is to "play" or to "animate" the model, making its state evolving, step by step. Execution operations, implemented by the execution engine and potentially attached to an element of the meta-model, manage each execution step of the model. The current state of the model can be maintained locally within the engine or, differently, embedded into the model itself. The i-DSML approach singles out having self-contained models embedding their current state but the former solution can be useful in some cases. Typically, this is when one requires to execute models conforming to a meta-model not designed for managing the current state of the model, such as all the dynamic diagrams of UML. Indeed, the UML standard does not define a current model state for any of these diagrams that could be executable. In this case, the solution is to store the current state of the model within the memory of the engine or to extend the meta-model for managing a current model state. For instance, [4] did it for UML state machines. However, the extended meta-model differs from the UML standard.

2.3 Self-contained Executable Models

When self-contained, the current state of the model being executed is stored in the model itself. Thus, each execution step changes this state. At first glance, this strategy seems to pollute the meta-model with many details not relevant at design-time and seems to defeat the abstraction offered by traditional modeling principles (a model is the abstraction and a simplification of a given system). However, there are two main reasons justifying to have self-contained models.

The first one is that it offers the major advantage that after each execution step the current model state can be serialized into an output file[1], thereby providing a complete traceability of the execution as a sequence of models. Some works, such as [3], even consider that the model can embed its complete execution trace in addition to its current state. Based on this sequence of snapshots, one can perform some useful operations like rollbacks, runtime verification (such as the black-box execution verification of [4]), debugging, testing, and so forth.

The second and main reason is related to the essence of the executable models. Such models aim at being substituted to the code, at the lowest level, far away from abstract design models. Hence they have an increased level of details and complexity required for their executability. Moreover, executability being part of

[1] That is why some authors may consider an execution process just as a sequence of endogenous model transformations, as explained in [4].

their nature, it is unsurprising that they contain elements related to executability such as the definition of their current state.

2.4 The Design of an i-DSML

All elements required for model execution are located at the meta-level. Indeed, a language is a self-contained i-DSML if one can instantiate models that are executable through an execution engine. One can identify a recurring pattern [9] about the constituents of an i-DSML (figure 2):

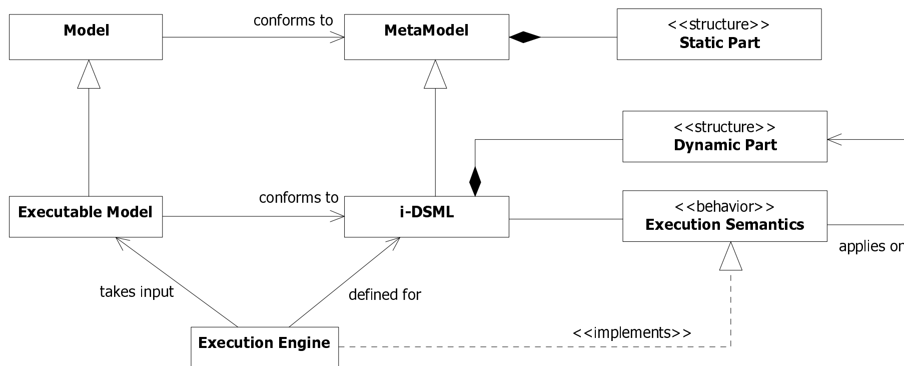


Fig. 2. Conceptual framework for an i-DSML

1. Meta-elements which express the organization of the process modeled,
2. Meta-elements which express the state of the model being executed,
3. Execution semantics which express how the model behaves when being executed.

The item (1) is realized by traditional meta-modeling since the engineers concentrate onto the static structure and associated well-formedness rules of the models to be built. This is called the Static Part. Item (2) introduces structural meta-elements intended to store the state of the model at a given point of the time, also associated with their well-formedness rules. This is called the Dynamic Part. Last but not least, item (3) deals with defining how the the model is evolving over time, modifying only the dynamic part (*i.e.* the static part never changes). An execution semantics can be defined under several declinations. An axiomatic semantics enables to complete the specification of the meta-model with well-evolution rules defining the constraints on how the model evolves [4]. A translational semantics can be used to apply simulation or verification techniques of another technological space, such as in [8]. Finally, an operational semantics is the operationalization of the execution behavior in terms of actions through an action language and is implemented by an execution engine. As depicted by the figure 2, the Dynamic Part and the Execution Semantics are specific to i-DSML.

2.5 Executable State Machines

UML state machines are typically one of the best examples of well-known executable models. In this paper, we then link our examples to them but, for fluency, we define concise state machines restricted to a limited number of features: Composite states with deep history states and transitions associated with an event. Moreover, UML state machines as defined by the OMG do not include a dynamic part as required for a full-model execution (however [4] proposes an extension of the UML meta-model for defining a dynamic part for state machines).

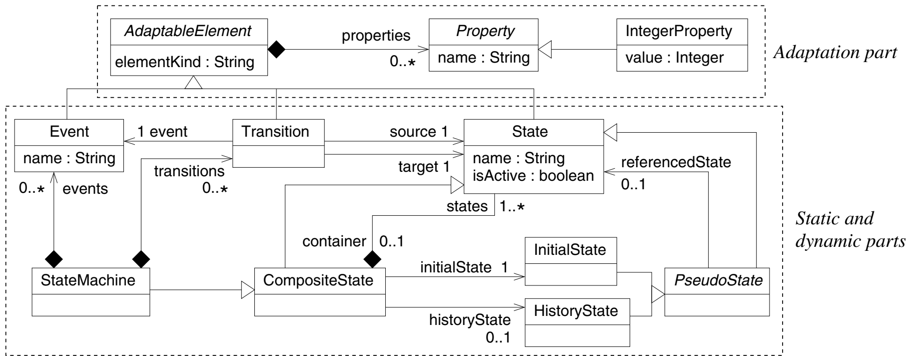


Fig. 3. Meta-model of adaptable and executable basic state machines

The meta-model of our basic state machines is represented on figure 3 (the `AdaptableElement` and `[Integer]Property` elements are dedicated for managing the adaptation and will be introduced in the next section). The static part of the meta-model is composed of the following elements:

- A *state* that is named and is contained in a *composite state*.
- Two kinds of *pseudo states* can be defined: An *initial state* and an *history state*, each one referencing a state of the composite state to which they belong. Each composite state must have one and only one initial state. Having an history state (but only one) is optional.
- A *transition* between a source and a target states, associated with an *event* that is simply defined with a name.
- A *state machine* is a special kind of composite state. It owns all transitions and events of the model and must be unique within the model.

The dynamic part of the meta-model is simply composed of two elements. The first one is the `isActive` boolean attribute of the `State` element. It enables to set that a state is currently active or not. The second is the referenced state of an history state that references the last active state of the composite state to

which it belongs. One can note that the `referencedState` relation of a pseudo state is either playing a static role (for an initial state) or a dynamic one (for an history state). This state machine meta-model has been implemented in Ecore for the EMF platform.

Static and dynamic parts are complemented with OCL invariants defining the well-formedness rules for fully specifying the meta-model. For the static part, it is for instance required to express that a pseudo state references one of the states of its composite state. For the dynamic part, the main invariant specifies the active state hierarchy consistency: Either all the states are inactive (the model is not being executed) or there is in the state machine one and only one active state, and if this state is composite, it contains one and only one active state and so on.

Concerning the execution semantics, both well-evolution rules, defined using OCL, and an operational semantics, implemented by a Kermeta[2] engine, have been defined. For the sake of brevity, we will not present them. Just note that their main goal is to define and to implement the main execution step ensuring that, for an event occurrence, the right transition is triggered depending on the current active states (that is, the active state hierarchy is accordingly modified).

2.6 A Train Example

The example of this paper is freely inspired of a railway system[3]. The behavior of a train is specified through a state machine. The train is stopped or is running at a given speed, depending on the light signals along the railway. The environment of execution of the train is the signals that control its speed. Concretely, the different speeds of the train are specified through the states of the state machine whereas the signals are the events associated with the transitions between these states. Within the same state machine, one can specify the behavior of the system (the train) and its interaction with the execution environment (the light signals).

Execution Environment. The train is running on railways having signals with 3 or 4 different color lights. There are two different kinds of railways: Normal speed sections (up to 130 km/h) and high speed sections (up to 300 km/h). The signal with 3 colors is for normal speed sections while the signal with 4 colors is for high speed ones. The meanings of the colors are the following (only one light is put on at the same time): *red* means that the train must stop immediately, *amber* expresses that the train must not run at more that 40 km/h, *green* means that the train can run at a normal speed (but not more) and *purple* that the train can run at a high speed.

Basic Train Model. The figure 4 represents a state machine defining the behavior of a non high-speed train and some steps of its execution. Concretely,

[2] `http://www.kermeta.org/`

[3] The state machines of train behaviors and their associated signals of this paper are not at all intended to be considered as realistic specification of a railway system.

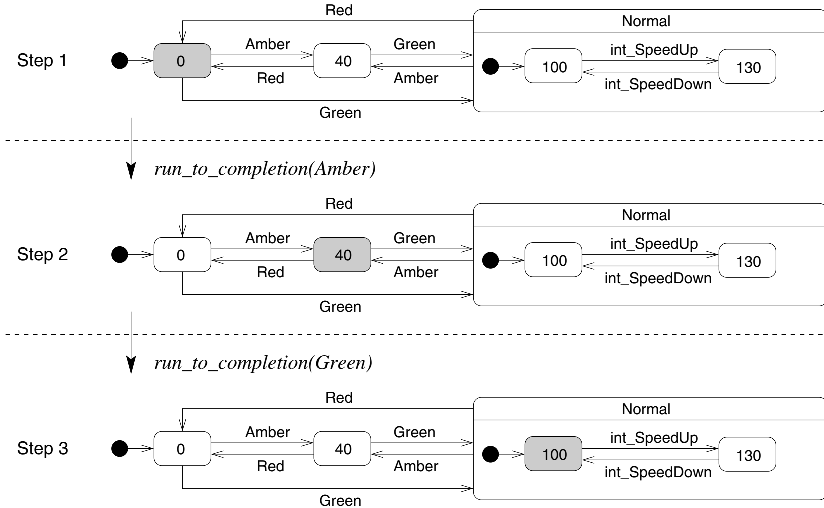


Fig. 4. A train state machine execution

this train is not able to run at more than 130 km/h but is allowed to run at this speed on high-speed sections. The states define the speeds of the train. For simplifying, the name of a state is the train speed in km/h associated with this state. 0 and 40 are then speeds of 0 km/h and 40 km/h, that is the stop state and the low speed state. When running at a normal speed, the train driver can choose between two speeds, either 100 or 130 km/h. These two states have been put into a composite one representing the normal speeds. Transitions between states are associated with the signal color lights: *red*, *green* and *amber*. The *purple* color is not managed here, as the train cannot run at more than 130 km/h, but it can run at 100 or 130 km/h on a high speed section. There are two particular events: `int_SpeedUp` and `int_SpeedDown`. These events are internal actions of the train, that is, correspond to direct train driver actions. For not confusing them with the external events coming from the execution environment, their names is prefixed by "int_".

Execution of the Train State Machine. The figure 4 shows three steps of execution of the state machine, through the `run_to_completion` operation taking an event name as parameter. This operation processes an event occurrence. The first step is the initial active configuration of the state machine: Its initial state `0` is active (an active state is graphically represented with a grey background). Then, for the second step, the `Amber` event occurs and it makes changing the current active state that is now the `40` one. Finally, the third step is the result of the `Green` event occurrence which leads to activate the `Normal` state and in consequence its initial state, the state `100`.

Processing a state machine execution consists only in modifying the `isActive` attribute value for the states and, for composite states, in changing the referenced state of its potential history state. As a conclusion, only the dynamical elements of the model are modified during the execution.

3 Characterization of Adaptable i-DSML

We consider that an adaptable i-DSML is the logical extension of an i-DSML. Indeed, adaptable models are executable models endowed with adaptation capabilities.

3.1 The Design of an Adaptable i-DSML

Figure 5 depicts the design of an adaptable i-DSML. As an adaptable i-DSML is an extension of an i-DSML, this figure extends the figure 2. The added elements are:

1. Meta-elements which express properties on the model and that should help its adaptation;
2. Adaptation semantics, leveraging from the aforesaid properties, which express the adaptation problem and its solution.

Item (1) makes reference to any structural elements and their well-formedness rules that are added in the meta-model and whose role is to facilitate the subsequent adaptations. This is called the Adaptation Part. Item (2) denotes the adaptation semantics that is a specialization of an execution semantics. Indeed, while execution semantics prescribes a nominal behavior, the adaptation semantics expresses also a behavior but for extra-ordinary or abnormal situations requiring an adaptation. Again, an adaptation semantics can be declined under the specification form for complementing the meta-model definition [5], or under the operational form. As a consequence, an adaptation engine implementing the adaptation semantics is an extension of an execution engine: It processes both the execution-related operational semantics and the adaptation-related operational semantics.

Without going into details of how an adaptation semantics is managed or processed by the engine, we can say that it will mainly be composed of a set of fine-grained adaptation operations combined by a set of rules. Some operations are dedicated to checking the consistency of the model and others are actions concretely realizing the adaptation. The rules are expressed under the form "`if <check> then <action>`" and any more complex or recursive combination of the same kind.

The major point is that the adaptation semantics applies on elements of all constituents of the adaptable i-DSML: All the structural parts (static, dynamic and adaptation) and the behavioral ones (execution semantics and adaptation semantics) are concerned. Concretely, at runtime, the model's entire content can be changed including the executed semantics. This brings reflexivity to the adaptable i-DSML since enabling the adaptation of the adaptation (*i.e.* meta-adaptation).

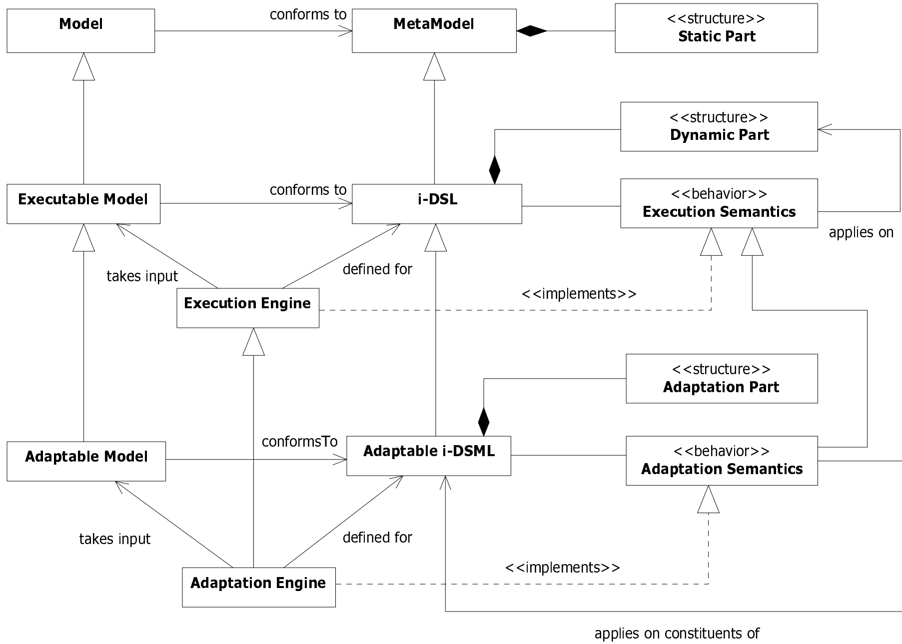


Fig. 5. Conceptual framework for adaptable i-DSML

Categories of Adaptation Actions. We identified two categories of adaptation actions: Create/Update/Delete (CUD) and Substitution. CUD actions target instances of any structural parts (static, dynamic, and adaptation). Substitution is an action which targets the behavioral parts (execution and adaptation semantics). We consider only substitution for behavioral parts because it is not feasible to define a new semantics from scratch (concretely, it will consist in writing lines of code in the engine). Instead, having a set of existing semantics (*i.e.* already implemented) and choosing at runtime which of them to process is straightforward. CUD and substitution can be applied with the following purposes:

- CUD on the dynamic part: The current state of execution is altered (*e.g.* force to go back/go forward, restart such as activating a given state for a state machine),
- CUD on the static part: The structure of the model itself is changed (*e.g.* modification of the modelized process such as adding a state or changing the initial state of a composite for a state machine),
- CUD on the adaptation part: An adaptation-related element is changed (*e.g.* the value of a QoS property is modified accordingly to a changing execution environment),
- Substitution on the execution semantics: Switch from a given interpretation of the model to another one (*e.g.* managing execution variants such as the Harel vs UML transition conflict semantics for state machines),

– Substitution on the adaptation semantics: Switch from a given set of adaptation operations to other ones within the adaptation rules (*e.g.* checking the consistency of the model with the execution environment in an exact or fail-soft mode).

Table 1. Adaptation and execution characteristics

Elements of adaptable i-DSML		**Execution actions**	**Adaptation actions**
<<Structure>>	Static Part	N/A	Create/Update/Delete
	Dynamic Part	Create/Update/Delete	Create/Update/Delete
	Adaptation Part	N/A	Create/Update/Delete
<<Behavior>>	Execution Semantics	N/A	Substitution
	Adaptation Semantics	N/A	Substitution

Table 1 sums up these categories of adaptation actions and contrasts with the actions processed by a simple model execution. Indeed, in this case, only the dynamic part of the model is modified whereas for model adaptation, all parts and semantics can be changed.

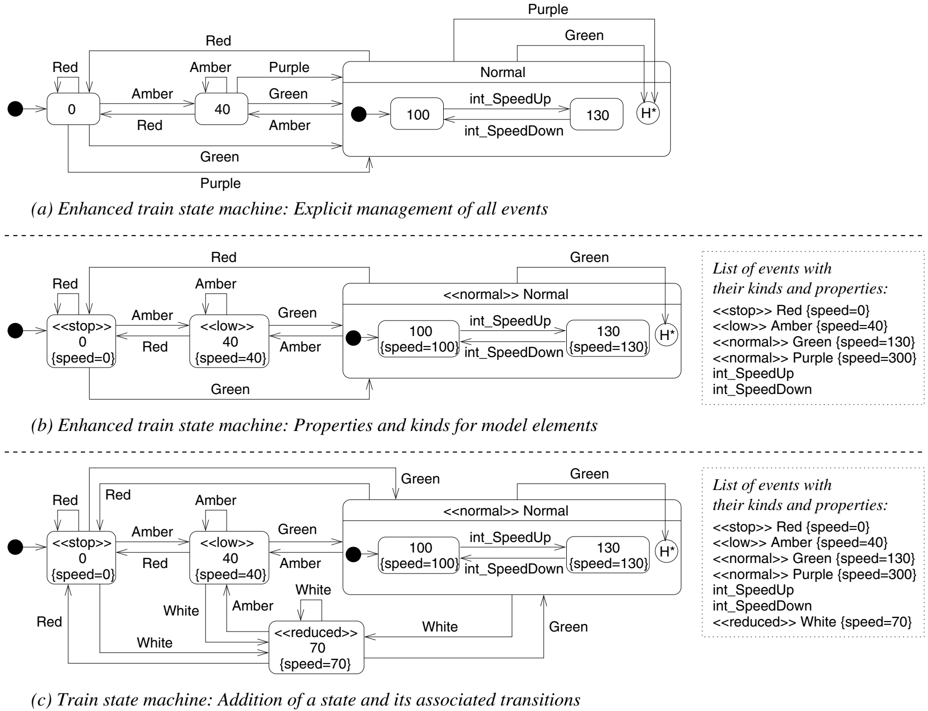
3.2 Adaptation Part for the State Machine Meta-model

The meta-model of state machines of the figure 3 includes elements dedicated to the adaptation management. The first one is the `elementKind` attribute available for the `Event`, `Transition` and `State` elements through the specialization of `AdaptableElement`. This attribute allows the definition of "kinds" for events, transitions and states of a state machine. A kind has for goal to precise that an element is playing a particular role. Conceptually, a kind is equivalent to a stereotype of UML profiles. In addition, through the `properties` association, events, transitions and states can be associated with properties. A property is basically composed of a name and a value. For simplicity, only integer properties are considered here, but of course, properties of any type could be added on the meta-model (the definition of properties can of course be based on reusing existing adaptation works, such as [11] which defines a generic meta-model for specifying properties and associated rules depending on their values). Properties can deal if required with QoS parameters and values. Conceptually, a property is equivalent to a tagged value of UML profiles.

As shown in the following, kinds and properties can be used for managing the adaptation of a state machine execution thanks to the additional information they offer. Kinds can be used to define fail-soft mode of consistency against an execution environment. Properties associated with events of the execution environment enable the modification of the executed state machine for managing unexpected events.

3.3 Runtime Adaptation of the Train State Machine

The main adaptation rule consists in first checking if the behavior of the system is consistent with the execution environment, that is concretely here, if any signal is understandable and correctly processed by the train state machine. Secondly, when the system is not consistent with the environment, to perform an adaptation action such as switching in a fail-soft checking mode or modifying the structure of the model.



(a) Enhanced train state machine: Explicit management of all events

(b) Enhanced train state machine: Properties and kinds for model elements

(c) Train state machine: Addition of a state and its associated transitions

Fig. 6. Adaptation of the train state machine

As explained in [5,6], specific design constraints can be applied for verifying the consistency of a model against an execution environment. The problem is that it is necessary to be able to distinguish an unexpected interaction – requiring to take an adaptation decision – from an expected and already managed one. For state machines, it is required to be able to determine if an event is expected or not. Several solutions are possible, such as parameterizing the execution engine with the list of expected events and to verify for each event occurrence if it is in the list. The solution applied here is self-contained in the model by adding explicitly a transition starting from each state for each expected event. Figure 6, part (a), shows the modification of the train state machine: For every expected

color events (*red*, *amber*, *green* and *purple*), there is a transition starting from each state. When a color event leads to remain in the same state, it is a self-transition (leading to the historical state of a composite for composite states). Now, excepting for internal events starting by “int_”, each event, corresponding to a color of signal crossed by the train, triggers a transition. If no transition is associated with an event according to the current active states, then this event is unexpected and an adaptation action has to be performed.

Based on this new train state machine definition, as an illustration, we describe here seven adaptation operations (two adaptation checking and five adaptation actions including one modifying the adaptation semantics) and two execution semantics variants for the state machine meta-model.

Adaptation Checking. The main verification is to determine if an event is expected or not. This verification can be made in an exact mode or in a fail-soft mode through the kinds of events. Let us consider the occurrence of the *purple* signal event. The train is not able to run at a high speed level, so, when crossing such a *purple* signal, the train will run at a normal speed. For this reason, the train state machine of figure 6 part (a) leads to the state `Normal` for each occurrence of the *purple* event.

The exact mode consists in verifying that there is a transition associated with the exact event (through its name) and the fail-soft mode that there is a transition for the kind of event and not necessary for the exact event. Figure 6, part (b), shows a variant of the train state machine where kinds and properties have be added onto states and events. There are three kinds of states (the kinds are represented through the << .. >> UML stereotype notation): The `0` state is a stop state, the `40` one is a low speed state and the composite state `Normal` is a normal speed state. The events, depending of their associated target state[4], are also tagged with kinds: *red* is a stop event, *amber* is a low speed event, *green* and *purple* are normal speed events. One can notice that transitions associated with the *purple* color have disappeared. Indeed, in a fail-soft mode, the *purple* event will be processed as the *green* one because there come from the same normal kind. The *purple* color is for the train considered as a *green* one even if they do not have the same role and meaning.

The verification mode can be changed at runtime: A checking adaptation operation can be substituted by another one (changing in that way the adaptation semantics). If the model of the figure 6, part (b), is executed and if the train is running on normal speed sections, then the verification mode can be the exact one because the *red*, *amber* and *green* signals that can be crossed are directly and exactly processed for each state of the state machine. However, if the train is now running on a high-speed section, it can cross a *purple* signal that is not directly processed in the exact mode. An adaptation action can be to switch into

[4] We rely on a dedicated restriction on the state machines: All transitions associated with a given event are always leading to the same state. For instance for the train state machine, independently of the source state, the *amber* event always leads to the state `40`.

a fail-soft verification mode and to recheck the validity of the *purple* signal. In this mode, as explained, this signal will be considered as a *green* one and will be processed.

From an implementation point of view, our Kermeta execution engine has been extended for managing the adaptation. Mainly, a pre-condition has been added for the `run_to_completion` operation that processes each event occurrence. This pre-condition performs the chosen adaptation checking and, if not respected, an adaptation action is executed.

Adaptation Actions. In addition to substituting an adaptation checking operation by another one, several adaptation actions can be taken in case of unexpected events, that is, in case of a changing execution environment. A very basic action is to stop the execution of the state machine if there is no way to decide how to handle the event. A more relevant adaptation action is to load a new state machine (a reference one as defined in [6]) that is able to manage the new execution environment if such a state machine is available.

If the unexpected event is associated with properties, they can be used to determine if this event can target an existing state of the state machine or, if not, to add the associated state and transitions on the state machine. Figure 6, part (b), defines a speed property for each event and state. Properties are represented similarly to tagged values of UML profiles (as the {`speed=XXX`} ones). For instance, the *amber* event and the state `40` are both associated with a speed property of the value 40, that is, 40 km/h. Let us suppose that, if this train state machine is executed, a white signal, of a `reduced` kind and a speed property of 70 km/h, is crossed. In both exact and fail-soft verification modes, this white event is an unexpected one. As no state has a speed property with a value of 70, a new state called `70` is created with the same kind and speed property as the white event. All required transitions, starting from or leading to this new state, have also to be added: Each existing state must be the source of a transition associated with the white event and leading to this new state and for each color event (*red*, *amber* and *green*) there must be a transition starting from this new state and leading to the required state. The figure 6, part (c), shows the resulted runtime modification for managing the white signal. An important point to notice about this model modification is that it is based on the comparison of the properties without requiring to know what they are and what they are representing. The adaptation engine simply compares two sets of properties through their names and values.

A last adaptation action could be to force the activation of the state that is from a stop kind (the state `0` for the train state machine) in case of an unexpected event (this action is different from stopping the execution of the state machine as described above because here the state machine is still being executed). The idea is that the train stops if it does not understand a signal.

Execution Semantics Variants. For state machines, a transition conflict appears when a transition is starting from a composite state and that there is also

a transition associated with the same event starting from an internal state of this composite. The way to choose which transition to fire when this event occurs is a semantics variation point. According to the UML semantics[5], it is the most internal one that is fired while the original Harel statecharts semantics [12] leads to fire the external one starting from the composite.

When loading a state machine model, including at runtime when changing the current state machine by another one suited for a new context of execution, it is required to know with which execution semantics it was designed. We can imagine a property associated with the state machine and precising which kind of transition processing variant must be used. Then, the execution engine embeds the operational code for each semantics and the right one is processed depending of this property value. In other words, when changing the executed model as an adaptation action, the current operational semantics can be substituted by another one if needed.

4 Related Work

As written, several papers such as [3,4,7,8,9,10,13,15] have already studied model execution. Section 2 summarizes the consensual characterization of model execution based on these works.

Concerning the adaptation of model execution, as far as we know, there are no other works except ours which have studied this subject. [5,6] have been used as a base for defining the direct model execution characterization exposed in this paper. The MOCAS platform [1] defines a UML agent state machine observing an executed business UML state machine and requiring changes on its current state or structural content. The adaptation is then made following common models@run.time adaptation techniques (case (c) of figure 1). The problem is that the adaptation and the execution operations are strongly mixed-up in the implementation platform. This leads to the absence of separation of the adaptation logic from execution. Moreover, the platform does not enable to replace at runtime the adaptation or execution semantics as we propose.

[13] offers a characterization of models at runtime and related adaptation in the same spirit of this paper. But there are two main differences with our characterization of direct adaptation of model execution: (a) It always considers that the model, even when executable, is causally connected with a running system whilst for us the executed model is by essence the system and (b) it does not go as far as us about the elements that can be modified by the adaptation: It does not consider that the execution semantics or the adaptation semantics can be changed.

5 Conclusion

In this paper we propose a conceptual characterization of the direct adaptation of model execution, through the concept of adaptable i-DSML. Albeit model

[5] `http://www.omg.org/spec/UML/2.2/`

execution and adaptation are closely related (as an adaptation semantics is a specialized execution semantics), there are two sharp demarcation lines. The first one, from a general point of view, is about the intention embodied in the semantics. Indeed, an execution semantics deals with a nominal behavior whereas an adaptation one concerns extra-ordinary or abnormal situations. The second one, from a technical point of view, is that model execution only modifies the dynamic elements of the model whereas model adaptation can modify each part of the model and the execution and adaptation semantics.

The presented execution and adaptation engine is a first prototype showing the interest of studying the adaptation of an executed model. We need to develop more realistic and complex case studies and to consider the adaptation of other kinds of adaptable i-DSML. Notably, we plan to extend our existing tools dedicated to execute full standard UML state machines: SimUML[6] is a simulation tool at the design level and PauWare[7] a Java library implementing and executing UML state machines for any Java platform, including Android devices. The goal is to define and implement adaptation operations and semantics for UML state machines. We plan also to enhance our MOCAS platform for making it clearly separating the adaptation from the execution. These platforms and complex case studies will allow us to study the limits of directly adapting a model execution versus applying common models@run.time techniques on it. One of our perspective is to determine when one approach is more suited than the other one. Finally, just as an action language is provided for expressing the execution semantics, a mid-term perspective is to provide a full-fledged adaptation language. This DSML will support the adaptation loop by offering language constructs for both checking and actions. Concretely, it will enable to define separately the execution logic and the adaptation one, and then to express how the adaptation checking and actions are orchestrated and weaved with the execution operations.

References

1. Ballagny, C., Hameurlain, N., Barbier, F.: MOCAS: A State-Based Component Model for Self-Adaptation. In: Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2009). IEEE Computer Society (2009)
2. Blair, G.S., Bencomo, N., France, R.B.: Models@run.time. IEEE Computer 42(10), 22–27 (2009)
3. Breton, E., Bézivin, J.: Towards an understanding of model executability. In: Breton, E., Bézivin, J. (eds.) Proceedings of the International Conference on Formal Ontology in Information Systems (FOIS 2001). ACM (2001)
4. Cariou, E., Ballagny, C., Feugas, A., Barbier, F.: Contracts for Model Execution Verification. In: France, R.B., Kuester, J.M., Bordbar, B., Paige, R.F. (eds.) ECMFA 2011. LNCS, vol. 6698, pp. 3–18. Springer, Heidelberg (2011)

[6] http://sourceforge.net/projects/simuml/
[7] http://www.pauware.com/

5. Cariou, E., Graiet, M.: Contrats pour la vérification d'adaptation d'exécution de modèles. In: 1ère Conférence en IngénieriE du Logiciel (CIEL 2012) (2012)
6. Cariou, E., Le Goaer, O., Barbier, F.: Model Execution Adaptation? In: 7th International Workshop on Models@run.time (MRT 2012) at MoDELS 2012. ACM Digital Library (2012)
7. Clarke, P.J., Wu, Y., Allen, A.A., Hernandez, F., Allison, M., France, R.: Formal and Practical Aspects of Domain-Specific Languages: Recent Developments. In: Towards Dynamic Semantics for Synthesizing Interpreted DSMLs, ch. 9. IGI Global (2013)
8. Combemale, B., Crégut, X., Garoche, P.-L., Xavier, T.: Essay on Semantics Definition in MDE – An Instrumented Approach for Model Verification. Journal of Software 4(9) (2009)
9. Combemale, B., Crégut, X., Pantel, M.: A Design Pattern to Build Executable DSMLs and associated V&V tools. In: The 19th Asia-Pacific Software Engineering Conference (APSEC 2012). IEEE (2012)
10. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Meta-Modeling: A Graphical Approach to the Operational Semantics of Behavioral Diagrams in UML. In: Evans, A., Caskurlu, B., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 323–337. Springer, Heidelberg (2000)
11. Fleurey, F., Solberg, A.: A Domain Specific Modeling Language Supporting Specification, Simulation and Execution of Dynamic Adaptive Systems. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 606–621. Springer, Heidelberg (2009)
12. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming 8(3) (1987)
13. Lehmann, G., Blumendorf, M., Trollmann, F., Albayrak, S.: Meta-Modeling Runtime Models. In: Dingel, J., Solberg, A. (eds.) MODELS 2010 Workshops. LNCS, vol. 6627, pp. 209–223. Springer, Heidelberg (2011)
14. Mernik, M., Heering, J., Sloane, A.M.: When and how to develop domain-specific languages. ACM Comput. Surv. 37(4) (2005)
15. Pons, C., Baum, G.: Formal Foundations of Object-Oriented Modeling Notations. In: 3rd International Conference on Formal Engineering Methods (ICFEM 2000). IEEE (2000)
16. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. 4, 14:1–14:42 (2009)

Transformation as Search

Mathias Kleiner[2], Marcos Didonet Del Fabro[1], and Davi De Queiroz Santos[1]

[1] C3SL Labs, Depto. de Informatica
Universidade Federal do Parana', Curitiba, PR, Brazil
{marcos.ddf,daqsantos}@inf.ufpr.br
[2] Arts et Métiers ParisTech, CNRS, LSIS, 2 cours des Arts et Métiers, 13697
Aix-en-Provence, France
mathias.kleiner@ensam.eu

Abstract. In model-driven engineering, model transformations are considered a key element to generate and maintain consistency between related models. Rule-based approaches have become a mature technology and are widely used in different application domains. However, in various scenarios, these solutions still suffer from a number of limitations that stem from their injective and deterministic nature. This article proposes an original approach, based on non-deterministic constraint-based search engines, to define and execute bidirectional model transformations and synchronizations from single specifications. Since these solely rely on basic existing modeling concepts, it does not require the introduction of a dedicated language. We first describe and formally define this model operation, called *transformation as search*, then describe a proof-of-concept implementation and discuss experiments on a reference use case in software engineering.

1 Introduction

In existing Model-driven Engineering (MDE) approaches, model transformations are a key element for performing operations between different models. These operations may be of different nature, such as migration, evolution, composition, exchange, and others. Despite the existing solutions being very different on the set of available capabilities (see [7] for a survey), most of them are rule-based approaches.

These approaches have a simple and efficient principle: given a source metamodel MMa and a target metamodel MMb, the developer defines a set of pattern-matching rules to transform all the model elements from MMa into elements of MMb. These transformation engines have a deterministic behavior, i.e., one source model always produces the same target model. In addition, the transformation rules are unidirectional and need to be fully-specified, i.e., it is necessary to write rules that cover all the (relevant) elements of MMb.

These properties limit the scope of most transformation languages in various scenarios [6]. For instance it may be hard to write rules that cover all the transformation cases. Similarly, it is sometimes desirable to produce more than one target model in order to study the alternatives. Bidirectional behavior requires

P. Van Gorp, T. Ritter, and L.M. Rose (Eds.): ECMFA 2013, LNCS 7949, pp. 54–69, 2013.
© Springer-Verlag Berlin Heidelberg 2013

to write or derive a reverse transformation. Finally, these approaches hardly allow to maintain models consistency without additional mechanisms.

Some of these limitations are directly linked to the fact that most tools do not allow for disjunctions (i.e., choice points in the sense of combinatorial problems) and take decisions solely on the basis of the source model. Indeed searching for multiple target models requires non-deterministic properties: a given source model may produce zero, one or multiple target models that satisfy a set of constraints.

In this paper, we present a novel approach that uses constraint-based search for executing model transformations and synchronizations. At this stage, our challenge is to present this approach to the community and to integrate it at its best in current MDE practices. To this aim, we reuse and extend a *model search* operation proposed in previous work[21] as well as its notion of *partial* model: a model whose known constituents actually conform to their corresponding metamodel, but that should be interpreted with a weaker conformance than the classical closed-world interpretation used in the MDE community.

The core idea is to create a unified metamodel containing source, transformation and target constraints. Different scenarios (creation of target model(s), synchronization of existing models, and others) then mainly resolve to searching for conforming model(s). First, we define a (potentially partial) set of correspondences (i.e., transformation constraints) between the source and target metamodels using basic modeling concepts. Second, we transform the input artifacts into a solver which executes the scenario through a generic model search operation. As a result the engine produces none, one or several output models that *contain* the solution(s) and its generation traces. The specifications are bidirectional and flexible: they can be used to produce either the source or the target model, or even propagate changes from one model to the other depending on the chosen scenario. Another interesting property is that one could introduce an optimization criterion that will be used by the search process to produce one specific solution (the “best” one).

Plan of the article. Section 2 briefly introduces the context of model-driven engineering, constraint programming main principles, and recalls necessary definitions from previous work on *model search*. Section 3 formally defines the *transformation as search* operation, and describes a generic process for its realization along with a running example. Section 4 proposes a prototype implementation and discusses preliminary experiments on a reference use case. Finally, Section 5 discusses related work and Section 6 concludes.

2 Context

2.1 Introduction to MDE and Model Transformation

Model Driven Engineering considers models, through multiple abstract representation levels, as a unifying concept. The central concepts that have been introduced are terminal model, metamodel, and metametamodel. A terminal

model is a representation of a system. It captures some characteristics of the system and provides knowledge about it. MDE tools act on models expressed in precise modeling languages. The abstract syntax of a modeling language, when expressed as a model, is called a metamodel. The relation between a model and the metamodel of its language is called $conformsTo$. Metamodels are in turn expressed in a modeling language for which conceptual foundations are captured in an auto-descriptive model called metametamodel. The main way to automate MDE is by executing operations on models. For instance, the production of a model Mb from a model Ma by a transformation Mt is called a model transformation. The OMG's Query View Transform (QVT) [24] defines a set of useful model operations languages. In particular, it defines a language called QVT-operational which is restricted to unidirectional transformations scenarios, and a language called QVT-relational which can be used for bidirectional and synchronization scenarios. There are multiple model definitions in the literature (see [22] for a deep study), we refine in this article the ones introduced in [18][1].

Definition 1 (model). *A model M is a triple* $< G, \omega, \mu >$ *where:*

- *G is a directed labelled multigraph,*
- *ω (called the reference model of M) is either another model or M itself (i.e., self-reference)*
- *μ is a function associating nodes and edges of G to nodes of G_ω (the graph associated to its reference model ω)*

Definition 2 (conformance). *The relation between a model and its reference model is called conformance and noted $conformsTo$.*

Definition 3 (metametamodel). *A metametamodel is a model that is its own reference model (i.e., it $conformsTo$ itself).*

Definition 4 (metamodel). *A metamodel is a model such that its reference model is a metametamodel.*

Definition 5 (terminal model). *A terminal model is a model such that its reference model is a metamodel.*

2.2 Constrained Metamodels

The notion of constraints is closely coupled to MDE. Engineers have been using constraints to complete the definition of metamodels for a long time, as illustrated by the popular combination UML/OCL [25]. Constraints can be, for instance, checked against one given model in order to validate it. In our approach we will always consider metamodels with potential constraints attached. We refine in this article definitions from [21] to formally define the combination:

[1] Though these may not be the most precise on object-oriented concepts and different model relationships, simple graph-based definition will prove useful in our context.

Definition 6 (constrained metamodel). *A constrained metamodel CMM is a pair $< MM, C >$ where MM is a metamodel and C is a set (a conjunction) of predicates over elements of the graph G associated to MM. We will consider an oracle that, given a model M, returns true (noted $M \in C(MM)$ where $C(MM)$ is the set of all valid models) iff M satisfies all predicates from C.*

The conformance relation between a model and its reference is then naturally extended to constrained metamodels.

Definition 7 (constrained conformance). *A model M* conformsTo *a constrained metamodel CMM iff it $conformsTo$ MM and $M \in C(MM)$.*

Many languages can be used to define predicates (i.e., constraints) with different levels of expressiveness. OCL supports operators on sets and relations as well as quantifiers (universal and existential) and iterators. To ease the specification of metamodel static constraints, we use in this article an OCL-compatible extension (OCL+ [15]) that extends it with multi-context constraints.

2.3 Introduction to Constraint Programming

Constraint programming (CP) is a declarative programming technique to solve combinatorial (usually NP-hard) problems. A constraint, in its wider sense, is a predicate on elements (represented by variables). A CP problem is thus defined by a set of elements and a set of constraints. The objective of a CP solver is to find an assignment (i.e, a set of values for the variables) that satisfy all the constraints. There are several CP formalisms and techniques [17] which differ by their expressiveness, the abstractness of the language and the solving algorithms.

2.4 Introduction to Model Search

A solver-independent integration of constraint programming, called *model search*, for the automatic generation (or completion) of constrained models has been described in [21]. This article builds on those foundations to propose a generic model transformation method based on constrained search. Therefore we need to briefly recall here the main principles and definitions of the model search operation.

Definition 8 (relaxed metamodel). *Let $CMM =< MM, C >$ (with $MM =< G, \omega, \mu >$) be a constrained metamodel. $CMM_r =< MM_r, C_r >$ (with $MM_r =< G_r, \omega, \mu >$) is a relaxed metamodel of CMM (noted $CMM_r \in Rx(CMM)$) if and only if $G_{MM_r} \subseteq G_{MM}$ and $C_r \subseteq C$.*

In other words, a relaxed metamodel is a less constrained (and/or smaller) metamodel. A simple one can be obtained by the removal of all constraints. Computing such a relaxed metamodel, a simple operation which can obviously be done easily with existing techniques, is called *relaxation* in the following.

Definition 9 (partial model, p-conformsTo). *Let* $CMM =< MM, C >$ *be a constrained metamodel and* M_r *a model.* M_r *p-*$conformsTo$ CMM *iff it conforms to a metamodel* CMM_r *such that* CMM_r *is a relaxed metamodel of* CMM $(CMM_r \in Rx(CMM))$*.* M_r *is called a partial model of* CMM*.*

Informally, a partial model is simply understood as being an incomplete model.

Definition 10 (model search). *Let* $CMM =< MM, C >$ *be a constrained metamodel, and* $M_r =< G_r, MM_r, \mu_r >$ *a partial model of* CMM*. Model search is the operation of finding a (finite) model* $M =< G, MM, \mu >$ *such that* $G_r \subseteq G$*,* $\mu_r \subseteq \mu$ *(embedding i.e,* $\forall x \in Gr, \mu(x) = \mu_r(x)$*), and* M $conformsTo$ CMM*.*

In other words, model search extends a partial model into a "full" model conforming to its constrained metamodel (or generates one when an empty request M_r is given). An example process to achieve this operation in a MDE framework is illustrated in Figure 1. Briefly, the request M_r and the metamodel CMM are transformed into the search engine input format where search takes place. The solutions, if any, are then transformed back into the modelling paradigm.

We may thus consider model search as a model transformation where the source (metamodel and model) is an instance of a non-deterministic (combinatorial) problem and the target model is a solution (if any exists). From the CP point of view, the target metamodel acts as the constraint model whereas the source model (the request) is a given partial assignment that needs to be extended.

For deeper information on how this operation is formalized, achieved and integrated as a first-class MDE operation, the reader is kindly referred to [21].

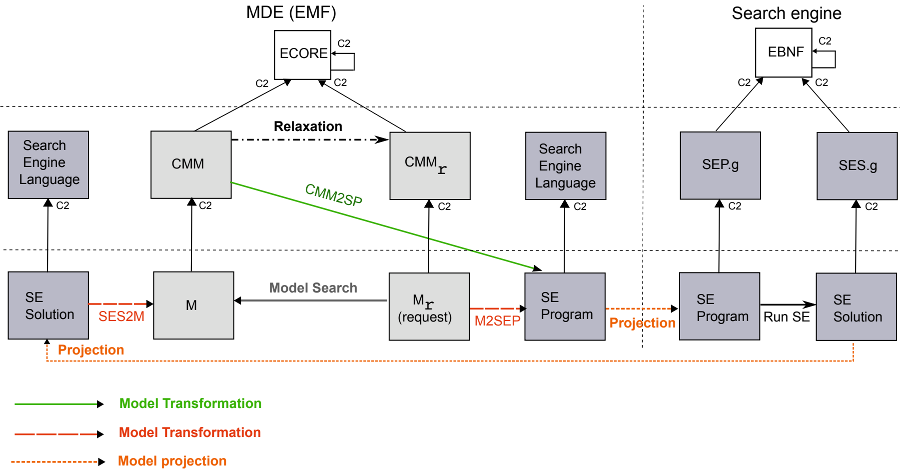


Fig. 1. Model search: example process

3 Transformation as Search

The following proposes to generalize model search to constraint-based model transformation/synchronization by considering different source and target metamodels. The main idea is to define the transformation/synchronization as a set of relations and constraints between elements of the metamodels that are to be related (these may be called *weaving links*). All these artifacts are then unified into a *transformation metamodel*. By applying model search on this unified metamodel, a model which *contains* valid solution model(s) is created.

In the following, we first introduce a running example of a classic transformation scenario (creation of a single target model out of a single source model) which is then used to illustrate the generic process. Its adaptation to other scenarios is then briefly discussed.

3.1 Running Example

The chosen use case is a transformation of a class schema model (MM^A) into a relational schema model (MM^B), known as the *Class2Relational* transformation. This use case is well-known due to its large applicability and it has been studied in other works to demonstrate different aspects about transformation languages, such as [24], [23], and others. The initial metamodels are extracted from the public transformation repository at [3] and illustrated at both sides of Figure 2 (some elements have been omitted to improve readability).

The main scenario, which process is described in the following is the creation of a target model (a relational schema) from a source model (a class schema).

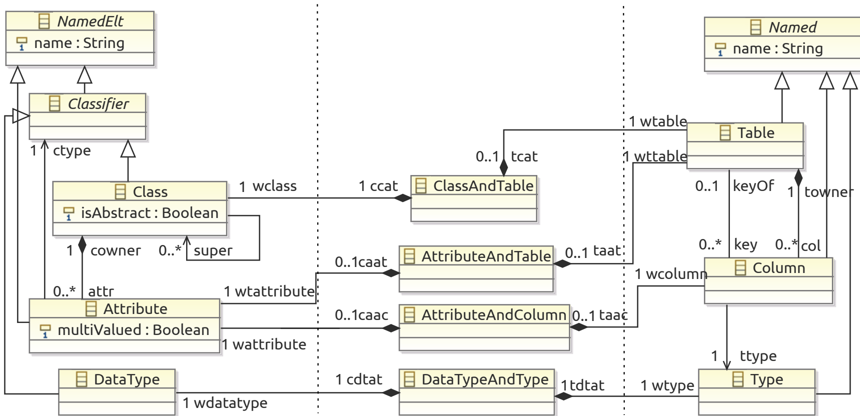


Fig. 2. Extract of the running example transformation metamodel (V1) as an ecore diagram. Initial metamodels are on the sides, weaving metamodel is in the middle.

3.2 Process

Obtaining the Transformation Metamodel by Unification. Figure 3 shows how to obtain a transformation metamodel, called CMM^T, by unification of the source (CMM^A), target (CMM^B) and weaving (CMM^W) metamodels. In our example these inputs are respectively the class schema structure (left part of Figure 2), the relational schema structure (right part), and a set of weaving elements and constraints (middle part, constraints are not shown in the Figure). This operation, consisting merely in copying and combining the sources, can be done with existing transformation techniques. Formal definitions of CMM^W and CMM^T are given below:

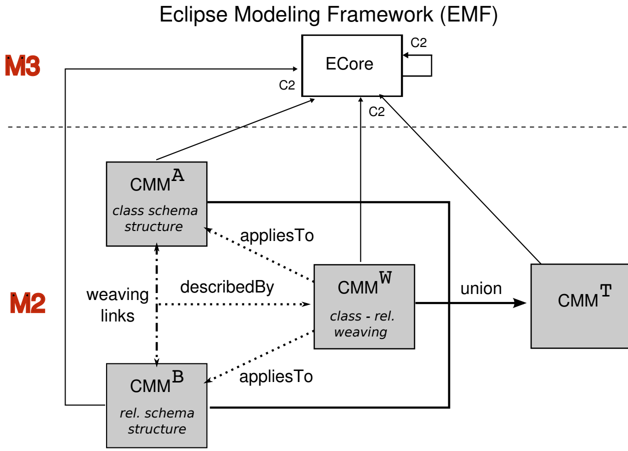


Fig. 3. Obtaining the transformation metamodel by unification (application on the running example shown in italics)

Definition 11 (weaving metamodel). *We call* weaving metamodel *between metamodels* CMM^A *and* CMM^B*, a constrained metamodel* CMM^W *defined by* $CMM^W =< MM^W, C^W >$*, where* MM^W *and* C^W *are respectively a set of metamodel elements and constraints that define the weaving relationships between the elements of* CMM^A *and* CMM^B *(it requires the use of inter-model references).*

Definition 12 (transformation metamodel). *We call* transformation metamodel *between metamodels* $CMM^A =< MM^A, C^A >$ *and* $CMM^B =< MM^B, C^B >$*, using a weaving metamodel* CMM^W*, a constrained metamodel* CMM^T *defined by* $CMM^T =< MM^T, C^T >$*, where* $MM^T = MM^A \cup MM^B \cup MM^W$ *and* $C^T = C^A \cup C^B \cup C^W$*.*

Obviously, unification turns inter-model references in the weaving metamodel into intra-model references in the transformation metamodel.

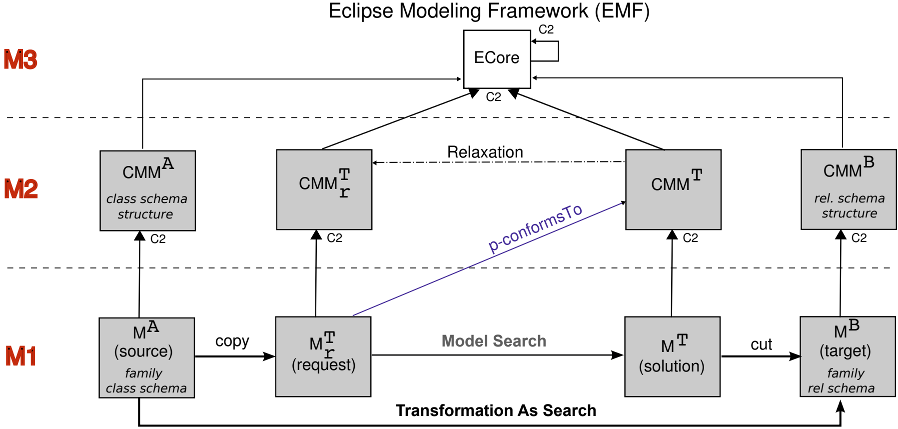


Fig. 4. Transformation as search operation (application on the running example shown in italics)

Searching for a Target Model. Figure 4 shows how a target model can be created by applying model search to the transformation metamodel. In this scenario we have an exogenous unidirectional transformation from one source model (M^A) to a target model (M^B).
The first step is to define the model search request. In this scenario it is simply a copy of the source model. In our running example, it corresponds to the "Family" class schema at the top of Figure 5. From definition 10, a valid request must be a *partial model* of the transformation metamodel (CMM^T). This property is ensured by the following proposition:

Proposition 1. *For all model M^A that conformsTo CMM^A, M^A is a* partial model *of (or p-conformsTo) CMM^T.*

Proof. From definition 9 of p-$conformsTo$, it resolves to finding a relaxed metamodel $CMM^T_r =< MM^T_r, C^T_r >\in Rx(CMM^T)$ such that M^A $conformsTo$ CMM^T_r. From definition 7 of conformance, this requires that (1) M^A $conformsTo$ MM^T_r and (2) $M^A \in C(MM^T_r)$.
Let CMM^T_r be the relaxed metamodel of CMM^T such that $MM^T_r = MM^T$ and $C^T_r = \emptyset$ (i.e., the one obtained by removing all constraints). (2) is obviously true as there are no predicates to satisfy. (1) requires that MM^T_r can be a reference model of M^A, i.e., its graph G^T_r contains all nodes (meta-elements) targeted by the graph G^A of M^A. This is clearly true since by definition 12 of CMM^T we have $MM^A \subset MM^T$ (in particular $G^A \in G^T$), and on the other hand $MM^T_r = MM^T$ (in particular $G^T_r = G^T$).

The second step is to perform the model search. This operation extends any model M^A that conforms to CMM^A into a model M^T that conforms to CMM^T (when there are solutions). By extending the source model, search produces a

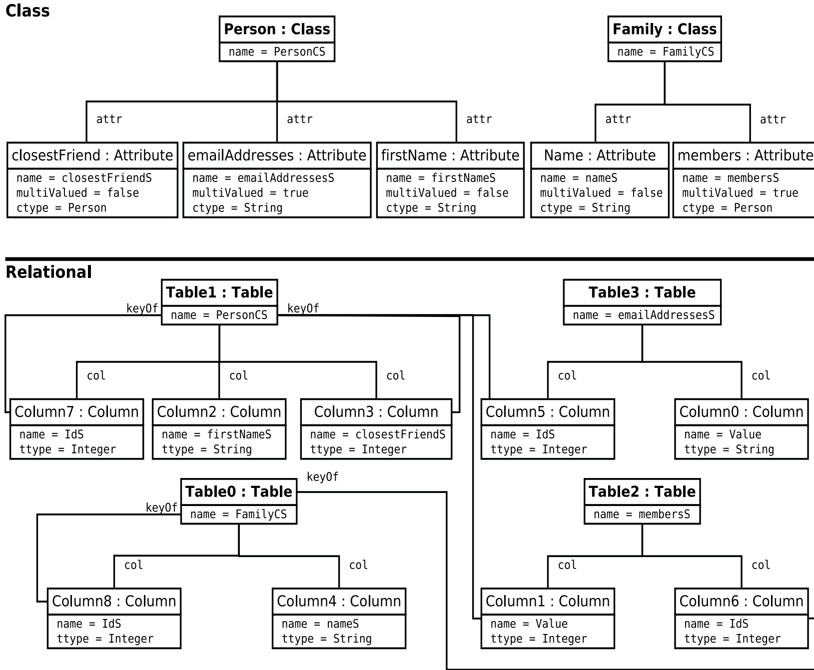


Fig. 5. Input and ouput from the running example (scenario 1) as instance diagrams

model M^T which contains both the target elements and the weaving elements (these can be understood as the transformation traces). Additionally, model search ensures that the target elements satisfy their metamodel constraints, and optimization or preferences may be applied to discriminate between different solutions. To avoid adding source elements to the produced model, as usually expected in classical target creation scenarios, a set of model-level constraints can be added to CMM^T in order to "freeze" the source model.
The final step is to isolate the target model M^B that conforms to CMM^B. This can easily be obtained by removing from M^T any element that is not associated to CMM^B. In our example, a sample result is the "Family" relational schema illustrated at the bottom of Figure 5.

3.3 Other Scenarios

The described scenario is a unidirectional one-to-one operation, but the approach is naturally bidirectional and can be used for different scenarios by varying the search request. Indeed, it suffices to use M^B as the request to obtain the reverse transformation (production of a model M^A). Additionally, the synchronization scenario (propagating source or target model modifications to the other one) can also be achieved with the same specifications: by using the union of the two models (M^A and M^B) as the request, model search will extend them to

satisfy the transformation constraints, and thus update source, target or both models (depending on the desired behaviour). Note that many CP solvers are restricted to constructive modifications since allowing for removal of elements from the request may yield tractability issues. However the approach itself does not prevent it and is only limited in this sense by the capabilities of the chosen underlying solver. All these scenarios are described and experimented on the running example in the next Section.
Finally, though these are not studied nor experimented in this paper, multi-source and/or multi-target transformations could also be achieved: it suffices to add the corresponding metamodels and weaving constraints to CMM^T, and corresponding models to the request. Indeed, constraints may have any arity (i.e., a single constraint may weave multiple metamodels elements).
In all these scenarios, the preceding propositions naturally hold as long as requests are partial models of CMM^T.

4 Implementation and Experiments

In this section we first present the components used for implementing a prototype software chain. Second, we further describe the chosen use case and its realization on different scenarios. Finally, we summarize the experimentation results.

4.1 Implementation

The transformation as search (TAS) components were implemented using:

- the Eclipse Modeling Framework (EMF) [8] for defining, viewing and manipulating the various (meta)models presented in the preceding Section.
- the ATL engine [19] for writing/executing the rule-based transformations of the process (unifying into CMM^T, projecting Ecore-OCL+ to/from the solver language, isolating the target model from the solution model).
- Alloy/SAT [16] as the constraint language/solver.

The software chain is freely available from a single package at [28].

4.2 Use Case

We have defined 3 scenarios:

1. (1) the forward transformation presented in the running example (from class to relational)
2. (2) the reverse transformation (from relational to class)
3. (3) a synchronization scenario (propagation of changes from one model to another after adding new elements to the class schema).

The first goal was to experiment whether one single specification could handle different scenarios. The specification consists of a weaving metamodel, shown

in the middle of Figure 2, which provides the basic correspondences between left and right metamodels. The figure shows correspondences between first-class elements (e.g., classes "Class" and "Table", classes "Attribute" and "Column", etc.). We defined a set of additional constraints, written in OCL+ (as illustrated in the code listing below), to further specify the weaving. Similarly to other transformation techniques, these specifications may be written in different ways. We provide results with two versions of the specifications, each of them used throughout all scenarios, to test the capabilities and behavior of the search. The first version (V1) mimics the behavior of the original (ATL) use case, in particular it creates simple integer columns for foreign keys. The second version (V2) has a more explicit (and better) handling of table primary/foreign keys. It also defines different weaving tables for each connected elements, allowing to write more specific constraints. Both versions can be found in package [28].

As an example, the (V1) OCL+ excerpt below depicts 4 different kind of constraints attached to the weaving metamodel from Figure 2.

Constraint (1) specifies equality between the "name" attribute of weaved "Datatype" and "Type" (similar constraints are given for "Attribute", "Class", "Table" and "Column").
Constraint (2) maps mono-valued attributes to columns (a similar constraint is given for multi-valued attributes/tables and classes/tables).
Constraint (3) ensures that weaved attributes/columns have a weaved datatype/ type (a similar constraint handles classes and integer key columns).
Constraint (4) ensures that attributes cannot have both a weaved table and a weaved column (a similar constraint handles tables with classes and attributes).

```
1- context dtt : DataTypeAndType inv:
    dtt.wdatatype.name = dtt.wtype.name;

2- context att : Attribute inv:
    not att.multiValued implies att.caac.size() = 1;

3- context ac : AttributeAndColumn inv:
    ac.wattribute.ctype.oclIsTypeOf(DataType)
    and not ac.wattribute.multiValued
      implies ac.wcolumn.ttype = ac.wattribute.ctype.cdtat.wtype;

4- context att : Attribute inv:
    att.caac.size() + att.caat.size() = 1;
```

4.3 Experiments

We experimented on the Family use case [3], which class schema is illustrated on the top of Figure 5. This application is used as a proof-of-concept of our approach.The Alloy solver needs a specified pool of available elements: we set

a maximum of 25 instances. We executed the model operations on an Intel Core Duo machine with 2.4GHz and 4GB of RAM. We used the Alloy Analyzer 4.2 with the MiniSat solver for generating search solutions. We summarize the results in Table 1. The given execution times correspond to the search of the first satisfying instance. They do not include the problem generation (projections from/to Ecore/Alloy, Alloy's compilation into SAT) since these are negligible in all our experiments and the metamodel projection only has to be done once for a given specification.

Table 1. Summary of the experiments (Family model)

scenario-version	#variables (primary)	#constraints (unfolded)	execution time (ms)
(1) - V1	9956	845357	3432
(1) - V2	7179	866894	2483
(2) - V1	10114	791167	5529
(2) - V2	5725	866894	1655
(3) - V1	6496	505227	324
(3) - V2	5448	1231787	666

Alloy does not natively support strings as datatypes, we thus used predefined atoms and scalar equality (i.e., "nameS" is the string "name"). Both versions of the specifications generate the relational schema illustrated on the bottom of Figure 5. However, if we cycle through the solutions, V1 also proposes solutions with additional "orphan" tables. This can be prevented with a constraint that forbids creation of elements without a weaving association.

In the reverse transformation scenario (2), we used the exact generated target model (the Family table schema) as input request, freezing again the initial model through the pool of 25 available instances. Both versions generate the original class schema. However in V1, it is not the first found solution. Indeed, the first proposition is a class schema with an "EmailAdresses" class (referenced by "Person") and a "Members" class (referenced by "Family" and with an outgoing reference to "Person"). This is a valid but nevertheless surprising solution at first. If desired, it is possible to prevent such behaviour with a weaving constraint that forces tables to become attributes whenever possible. This is done in V2 using the (explicit) table keys as conditions.

In the synchronization scenario (3), we used the whole transformation result from scenario 1 to create the input request: the class and relational schemas (as shown in Figure 5), plus the generated weaving elements between them (transformation traces, not shown in the Figure). We then added to the class schema an "Animal" class and its outgoing "owner" reference to the "Person" class. The goal was to test the propagation of changes to the table schema. Both versions update accordingly the relational schema, adding an "Animal" table with an "owner" (foreign) key column. The lower execution times are easily explained: search space is reduced since most of the model is given as input (only a limited number of SAT variables have to be assigned).

4.4 Analysis and Future Work

Experiments on these examples show that valid models can be created in different scenarios using single specifications, and that different solutions can be proposed to the designer. Although it is only a first step to assess the potential and flexibility of the approach, this prototype implementation can serve as a proof of feasibility. It is obvious that the produced models could already be obtained using existing transformation techniques, though this may require to write (or derive) additional specifications to handle all scenarios. Also, these experiments raise a number of modeling and operational challenges that will be considered in future work.

From the modeling point of view, defining specifications as weaving elements and constraints can be confusing for a designer used to traditional rule-based engines. For a given set of constraints, cycling through solutions or testing different scenarios may reveal a need to incrementally narrow some constraints, but also exhibit unexpected valid solutions. To ease the writing of specifications, a set of weaving templates can be provided as guidelines. Similarly, a mapping from a more specific language (such as QVT-relations [24]) to an equivalent set of weaving templates can be studied. Also, additional scenarios can be considered. For instance, two independently-obtained models can be given to test (and exhibit) whether a valid mapping is able to relate them. The optimization/preferences capabilities of the approach are also to be experimented more deeply. Overall, an extensive evaluation on different sets of examples is necessary to confirm and exhibit the benefits that can be envisioned compared to existing techniques.

From the operational point of view, this prototype implementation will be extended and experiments conducted to assess tractability on larger models. Early experiments on a larger application, a graduate course management system extracted from a real-world system, show that the current Alloy-based implementation does not easily scale to medium-size models. The size of the instance pool is a major factor, and SAT compilation rapidly induces a combinatorial explosion in transformation scenarios (about 30000 variables and 10 minutes execution time). The propagation scenario seems more tractable (about 1 minute). Source and details of these experiments are also included in the package for reference. It is obvious that constraint solving is more (time) expensive than rule-based approaches (both theoretically and practically). However the constraint programming community has developped a large set of techniques to counter combinatorial explosion (structure-based symmetry breaking, decomposition, heuristics, etc.). The current implementation uses none of these.

Clearly, our aim is not to replace techniques that have proven suitable in many applications, but to show that the properties inherent to this approach are relevant for various problems, and how to integrate it in current MDE practices.

5 Related Work

There are several approaches studying model transformations. A classification can be found at [7]. In particular, the OMG standard QVT [24] has defined

specifications for different type of model operations. Its QVT-operational part has been implemented by various tools such as the popular ATL [19], TEFKAT [23], VIATRA [9], and others. These approaches allow for fast and efficient unidirectional transformations. QVT-relations defines a broader set of capabilities, such as incremental transformations, but has fewer (and only partial) implementations such as [1].

Further studies have tackled bidirectional model transformations or synchronizations (see [6] for a survey), often through mechanisms which require additional specifications to an existing rule-based transformation. For instance, [14] proposes, based on abductive reasoning, to reverse an unidirectional transformation in order to provide synchronization of the source model with the previously obtained target model. In particular, it shares our ability to compute different alternative solutions through combinatorial logic inference. A number of so-called incremental approaches [13,29,4] allow to update a target model by taking into account incremental changes to the source model. Triple Graph Grammars (TGGs) [27] are an approach for defining correspondences between two graphs. The TGG definition is similar to our model unification (i.e., a unique graph created from one left, one right and one correspondence graph) though not grounded in usual metamodelling principles. TGGs are also used for generating transformations (e.g., the [11] tool), but without search capabilities. [2,10] also expore the notion of weaving relationships but do not use them as direct transformation specifications. However, the described patterns may be useful in our context as specifications templates. Due to the high number of other existing transformation approaches and their spreading among different research communities, this overview can hardly be exhaustive. Clearly, many of these techniques have their own benefits over our approach, such as easier removal of elements or faster execution times. However, non-deterministic constrained search offers an original combination of properties: flexibility of having a single specification for different scenarios, ability to provide various solutions, automatic checking/optimizing capabilities, and in our case the sole use of existing modeling concepts.

Finally, the MDE community has also been using constraint-based solvers for various model operations such as model checking [12,5,30], finding optimized transformations [20], or extending transformation capabilities [26]. Although there are definitely similarities in the use of constraint tools, and in particular alternative mappings to solver languages, the pursued goals are different.

6 Conclusion

As model transformations have gained maturity in the past years, novel scenarios have arised that cannot be handled using classical rule-based solutions, mainly because of their non-deterministic nature. In this paper we described an original approach in which the core idea is to *search* for a solution model by satisfying a set of weaving relations and constraints. The approach is built on a previously defined operation, called model search, for the automatic generation of complete

valid models conforming to a metamodel and its constraints. Both model search and transformation as search operations are based on the concepts of partial model and partial conformance.

The paper also describes how this model transformation technique is formalized and automated as a first-class model operation, independently from the solving engine, in order to fit in current MDE practices. The nature of the operation provides original properties: bidirectionality, ability to explore and discriminate different target alternatives based on optimization criteria, synchronization of existing models, etc. However a larger set of examples and an extensive evaluation are required to assess practical benefits. Indeed, the provided prototype implementation on a reference usecase is only a first step that raises a number of modeling and operational challenges for future research.

Acknowledgements. This article was partially funded by project CNPq Universal (481715/2011-8) and Convênio Fundação Araucária/INRIA.

References

1. Medini (2012), http://projects.ikv.de/qvt
2. Akehurst, D.H., Kent, S., Patrascoiu, O.: A relational approach to defining and implementing transformations between metamodels. Software and System Modeling 2(4), 215–239 (2003)
3. ATL Class to Relational transformation, eclipse.org (March 2005)
4. Bergmann, G., Ökrös, A., Ráth, I., Varró, D., Varró, G.: Incremental pattern matching in the viatra model transformation system. In: Proceedings of the Third International Workshop on Graph and Model Transformations, GRaMoT 2008, pp. 25–32. ACM, New York (2008)
5. Cabot, J., Clarisó, R., Riera, D.: Umltocsp: a tool for the formal verification of uml/ocl models using constraint programming. In: Proceedings of the International Conference on Automated Software Engineering, pp. 547–548 (2007)
6. 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)
7. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Syst. J. 45, 621–645 (2006)
8. EMF (2009), http://www.eclipse.org/modeling/emf/
9. Ehrig, K., Guerra, E., de Lara, J., Lengyel, L., Levendovszky, T., Prange, U., Taentzer, G., Varró, D., Varró-Gyapay, S.: Model transformation by graph transformation: A comparative study. In: MoDELS Satellite Events (2005)
10. Didonet Del Fabro, M.: Metadata management using model weaving and model transformations. PhD thesis, University of Nantes (2007)
11. Fujaba Tool Suite (2011), http://www.fujaba.de/
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. 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)

14. Hettel, T., Lawley, M., Raymond, K.: Towards model round-trip engineering: An abductive approach. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 100–115. Springer, Heidelberg (2009)
15. OCL+ usecase (2010), http://www.lsis.org/kleinerm/MS/OCLP_mm.html
16. Jackson, D.: Automating first-order relational logic. In: FSE, pp. 130–139 (2000)
17. Jaffar, J., Maher, M.J.: Constraint logic programming: A survey. J. Log. Program. 19(20), 503–581 (1994)
18. Jouault, F., Bézivin, J.: KM3: A DSL for metamodel specification. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 171–185. Springer, Heidelberg (2006)
19. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
20. Kessentini, M., Sahraoui, H.A., Boukadoum, M.: Model transformation as an optimization problem. In: Proceedings of the MoDELS Conference, pp. 159–173 (2008)
21. Kleiner, M., Didonet Del Fabro, M., Albert, P.: Model search: Formalizing and automating constraint solving in MDE platforms. In: Kühne, T., Selic, B., Gervais, M.-P., Terrier, F. (eds.) ECMFA 2010. LNCS, vol. 6138, pp. 173–188. Springer, Heidelberg (2010)
22. Kühne, T.: Matters of (meta-)modeling. Software and System Modeling 5(4), 369–385 (2006)
23. Lawley, M., Steel, J.: Practical declarative model transformation with tefkat. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 139–150. Springer, Heidelberg (2006)
24. Object Management Group. Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT) Specification, version 1.1 (2011)
25. OCL 2.0 specification (2008), http://www.omg.org/spec/OCL/2.0/
26. Petter, A., Behring, A., Mühlhäuser, M.: Solving constraints in model transformations. In: Paige, R.F. (ed.) ICMT 2009. LNCS, vol. 5563, pp. 132–147. Springer, Heidelberg (2009)
27. 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)
28. Class-Relational usecase (2013), http://www.lsis.org/kleinerm/TAS/TAS-CLAR-usecase.html
29. Vogel, T., Neumann, S., Hildebrandt, S., Giese, H., Becker, B.: Incremental model synchronization for efficient run-time monitoring. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 124–139. Springer, Heidelberg (2010)
30. White, J., Schmidt, D.C., Benavides, D., Trinidad, P., Ruiz-Cortez, A.: Automated diagnosis of product-line configuration errors in feature models. In: Proceedings of the Software Product Lines Conference (2008)

Model-Based Generation of Run-Time Monitors for AUTOSAR

Lars Patzina[1], Sven Patzina[1], Thorsten Piper[2], and Paul Manns[2]

[1] Real-Time Systems Lab, Technische Universität Darmstadt, Darmstadt, Germany
{sven.patzina,lars.patzina}@es.tu-darmstadt.de
[2] DEEDS Group, Technische Universität Darmstadt, Darmstadt, Germany
{piper,manns}@cs.tu-darmstadt.de

Abstract. Driven by technical innovation, embedded systems, especially in vehicles, are becoming increasingly interconnected and, consequently, have to be secured against failures and threats from the outside world. One approach to improve the fault tolerance and resilience of a system is run-time monitoring. AUTOSAR, the emerging standard for automotive software systems, specifies several run-time monitoring mechanisms at the watchdog and OS level that are neither intended, nor able to support complex run-time monitoring. This paper addresses the general challenges involved in the development and integration of a model-based generation process of complex run-time security and safety monitors. A previously published model-based development process for run-time monitors based on a special kind of Petri nets is enhanced and tailored to fit seamlessly into the AUTOSAR development process. In our evaluation, we show that efficient monitors for AUTOSAR can be directly modeled and generated from the corresponding AUTOSAR system model.

Keywords: AUTOSAR, extended live sequence charts, model-based, monitor petri nets, run-time monitoring, signatures.

1 Introduction

Embedded systems are becoming increasingly interconnected and can no longer be considered as being separated from the outside world. A prominent example are multimedia systems in the automotive domain that are connected to the internet without being totally separated from safety-critical components. Many systems have been developed as closed systems and often little attention has been paid to security mechanisms such as encryption and safe component design to deal with errors and attacks. Even modern networks in the automotive domain are vulnerable to passive and active attacks [7] and protocols such as the CAN bus protocol have been identified as a major security drawback [9]. This makes it necessary to secure safety-critical components and their communication, even if they are not directly accessible. To secure these systems, Papadimitratos et al. [13] propose a secured communication and demand a secure architecture.

P. Van Gorp, T. Ritter, and L.M. Rose (Eds.): ECMFA 2013, LNCS 7949, pp. 70–85, 2013.
© Springer-Verlag Berlin Heidelberg 2013

However, in the majority of cases, even such efforts cannot eliminate all security vulnerabilities that can lead to safety threats as it is impossible to foresee all attacks during development. Moreover, it is often economically or technically infeasible to secure existing systems retroactively against external adversaries. Hence, systems and especially electronic control units in cars cannot be considered as secure against attacks, either caused by unknown vulnerabilities or by the required integration of legacy components. To cope with these security and safety issues, run-time monitoring is a feasible approach to detect attacks that exploit previously unknown errors and security vulnerabilities [11].

The AUTOSAR (AUTomotive Open System ARchitecture) platform[1] is emerging in the automotive domain as an open industry standard for the development of in-vehicular systems. To cope with the growing complexity of modern vehicular systems, it provides a modular software architecture with standardized interfaces and a run-time environment (RTE) that separates application level software components from the underlying basic software modules and the actual hardware. AUTOSAR offers a clearly structured system specification, which is stored in the standardized AUTOSAR XML (ARXML) format. The AUTOSAR development process supports the monitoring of control flow and timing properties at a low abstraction level [1,2], but does not provide support for modeling complex monitoring functionality at the software component level.

For this purpose, we have adopted our generic Model-based Security/Safety Monitor (MBSecMon) development tool chain [15]. It is based on the Model-Driven Development (MDD) concept that supports the generation of monitors from signatures describing the interactions between the components of a system. The MBSecMon specification language (MBSecMonSL) is based on Live Sequence Charts (LSC) introduced by Damm and Harel [5], which have been extended [15] for the modeling of behavioral signatures. A specification based on these extended LSCs (eLSC) and a structural description of the system constitutes the input set of the MBSecMon process. The signatures are divided in intended system behavior (use cases) and known attack patterns and attack classes (misuse cases). These signatures are automatically transformed to a formally defined Petri net language, the Monitor Petri nets (MPNs) [14], which serve as a more explicit, intermediate representation. With the provision of platform-specific information, security/safety run-time monitors are automatically generated for different target platforms.

The contribution of this paper is the development of a methodology for the model-based development of complex run-time monitors for AUTOSAR that operate directly on the AUTOSAR system model. We depict the challenges that arise during the development and integration of a model-based monitor generation framework into the AUTOSAR development process and present solutions to each. In summary, these challenges are as follows.

C1 Integrating existing AUTOSAR development fragments into the monitor generation process.

[1] AUTOSAR: `http://www.autosar.org`

C2 Providing type safety during the whole modeling and generation process of the monitors.
C3 Modeling monitor signatures on the same abstraction level as the AUTOSAR system models.
C4 Mapping of the abstract signature descriptions to platform specific monitoring code.
C5 Providing communication data of the generated AUTOSAR software components (SW-C) to the monitors.
C6 Supporting the relocatability of software components by generating distributed monitors from global signatures.
C7 Generating monitors with a minimal overhead for the control units.

The remainder of the paper is structured as follows: Section 2 gives an overview of existing approaches for monitoring and instrumentation and describes how they can be applied in the AUTOSAR process. In Sect. 3, the challenges are described in detail, and solutions are presented based on the adaptation of the MBSecMon process for the AUTOSAR development process. The generated monitors that are connected to the example system are evaluated in Sect. 4. In Sect. 5, a conclusion is drawn, and possible future work is discussed.

2 Related Work

The AUTOSAR standard specifies several run-time monitoring mechanisms that are provided by the Operating System (OS) and the Watchdog Manager (WdM). In the *Specification of Operating System* [1], three monitoring mechanisms for the detection of timing faults at run-time, i.e., tasks or interrupts missing their deadline, are considered. *Execution time protection* monitors the execution budget of tasks and category 2 Interrupt Service Routines (ISRs), in order to guarantee that the execution time does not exceed a statically configured upper bound. *Locking time protection* supervises the time that a task or category 2 ISR can hold a resource, including the suspension time of OS interrupts or all interrupts of the system. This is done to prevent priority inversions and to recover from potential deadlocks. The third monitoring mechanism is *inter-arrival time protection*, which controls the time between successive activations of tasks and the arrival of category 2 ISRs. These mechanisms monitor the system at the task level and are not suited to implement control flow or data flow monitoring. The configuration is done at the OS level and does not factor the system view of the model.

The *Specification of Watchdog Manager* [2] provides three monitoring mechanisms that are complementary to those offered by the AUTOSAR OS. All of the implemented mechanisms are based on checkpoints that report to the watchdog manager (WdM) when they are reached. Supervised functions have to be instrumented with calls to the watchdog, which verifies at run-time the correct transition between two checkpoints as well as the timing of the checkpoint transitions. For *alive supervision*, the WdM periodically checks if the checkpoints of a supervised entity have been reached within the given limits, in order to detect

if a supervised entity is run too frequently or too rarely. For the supervision of aperiodic or episodical events that have individual constraints on the timing between checkpoints, the WdM offers *deadline supervision*. In this approach, the WdM checks if the execution time of a given block of a supervised entity is within the configured minimum and maximum bound. The third mechanism that the WdM provides is *logical monitoring*, which focuses on the detection of control flow errors, which occur if one or more program instructions are processed either in an incorrect sequence or are not processed at all. This approach resembles the work of Oh et al. [12], in which a graph representation of a function is constructed by dividing the function into *basic blocks* at each (conditional) branch. Basic blocks are represented by nodes, whereas legal branches are represented by arcs that connect the nodes. Whenever a new basic block is entered, the monitor verifies that the taken branch was legal, according to the graph representation.

Except for control flow monitoring, the monitoring mechanisms that AUTOSAR currently specifies are only suitable for monitoring timing properties at a low level of abstraction. None of the approaches is configured on the level of the AUTOSAR system model, which offers the developer an intuitive and integrated view on the system. Apart from the monitoring services offered by AUTOSAR, few research has covered this area so far. One exception are two articles by Cotard et al. [3,4], in which the authors describe a monitoring approach for the synchronization of tasks on multi-core hardware platforms. In their approach, dependencies between tasks are first modeled as a finite state machine (FSM). The FSM is then translated into a linear temporal logic (LTL) specification, from which Moore machines and, subsequently, C code is generated. Their primary focus is on synchronization and not on control or data flow monitoring.

In our MBSecMon process, an extended version of the expressive and compact Live Sequence Charts is used as signature specification language. Kumar et al. [10] specifies protocols with LSCs and transforms them for verification to temporal logic formulas. Thereby, complex LSC specifications lead to an explosion of the formula and are, therefore, with LSCs as specification language not suitable as intermediate language for the code generation for embedded systems.

Besides these transformations to LTL, there are some approaches that use Petri nets directly for the specification of monitors. Frankowiak et al. [6] uses Petri nets to specify low cost process monitors on a micro controller. He enhances regular Petri nets by token generators and special end places (bins). Additionally, subnets are linked by a control net. For complex monitor specifications, these nets get relatively large compared to MPNs, whose semantics include an implicit evaluation logic when an event does not match to the specified sequence. The MPNs are tailored to describe monitor specifications in an explicit but nevertheless compact form.

3 The MBSecMon Framework

The Model-based Security/Safety Monitor (MBSecMon) Framework [15] has been developed as a generic approach to build tool chains for monitor generation.

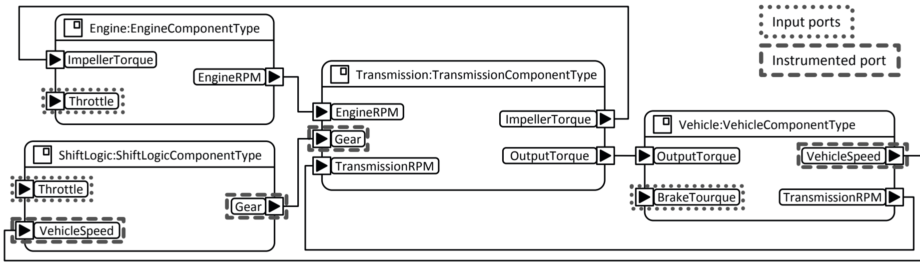


Fig. 1. AUTOSAR example model of an automatic transmission

In this section, we show how this framework has been tailored to fit seamlessly in a model-based AUTOSAR development process and solve the challenges that are raised in Sect. 1. This overcomes the current lack of model-based monitoring support in AUTOSAR tool chains that we attested in Sect. 2.

3.1 Example: Automatic Transmission Controller

The example used throughout this paper focuses on monitoring the communication between AUTOSAR software components (SW-Cs). The AUTOSAR system is modeled with the tool OptXware Embedded Architect[2] and the implementation of the subsystems is generated by the AUTOSAR code generation of MathWork's Embedded Coder plugin for Simulink.

The implementation of the example system is based on the *Automatic Transition Controller* demo project [18], shipped with Matlab/Simulink. As depicted in the system model in Fig. 1, this model describes the internal behavior of three application-level software components, *ShiftLogic*, *Transmission*, and *Engine*, based on the input values *Throttle* and *BrakeTorque* and the interaction between these components. Influences such as aerodynamics and drag friction of the wheels are represented by the *Vehicle* block. To comply with the needs of the AUTOSAR code generation provided by Embedded Coder, the behavioral model has been adapted by replacing all continuous with discrete blocks. The generated code serves as the behavioral implementation of the skeleton generated by the AUTOSAR tool OptXware Embedded Architect. For the asynchronous communication between the components, the sender-receiver communication pattern is used.

3.2 The Tailored Monitor Generation Process for AUTOSAR

The generic MBSecMon development tool chain has been extended to enable a seamless integration into the AUTOSAR development process. Figure 2 depicts on the left side the original simplified AUTOSAR development process, starting

[2] OptXware: `http://www.optxware.com`

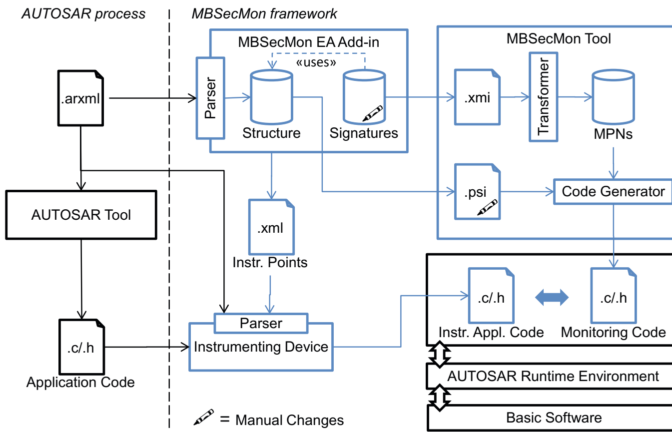


Fig. 2. MBSecMon framework embedded in AUTOSAR development process

with the system structure description persisted in the AUTOSAR XML format (ARXML). This file is used by the AUTOSAR tool chain to generate the RTE and a code skeleton for the implementation of the SW-Cs, which is supplied by the Simulink code generator. The final code is usable in an AUTOSAR simulation environment or directly on the Electronic Control Unit (ECU) of a vehicle.

On the right-hand side of Fig. 2, the framework is depicted that embeds the monitor generation tool chain in the AUTOSAR development process. The specification of monitor signatures is achieved with the help of a tailored version of the UML2 modeling tool Enterprise Architect[3] (EA). This tool has been extended by an add-in that allows the modeling of eLSCs (*Signatures*) depending on the imported component diagram view (*Structure*) of the AUTOSAR system.The modeled signatures are then exported together with additional platform specific information (PSI) extracted from the imported AUTOSAR model. Additionally, the *MBSecMon add-in* analyses the modeled signatures for needed *instrumentation points* in the AUTOSAR application code and persists this information in an XML file. Through a graph-based model-to-model transformation [16], the exported representations of the signatures are transformed in the formally defined Monitor Petri nets (MPNs) [14] that serve as an intermediate language used for a straight-forward code generation. The code generator translates the signatures, represented as MPNs, incorporating the additional information (PSI), to monitoring code. This monitor is stimulated by calls of its interfaces. Therefore, the *instrumenting device* uses the AUTOSAR system specification, together with the *instrumentation points* file, to instrument the application code. The instrumentation is realized via interface wrappers, which intercept the communication between two components. A detailed description of the instrumentation of the interfaces of AUTOSAR components with wrappers is given in [17].

[3] SparxSystems Enterprise Architect: `http://www.sparxsystems.com`

This process allows for specifying and automatically generating monitors on an abstract level based on the system information provided by the AUTOSAR development process. Only the specification for the monitors has to be modeled using the MBSecMon add-in and the platform specific information has to be extended by the system engineer. In the following, based on the example in Sect. 3.1, this process and the necessary adaptation based on the identified challenges are described in detail.

Challenge 1: Integrating Existing Development Fragments. In the AUTOSAR development process, the system structure is modeled at a high abstraction level, describing the software components (SW-C), their ports with specified data types, and the connections between the ports. The generated monitors should observe the communication between these SW-Cs. Therefore, a component view of the AUTOSAR system should be used to support the modeling of signatures.

MBSecMon process: The MBSecMon EA add-in incorporates an ARXML parser that imports the AUTOSAR software component structure, which has been modeled in an AUTOSAR system-level editing tool, such as OptXware Embedded Architect. Derived from this data, the add-in creates an UML component diagram as shown in Fig. 3 that includes the components, the ports and a connector representation based on the AUTOSAR naming scheme. Additional system information such as the names of the connectors and ports are stored as tagged values in the model elements and are abbreviated in the diagram view for a clear presentation.

Example: The software components in Fig. 3 comply to the blocks of the Simulink model used to generate the implementation of the application code. This model is used in the MBSecMon development process to describe the allowed and forbidden communication sequences between the components as signatures.

Challenge 2: Providing Type Safety. In the AUTOSAR development process and in the automotive domain, specialized data types are used. This allows for limiting the value range of these data types and prevents wrong assignments such as storing a speed value in a variable intended for the impeller torque. Generated monitors have to obey this implicit safety mechanism that is inherent in the AUTOSAR standard.

MBSecMon process: The special data types provided by the AUTOSAR system specification are imported along with the component diagrams and stored in its ports. While modeling the signatures, the developer's choice is constrained to these data types. These are further used in the monitor generation process and result in type safe monitoring code.

Example: In the ARXML file the data type for the vehicle speed, shown in Listing 1.1, is defined as *VehicleSpeedDataType* with a base type *Double* and is limited to a range of possible values.

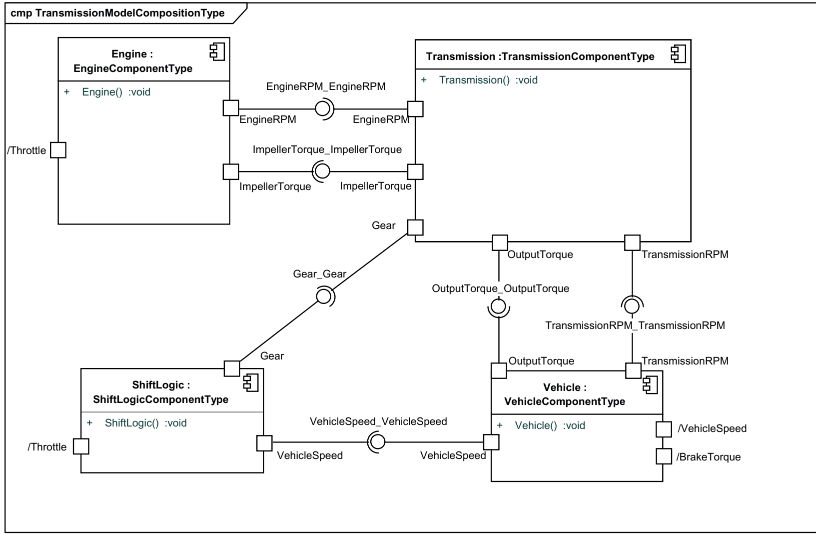


Fig. 3. UML component diagram of the system imported to EA

Listing 1.1. Type safety in AUTOSAR (ARXML file)

```
<REAL-TYPE >
  <SHORT-NAME>VehicleSpeedDataType</SHORT-NAME>
  <LOWER-LIMIT INTERVAL-TYPE="CLOSED" >-1000.0</LOWER-LIMIT>
  <UPPER-LIMIT INTERVAL-TYPE="CLOSED" >1000.0</UPPER-LIMIT>
  <ALLOW-NAN>false</ALLOW-NAN>
  <ENCODING>DOUBLE</ENCODING>
</REAL-TYPE>
```

Challenge 3: Modeling at the Same Abstraction Level. AUTOSAR system specifications are modeled at a high abstraction level as presented in Challenge 1. The ports describe which type of communication is used to interact with other SW-Cs over the Virtual Function Bus (VFB). Thus, it makes sense to monitor the communication between the SW-Cs. A widespread approach to describe interactions between components are various kinds of sequence charts that are on the same abstraction level as the AUTOSAR specification. These descriptions of the interaction between SW-Cs (signatures) can be used to generate monitors.

MBSecMon process: For this purpose, the MBSecMon specification language (MBSecMonSL), which consists of extended Live Sequence Charts (eLSC) that are structured by use/misuse cases (UC/MUC), is used in the MBSecMon framework. In addition to the concepts of the wide-spread Message Sequence Charts [8], eLSCs distinguish between hot (solid red border) and cold (dashed blue border) elements, where hot elements are mandatory and cold elements are optional. Furthermore, two forms of eLSCs exist, an universal eLSC with a prechart (precondition) (blue dashed hexagon) before the mainchart (solid black rectangle) and an existential eLSC without a precondition. In the prechart, every element

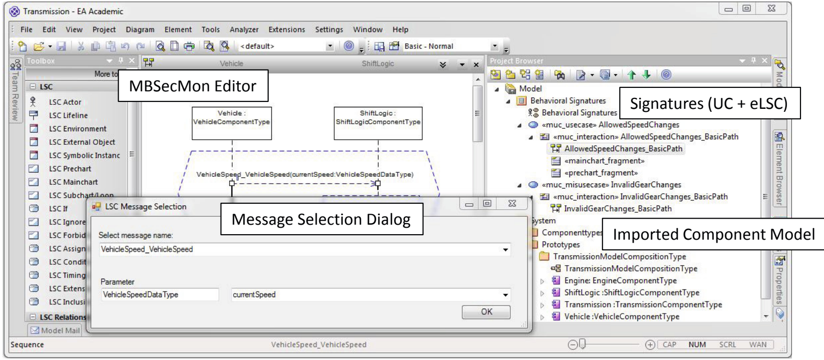


Fig. 4. Tailored Signature Editor for AUTOSAR

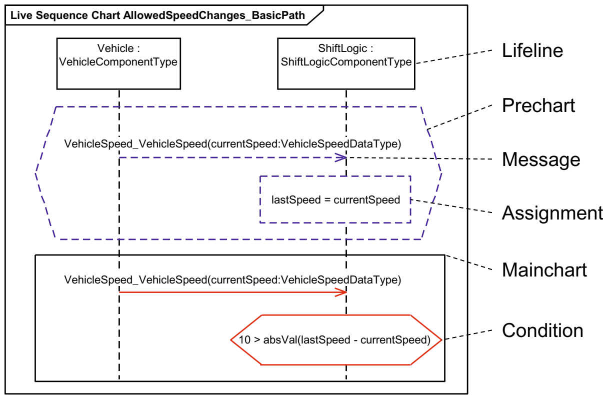


Fig. 5. Signature of the Use Case “AllowedSpeedChanges”

is interpreted as mandatory. This is based on the interpretation of the prechart as a precondition. When the prechart is fulfilled, the part of the signature in the mainchart is evaluated.

The import of the component view from an AUTOSAR tool to EA forms the basis for the signature modeling. The MBSecMon EA add-in (Fig. 4) provides a context sensitive choice of messages between components and their parameter types. This ensures the compliance to the modeled AUTOSAR system.

Example: Figure 5 shows a simple example of a concurrent signature that uses only the basic elements of the eLSC language. This signature monitors the communication between the *Vehicle* and the *ShiftLogic* components by initializing the monitor for every message, consisting of a sending and a receiving event, transmitted over the port *VehicleSpeed*. The message contains the value *currentSpeed* that is stored by the assignment to an eLSC specific variable *lastSpeed*. The first processed sending event triggers the initialization of a new instance of

Listing 1.2. LSC specific information in the PSI file

```
<entry key="AllowedSpeedChanges_BasicPath.context_method.absVal">fabs
</entry>
```

the signature that concurrently monitors the next messages. The next *VehicleSpeed* message is processed by the mainchart of the first instance and evaluated by the condition to the previous value stored in *lastSpeed*. This monitoring instance is then terminated based on the result of the condition. Subsequently, the same message is evaluated by the prechart of the second instance of the signature and overwrites the variable *lastSpeed* with the new value.

Challenge 4: Mapping to Platform Specific Monitoring Code. By modeling signatures on a much more abstract level than the code of the target platform and using an annotation language in the signatures, the code generator needs additional information about the mapping to the target platform.

MBSecMon process: To support type safe, AUTOSAR-compliant interfaces for the monitor, additional mappings are generated to the platform specific information (PSI) file. It contains mappings between AUTOSAR instrumentation interface and the internal events of the monitor, data types for transferred values, a code mapping of signature annotations to the target domain, and configuration details for the code generation. Most information for the file can be automatically derived from the imported system specification and the modeled signatures. Only mappings exported by the EA add-in from annotated pseudo code in the signatures have to be adapted manually to the target language. For more convenient usage, a mapping library for these annotations could automate this manual step.

Example: The generated PSI file provides the mapping of pseudo code used in the signatures to platform-conform code fragments, as depicted in Listing 1.2 for a method *absVal* in the condition of the signature in Fig. 5.

Challenge 5: Providing Communication Data to the Monitors. The system model of the AUTOSAR specification uses the concept of the Virtual Function Bus (VFB) and only describes the communication pattern (e.g. sender/ receiver). This communication is realized on the target platform depending on the RTE and the allocation of the SW-Cs to the different ECUs. This hampers the monitoring of the communication between the SW-Cs.

MBSecMon process: Based on the signatures, the *instrumenting device* needs a clear naming convention for the monitor interfaces to allow for the wrapping of write and read methods in the AUTOSAR system code. Additionally, the *instrumenting device* uses the exported information that specifies, based on the signatures, which ports in the AUTOSAR system have to be instrumented to minimize the footprint of the instrumentation by only adding method calls for the required events.

Listing 1.3. Monitor interfaces for the running example

```
void dispatchEvent_TransmissionModelCompositionType_Vehicle_
  VehicleSpeed(VehicleSpeedDataType*);
void dispatchEvent_TransmissionModelCompositionType_ShiftLogic_
  VehicleSpeed(VehicleSpeedDataType*);
```

Example: The interfaces of the monitor are named based on the naming conventions of the AUTOSAR standard. Thus, the *instrumenting device* can automatically generate calls to the interfaces into the system code. Listing 1.3 shows the interfaces of the generated example monitor. The method name is derived by concatenation of the string `dispatchEvent_` and the full name of the port of the AUTOSAR component and the data type of the parameter origins from the AUTOSAR model.

Challenge 6: Supporting the Relocatability of Software Components. One of the important concepts of the AUTOSAR standard is the relocatability of SW-Cs that allows for distributing them to different ECUs without changing the specification. Due to modeling the monitors on the same abstraction level as the AUTOSAR system specification, the developer of the signatures cannot incorporate information about the final distribution of the SW-Cs. The code generation process must support the generation of distributed monitors based on the actual distribution of the SW-Cs. This reduces the run-time overhead for single ECUs and the communication overhead over the buses between the ECUs in contrast to a central monitor.

MBSecMon process: In the MBSecMon generation process, the signatures modeled as eLSCs already contain an affiliation of the events (sending and receiving) to the SW-Cs. This affiliation is obtained when the exported signatures are transformed to the intermediate Monitor Petri nets (MPN), and allow for generating distributed monitors based on the SW-C located on the same ECU. To preserve dependencies between the part monitors (e.g. sending before receiving event of a message), an additional communication between the monitors has to be established. Therefore, the code generator has been prepared to create identifiers that can be replaced by macro definitions to system communication methods.

Example: In the example, we use the sender-receiver pattern that directly supports transferability and exchange of AUTOSAR software components. The two components, *ShiftLogic* and *Vehicle*, in the signature in Fig. 5 can be distributed to different ECUs by defining "ShiftLogic;{Vehicle}" in the PSI file the instance that is on the same ECU (*ShiftLogic*) and the instances (*Vehicle*) that is located on another ECU and should be synchronized with it.

Challenge 7: Generating Monitors with a Minimal Overhead. The generated run-time monitors are deployed on the ECUs and run alongside the SW-Cs. Hence, their induced run-time and memory overhead on the ECUs have

to be sufficiently small. The reasonable overhead depends on the required level of safety and security that has to be reached by monitoring. For the runtime-overhead, a worst case upper bound can be calculated by a static analysis of the signatures in the MPN format.

Example: In the presented example in Figure 5 the monitor has to evaluate two transitions per monitor instance (sending or receiving events) in one event processing step. Additionally, to this computation the annotated assignment or condition has to be taken into account.

The evaluation in Sect. 4 shows the overhead that the resulting monitors introduce to the system.

4 Evaluation

For the evaluation of the MBSecMon specification and generation process for AUTOSAR, an adaptation of the Simulink example model *Automatic Transmission Controller* [18], as presented in Sect. 3.1, is used. In order to provide compatibility with Simulink's AUTOSAR code generator, which does not support the continuous blocks (such as integrators) that were used in the original example, we decomposed the example model into separate software components and replaced the incompatible blocks with their corresponding discrete versions. The generated code contains the runnables (executable entities) of each software component (SW-C), which have to be integrated into the implementation code skeleton that was generated by the system level design and simulation tool OptXware Embedded Architect (OXEA). In OXEA, we have also designed the system model that corresponds to the Simulink example, and which is stored in the standardized ARXML format.

We instrumented the evaluation system with two different monitors (cf. Fig. 1). The first one is *AllowedSpeedChanges*, as presented in Sect. 3, which monitors the communication between the components *Vehicle* and *ShiftLogic* in order to detect a communication error between these components. The monitor signals an error in case that it detects a difference between two consecutive speed readings that is larger than 10 mph within a 20 ms timeframe (the period of the tasks). The second one is *InvalidGearChanges*, which, in contrast to the first signature, monitors the misuse case of a gear shifting by more than one step within a 20 ms timeframe. Therefore, the communication between the components *ShiftLogic* and *Transmission* is monitored. Based on these two signatures, a monitor that consists of a controller and a monitor representation for each signature is generated by the MBSecMon process.

The evaluation covers the run-time overhead that the monitoring induces per instrumented port (*Gear* for ShiftLogic and Transmission, and *VehicleSpeed* for Vehicle and ShiftLogic), and per instrumented runnable (Vehicle, ShiftLogic, Transmission). Furthermore, we analyze the memory overhead of the monitors, for both, code and data segments. Finally, a scalability analysis on an embedded system is performed.

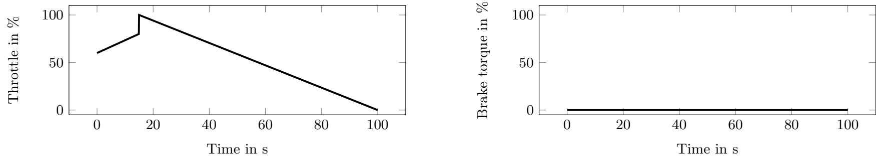


Fig. 6. Input data set for the passing maneuver

Table 1. Execution time comparison of original and monitored calls to the RTE

		Average		
Component	**RTE Call**	*Original (ticks)*	*Instr. (ticks)*	*Diff. (%)*
ShiftLogic()	Rte_Read_VehicleSpeed ...	24,76	38,87	57
ShiftLogic()	Rte_Write_Gear ...	18,00	23,60	31
Transmission()	Rte_Read_Gear ...	25,28	37,63	49
Vehicle()	Rte_Write_VehicleSpeed ...	21,56	27,62	28

Run-Time Analysis. The evaluation was conducted using OptXware EA's simulation environment on a AMD Phenom II X4 955 processor, running at 3.20 GHz. The timing measurements were taken using the Win32 API functions *MyQueryPerformanceCounter* and *MyQueryPerformanceFrequency*, which are Window's high resolution timing functions, providing CPU tick granularity. This is a best effort solution, as there is no commonly agreed on reference architecture for such evaluation. Therefore, we also provide relative measurements of the execution time overhead as comparison.

Figure 6 shows the input data (readings of the throttle and brake torque sensor) for the test run of the automatic transmission model. It represents a passing maneuver, where the vehicle approaches a slower car and then abruptly accelerates to pass the car.

Table 1 shows the run-time for performing calls to the RTE in the simulation environment without (*Original*) and with instrumentation (*Instr.*). The measured time for the instrumented RTE calls includes the wrapper around the call method, the invocation of the monitor, and the evaluation of the event by the monitor. The overhead is between 28 and 31% for write calls and between 49 and 57% for read calls. This difference results from the structure of the signatures that include an additional assignment or condition in the monitor for the transmitted value at the receiving side (read call).

Table 2 shows the influence of the monitor on the total run-time of the SW-C's runnables. For the ShiftLogic component, two ports have been wrapped and, therefore, the instrumented runnable calls the monitor twice as often as for the other components. The components Transmission and Vehicle, which contain only one instrumented port, therefore, have a smaller run-time overhead.

Memory Overhead. We conducted our analysis of the memory overhead using the tool *objdump* of the GNU binutils toolsuite on the compiled object files. Objdump provides a detailed overview of the memory consumption of the text

Table 2. Execution time comparison of original and monitored runnables

	Average			
Component	*Original (ticks)*	*Instr. (ticks)*	*Diff. (ticks)*	*Diff. (%)*
ShiftLogic()	56,03	70,97	14,94	28
Transmission()	57,28	63,77	6,49	11
Vehicle()	48,63	58,39	9,76	20

Table 3. Memory overhead caused by the monitors in byte

	Wrapper				Monitor		
Type	*Read_VS*	*Write_VS*	*Read_G*	*Write_G*	*Contr.*	*ASC*	*IGC*
Code	48	48	48	48	1632	2512	2160
Data	0	0	0	0	88	40	32

(aka. code) section, and the three data sections data, bss (uninitialized data) and rdata (read-only data). The analysis of the memory overhead of the integrated monitors is depicted in Tab. 3. The first four columns show the memory that is consumed by the RTE call wrappers that pass the signals to the monitor. As the functionality of each wrapper is similar, their overhead is constant.

The monitor component consists of a controller (*Contr.*) that manages the monitors and the signatures AllowedSpeedChanges (*ASC*) and InvalidGearChanges (*IGC*). The controller caches the transmitted values, triggers the signature representations, and evaluates their results. The memory overhead of the data section is very small and grows very slowly with the complexity of the signatures because only the state of the monitor and the monitor specific variables, as shown in Sect. 3.1, are stored there.

Scalability Analysis. The previous results have shown that the generated monitors can be used in an AUTOSAR environment with reasonable overhead. For the evaluation of the scalability of the monitors, various models of different complexity are generated and the run-time behaviour of the generated C code is measured on a Fujitsu SK-16FX-100PMC evaluation board equipped with an F^2MC-16FX MB96340 series microcontroller (16 bit, 56 MHz). Table 4 shows the different models, the run-time of the generated monitors for processing one event in the signature, where a message consists of a sending and receiving event, and the needed memory on the micro controller. The values for the needed data memory (RAM) include approximately 800 bytes of data that do not origin from the generated monitor. For the measurement of the run-time, 1000 complete runs of the signatures have been performed. Complex conditions and actions in the signature are dismissed and only the monitor itself is measured.

Model 1 to 6 demonstrate how the monitors scale if the number of messages increases in the signature. Model 1 and 7 to 11 show the overhead when all messages have the same message type. In Model 4 and 12 to 14, the number of signatures of constant size (M4) is increased. With an increasing number of processing steps to reach the final state of the signature the initialization

Table 4. Results of the scalability evaluation

Model	*#Messages*	*Different events*	*Run-time/Event in µs*	*Code in byte*	*Data in byte*
M1	1	2	28,384	896	1038
M2	2	4	25,360	1078	1044
M3	3	6	24,277	1230	1044
M4	4	8	23,728	1382	1044
M5	50	100	22,636	8898	1154
M6	100	200	22,660	17026	1268
M7	2	2	25,936	1043	1044
M8	3	2	25,099	1158	1044
M9	4	2	25,176	1562	1044
M10	50	2	46,239	7260	1156
M11	100	2	67,973	13605	1270
M12	2 * M4	8	46,424	2696	1056
M13	3 * M4	8	67,992	4000	1068
M14	4 * M4	8	89,744	5304	1080

overhead gets less important. The code memory consumption increases linearly with the number of messages located in a signature. In all cases, the RAM needed to store the state of the signature increases very slowly, because the state of the signature is binary coded. This is an important factor for use on resource constrained embedded systems. The evaluation shows that the monitors have a constant computing time per processed event (M1 to M6) and only a linear increase for processing an event that is annotated at multiple transitions (M1, M7 to M11).

5 Conclusion and Future Work

In this paper, we have identified and addressed the challenges that emerge from integrating run-time monitoring of complex signatures into the AUTOSAR development process. The presented continuous model-based development process for security and safety monitors (MBSecMon) is integrated into the AUTOSAR process and incorporates data of the AUTOSAR models. As shown, it allows the modeling of monitor signatures in a well comprehensible graphical modeling language and the automatic generation of monitors with a low overhead that fulfill the AUTOSAR conventions. This framework and the generated monitors have been evaluated utilizing an example model provided with the MATLAB suite, for which AUTOSAR code was generated and integrated into an AUTOSAR environment. With the presented approach, we have overcome the lack of support for complex monitoring in the AUTOSAR tool chains. It is applicable to white-box (source code) and black-box (binary) components, as communication between components is intercepted directly at their port interface by instrumentation techniques shown in [17].

In the future, an evaluation in a larger AUTOSAR project with more complex interactions is planned. It has to be evaluated if the MBSecMon process can be used for Logical Program Flow Monitoring [2], eventually with another source specification language such as UML2 activity diagrams or MPNs directly.

Acknowledgements. This work was supported by CASED (www.cased.de).

References

1. AUTOSAR: Specification of Operating System (2011), `http://www.autosar.org/download/R4.0/AUTOSAR_SWS_OS.pdf`
2. AUTOSAR: Specification of Watchdog Manager (2011), `http://www.autosar.org/download/R4.0/AUTOSAR_SWS_WatchdogManager.pdf`
3. Cotard, S., Faucou, S., Bechennec, J.L., Queudet, A., Trinquet, Y.: A Data Flow Monitoring Service Based on Runtime Verification for AUTOSAR. In: IEEE 14th International Conference on HPCC-ICESS 2012, pp. 1508–1515 (2012)
4. Cotard, S., Faucou, S., Béchennec, J.: A Dataflow Monitoring Service Based on Runtime Verification for AUTOSAR OS: Implementation and Performances. OSPERT pp. 46–55 (2012)
5. Damm, W., Harel, D.: LSCs: Breathing Life into Message Sequence Charts. Formal Methods in System Design 19(1), 45–80 (2001)
6. Frankowiak, M.R., Grosvenor, R.I., Prickett, P.W.: Microcontroller-Based Process Monitoring Using Petri-Nets. EURASIP Journal on Embedded Systems 2009, 3:1–3:12 (2009)
7. Groll, A., Ruland, C.: Secure and Authentic Communication on Existing In-Vehicle Networks. In: Intelligent Vehicles Symposium, pp. 1093–1097. IEEE (2009)
8. Harel, D., Thiagarajan, P.: Message Sequence Charts. In: Lavagno, L., Martin, G., Selic, B. (eds.) UML for Real, pp. 77–105. Springer (2004)
9. Koscher, K., Czeskis, A., et al.: Experimental Security Analysis of a Modern Automobile. In: IEEE Symposium on SP, pp. 447–462 (2010)
10. Kumar, R., Mercer, E., Bunker, A.: Improving Translation of Live Sequence Charts to Temporal Logic. ENTCS 250(1), 137–152 (2009)
11. Kumar, S.: Classification and Detection of computer Intrusions. Ph.D. thesis, Purdue University (1995)
12. Oh, N., Shirvani, P., McCluskey, E.: Control-flow Checking by Software Signatures. IEEE Transactions on Reliability 51(1), 111–122 (2002)
13. Papadimitratos, P., Buttyan, L., et al.: Secure Vehicular Communication Systems: Design and Architecture. IEEE Communications Magazine 46(11), 100–109 (2008)
14. Patzina, L., Patzina, S., Piper, T., Schürr, A.: Monitor Petri Nets for Security Monitoring. In: Proc. of 1st S&D4RCES, pp. 3:1–3:6. ACM (2010)
15. Patzina, S., Patzina, L., Schürr, A.: Extending LSCs for Behavioral Signature Modeling. In: Camenisch, J., Fischer-Hübner, S., Murayama, Y., Portmann, A., Rieder, C. (eds.) SEC 2011. IFIP AICT, vol. 354, pp. 293–304. Springer, Heidelberg (2011)
16. Patzina, S., Patzina, L.: A Case Study Based Comparison of ATL and SDM. In: Schürr, A., Varró, D., Varró, G. (eds.) AGTIVE 2011. LNCS, vol. 7233, pp. 210–221. Springer, Heidelberg (2012)
17. Piper, T., Winter, S., Manns, P., Suri, N.: Instrumenting AUTOSAR for Dependability Assessment: A Guidance Framework. In: 42nd Annual IEEE/IFIP International Conference on DSN, pp. 1–12. IEEE (2012)
18. The MathWorks, Inc.: Modeling an Automatic Transmission Controller (2012), `http://www.mathworks.de/de/help/simulink/examples/modeling-an-automatic-transmission-controller.html` (visited on Feburary 12, 2013)

End-User Support for Debugging Demonstration-Based Model Transformation Execution

Yu Sun[1] and Jeff Gray[2]

[1] University of Alabama at Birmingham, Birmingham AL 35294
yusun@cis.uab.edu
[2] University of Alabama, Tuscaloosa, AL 35401
gray@cs.ua.edu

Abstract. Model Transformation By Demonstration (MTBD) has been developed as an approach that supports model transformation by end-users and domain experts. MTBD infers and generates executable transformation patterns from user demonstrations and refinement from a higher level of abstraction than traditional model transformation languages. However, not every transformation pattern is demonstrated and specified correctly. Similar to writing programs, bugs can also occur during a user demonstration and refinement process, which may transform models into undesired states if left unresolved. This paper presents MTBD Debugger, which is a model transformation debugger based on the MTBD execution engine, enabling users to step through the transformation execution process and track the model's state during a transformation. Sharing the same goal of MTBD, the MTBD Debugger also focuses on end-user participation, so the low-level execution information is hidden during the debugging process.

Keywords: Model Transformation By Demonstration (MTBD), Model Transformation Debug, End-User Programming.

1 Introduction

Model transformation plays an essential role in many applications of Model-Driven Engineering (MDE) [2]. Although a number of model transformation languages (MTLs) have been developed to support various types of model transformation tasks [1], some innovative model transformation approaches and tools have also been introduced to address the complexity of learning and using MTLs, and the challenges of understanding metamodels [16]. Our earlier work on Model Transformation By Demonstration (MTBD) [5], which was influenced by the idea of Model Transformation By Example (MTBE) [3][4][7], enables users to demonstrate how a model transformation should be performed by editing the model instance directly to simulate the model transformation process step-by-step. A recording and inference engine has been developed to capture all user operations and infer a user's intention in a model transformation task. A transformation pattern is generated from the inference, specifying the precondition of the transformation and the sequence of operations

P. Van Gorp, T. Ritter, and L.M. Rose (Eds.): ECMFA 2013, LNCS 7949, pp. 86–100, 2013.
© Springer-Verlag Berlin Heidelberg 2013

needed to realize the transformation. This pattern can be further refined by users and then executed by automatically matching the precondition in a new model instance and replaying the necessary operations to simulate the model transformation process. This was the focus of our earlier MODELS paper [5].

Using MTBD, users are enabled to specify model transformations without the need to use a MTL. Furthermore, an end-user can describe a desired transformation task without detailed understanding of a specific metamodel. We have applied MTBD to ease the specification of different model transformation activities – model refactoring, model scalability, aspect-oriented modeling, model management and model layout [17][18].

Although the main goal of MTBD is to avoid the steep learning curve and make it end-user centric, there is not a mechanism to check or verify the correctness of the generated transformation patterns. In other words, the correctness of the final transformation pattern totally depends on the demonstration and refinement operations given by the user, and it is impossible to check automatically whether the transformation pattern accurately reflects the user's intention. In practice, this is similar to producing bugs when writing programs. It is also possible that errors will be introduced in the transformation patterns due to the incorrect operations in the demonstration or user refinement step when using MTBD. Incorrect patterns can lead to errors and transform the model into undesired states. For instance, users may perform a demonstration by editing an attribute using the value of a wrong model element; they may give preconditions that are either too restrictive or too weak; or they may forget to mark certain operations as generic (which forces the inferred transformation to be tied to a specific binding).

Obviously, an incorrect transformation pattern can cause the model to be transformed into an incorrect and undesired state or configuration, which may be observed and caught by users. However, knowing the existence of errors and bugs cannot guarantee the correct identification and their location, because MTBD hides all the low-level and metamodel information from users. Also, the final generated pattern is invisible to the end-users, which makes it challenging to map the errors in the target model to the errors in the demonstration or refinement step. This issue becomes even more apparent when reusing an existing transformation pattern generated by a different user, such that the current users who did not create the original pattern have no idea how to locate the source of an error.

In order to enable users to track and ascertain errors in transformation patterns when using MTBD, a transformation pattern execution debugger is needed that can work together with the pattern execution engine. In fact, a number of model transformation debuggers have already been developed for different MTLs [9]. However, the main problem with these debuggers is that they work by tracking the MTL rules or codes, which is at the same level of abstraction as the MTL and therefore not appropriate for some types of end-users and domain experts. Because MTBD has already raised the level of abstraction above the general level of MTLs, the associated MTBD Debugger should be built at the same level of abstraction. Thus, the goal of the MTBD Debugger presented in this paper is to provide users with the

necessary debugging functionality without exposing them to low-level execution details or metamodels.

A brief overview of MTBD will be given in Section 2, followed by an introduction to the MTBD Debugger in Section 3. Section 4 illustrates the usage of the MTBD Debugger for different debugging purposes through several examples. Section 5 summarizes the related work and Section 6 offers concluding remarks.

2 Overview of MTBD

Figure 1 (adapted from [6]) shows the high-level overview of MTBD, which is a complete model transformation framework that allows users to specify a model transformation, as well as to execute the generated transformation pattern in any desired model instances.

The specification of a model transformation using MTBD starts with a demonstration by locating one of the correct places in the model where a transformation is to be made, and directly editing a model instance (e.g., add a new model element or connection, modify the attribute of a model element) to simulate the maintenance task (*User Demonstration*). During the demonstration, users are expected to perform operations not only on model elements and connections, but also on their attributes, so that the attribute composition can be supported. At the same time, an event listener has been developed to monitor all the operations occurring in the model editor and collect the information for each operation in sequence (*Operation Recording*). The list of recorded operations indicates how a non-functional property should be composed in the base model. After the demonstration, the engine optimizes the recorded operations to eliminate any duplicated or meaningless actions (*Operation Optimization*). With an optimized list of recorded operations, the transformation can be inferred by generalizing the behavior in the demonstration (*Pattern Inference*). Because the MTBD approach does not rely on any MTLs, we generate a transformation pattern, which summarizes the precondition of a transformation (i.e., *where* to perform a transformation) and the actions needed in a transformation (i.e., *how* to perform a transformation in this location). Users may also refine the generated transformation pattern by providing more feedback for the precondition of the desired transformation scenario from two perspectives – structure and attributes, or identifying generic operations to be executed repeatedly according to the available model elements and connections.

After the user refinement, the transformation pattern will be finalized and stored in the pattern repository for future use (*Pattern Repository*). The final patterns in the repository can be executed on any model instances. Because a pattern consists of the precondition and the transformation actions, the execution starts with matching the precondition in the new model instance and then carrying out the transformation actions on the matched locations of the model (*Pattern Execution*). The MTBD engine also validates the correctness of the models after each execution process (*Correctness Checking*). Users can choose where to execute the pattern, a sequence of patterns to execute, and the execution times (*Execution Control*). More details about MTBD beyond this summary are in [5].

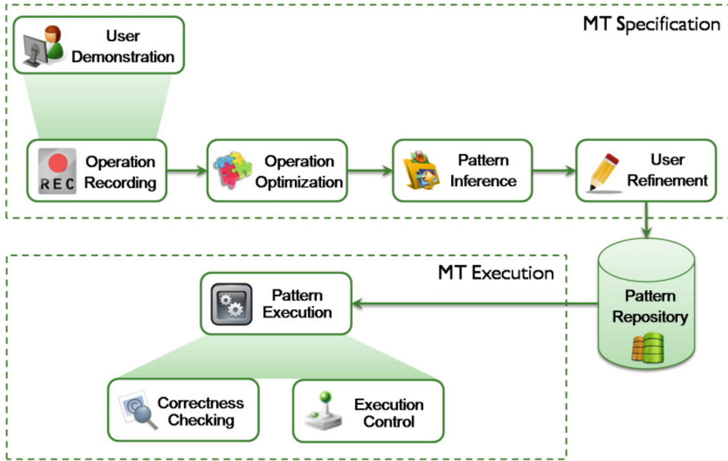


Fig. 1. High-level overview of MTBD (adapted from [6])

3 MTBD Debugger

MTBD Debugger is designed and implemented over the MTBD execution engine. The specific debugging sequence is based on the structure of a transformation pattern. As mentioned in Section 2, a transformation pattern contains the precondition of a transformation (i.e., including the structural precondition and attribute precondition) and the sequence of transformation actions. During the execution of a transformation pattern, any error that is discovered can be traced back to errors in either the precondition or the transformation actions. From the technical perspective as shown in Figure 2, the goal of MTBD Debugger is to help users correctly map the effect of a transformation exerted on the target model instance to the precondition and actions specified in the transformation pattern, so that users can track the cause of an undesired transformation result.

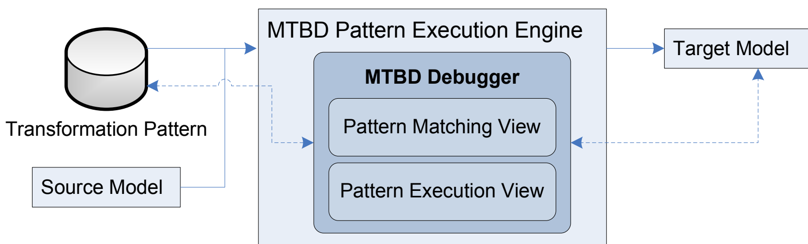


Fig. 2. Overview of MTBD Debugger

The main functionality of the MTBD Debugger is supported by enabling the step through execution of a transformation pattern and displaying the related information with each step in two views – *Pattern Execution View* and *Pattern Matching View*. Users can directly observe what action is about to be executed, what are the matched

model elements for the operation, and more importantly, how the matched elements are determined based on the types of preconditions. This allows the end-users to follow each step and check if it is the desired execution process. In addition, keeping the debugging process at the proper level of abstraction is an essential design decision of MTBD Debugger to assist end-users who are not computer scientists. Similar to MTBD, the MTBD Debugger separates users from knowing any MTLs and hides the low-level execution or metamodel details, so that the same group of users who implement model transformations using MTBD are enabled to debug the same model transformations using the language that represents their domain.

3.1 Pattern Execution View

The *Pattern Execution View* lists all the actions to be executed in a transformation pattern in sequence. As shown in a future example in Figure 5 (which is used later in a specific debugging context), the view displays the type of the action, the main target element used for this action, whether the action is generic or not, and the related details based on the type of the action. In the debugging mode, users can step through each action one-by-one. Before the execution of the action, all the matched elements that will be used for the action are highlighted in the *Pattern Matching View*, so that users can determine which elements will be used for the execution of the action. If the required target element cannot be matched, “null” will be displayed. After the action is executed, the *Pattern Execution View* highlights the next action. At the same time, the model in the editor is updated with the execution of the previous action. Users can check the properties and structure of the latest model instance and determine if it is transformed into the desired state.

3.2 Pattern Matching View

The *Pattern Matching View* works together with the *Pattern Execution View* to provide relevant information about the matched model elements. From Figure 5, it can be seen that it shows the model element type, the precondition associated with it, and the specific model element that is matched in the current model. The list includes all the model elements needed in the transformation pattern. The execution of each action will trigger the highlight of all needed model elements in this view.

4 MTBD Debugger in Action

This section presents a case study that illustrates the use of MTBD Debugger to support tracking and debugging errors in several practical model transformation tasks in a textual game application domain (for the Debugger, we use the same case study from [5] for consistent discussion for those who may refer back to the original MTBD paper).

4.1 Background: MazeGame Case Study

The case study is based on a simple modeling language called MazeGame. A model instance is shown in Figure 3. A *Maze* consists of *Rooms*, which can be connected to each other. Each *Room* can contain either pieces of *Gold*, or a power item such as *Weapon* or *Monster* with an attribute (*strength*) to indicate the power. The goal of the game is to let players move in the *rooms*, collect all pieces of *gold*, and use *weapons* to kill *monsters*. The full Java implementation of the game can be generated automatically from the game configuration specified in the model. We constructed this modeling environment in the GEMS [8] Eclipse modeling tool.

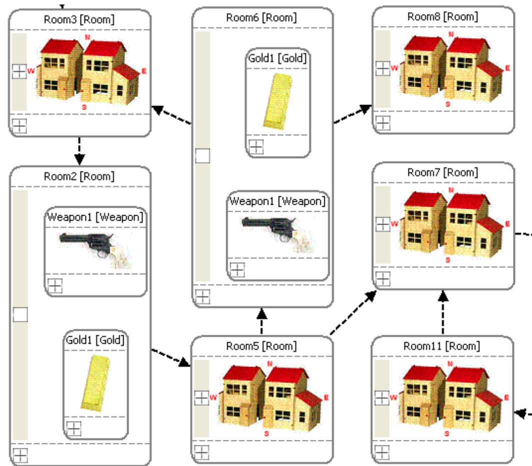


Fig. 3. An excerpt of a MazeGame model instance

Building various game configurations using the MazeGame modeling language often involves performing different model transformation tasks for maintenance purposes. For instance, if there are rooms that contain both *gold* and a *weapon* (the two unfolded rooms in Figure 3, *Room2* and *Room6*), we can implement a model transformation to remove the gold, and replace the *weapon* with a *monster*, with the *strength* of the new *monster* set to half of the *strength* of the replaced *weapon*. Game designers can apply this transformation when they discover that the number of *monsters* is far less than that of *weapons*, making the game too easy (we presented this scenario in [5], but used here for explanation of the MTDB Debugger).

4.2 Debugging in Action

In order to illustrate the usage of MTBD Debugger, we consider transformation errors that end-users may make in this case study when using MTBD, and show how MTBD Debugger can track and locate these errors.

Debugging Example 1. This first example is based on the following transformation task: if a *Monster* is contained in a *Room*, whose *strength* is greater than *100*, replace this *Monster* with a *Weapon* having the same *strength*, and add a *Gold* piece in the same *Room*. Figure 4 shows a concrete example for this transformation task.

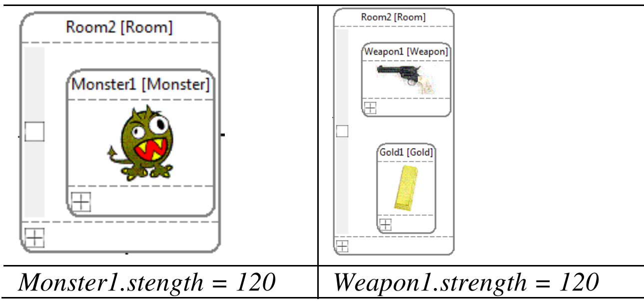


Fig. 4. The excerpt of a MazeGame model before and after replacing the monster

Based on this scenario, a user starts the demonstration by first locating a *Room* with a *Monster* in it, and deleting the *Monster*, followed by adding a *Weapon* plus a *Gold* piece. The *strength* of the new *Weapon* can be configured using the attribute refactoring editor. Finally, a precondition on *Monster* is needed to restrict the transformation (*Monster1.strength > 100*). As shown in List 1, the user performed all the correct operations except the incorrect precondition was provided (*Monster1.strength > 10*).

List 1 – Operations for demonstrating replacement of a Monster

Sequence	**Operation Performed**
1	Remove *Monster1* in *Root.TextGameFolder.Room2*
2	Add a *Weapon* in *Root.TextGameFolder.Room2*
3	Add a *Gold* in *Root.TextGameFolder.Room2*
4	Set *Root.TextGameFolder.Room2.Weapon.strength* = *Monster1.strength* = *120*
5	Set precondition on *Monster1*: *Monster1.strength > 10*

When applying this generated pattern to the model, it may be found that the transformation takes place in every *Room* with a *Monster* in it even the *strength* of the *Monster* is less than 100, which is not the desired result. Obviously, if the *strength* of every *Monster* is greater than 10, the incorrect precondition can be satisfied with all *Monsters* in the model instance. To debug the error, we execute the transformation pattern again using MTBD Debugger. As shown in Figure 5, the Pattern Execution view lists all the operations to be performed, while the Pattern Matching view provides the currently matched elements for the transformation pattern. Users can step through each of the operations, and the corresponding model elements needed for each operation will be highlighted. For instance, the very first operation in this scenario is to remove the *Monster* in the *Room*. Before executing this operation and stepping to the next one, we can determine which *Monster* is currently matched as the target to be removed. In this case, the *Monster1* in *Room12* is about to be removed. If we check the *strength* attribute of *Monster1* (e.g., *30*), we can observe that there is something wrong with the precondition we specified in the demonstration, because the *strength* of this *Monster* is not greater than *100*. At this point, we can focus on the

precondition in the Pattern Matching view, which shows the actual precondition is "*Strength* > *10*", not "*Strength* > *100*" as desired (the highlighted red box is added to the screenshot to draw attention to the location of the error for readers; this does not appear in the actual tool). The bug is therefore identified and located.

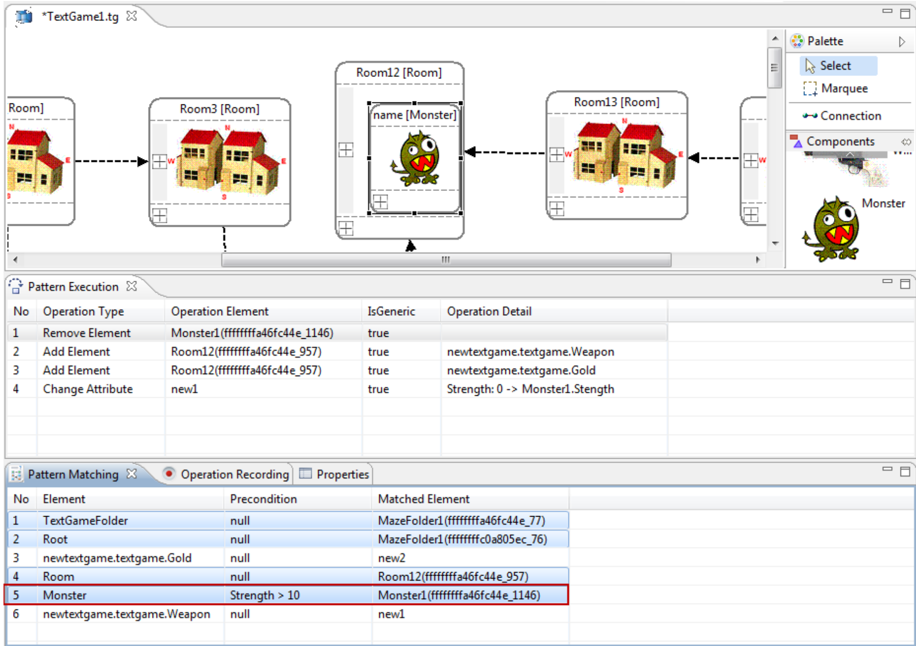


Fig. 5. Debugging the transformation pattern of Example 1

The error in the first example comes from a mistakenly specified precondition that over-matched the model elements. In the second example, we present how to debug a transformation pattern that contains preconditions that are under-matched.

Debugging Example 2. The second example is based on the same transformation scenario as the first one to replace the *Monster* with a *Weapon*. However, in this second demonstration, instead of giving the correct precondition "*Strength* > *100*", the user specified "*Strength* > *1000*" by mistake. As we can imagine, the result of executing this transformation pattern will probably not replace any of the *Monsters* in the model instance, because there are seldom *Monsters* whose *strength* is greater than *1000*.

Similar to the first example, when using the MTBD Debugger to step through the execution process, we can find out the currently matched model elements for each operation. As shown in Figure 6, the first operation to remove the *Monster* contains a null operation element as the target, which means that there is not a *Monster* in the current model instance that can be matched as an operand for this operation. We may think that there is again something wrong with the precondition, so we take a look at the precondition in the Pattern Matching view, and find the precondition set incorrectly as "*Strength* > *1000*".

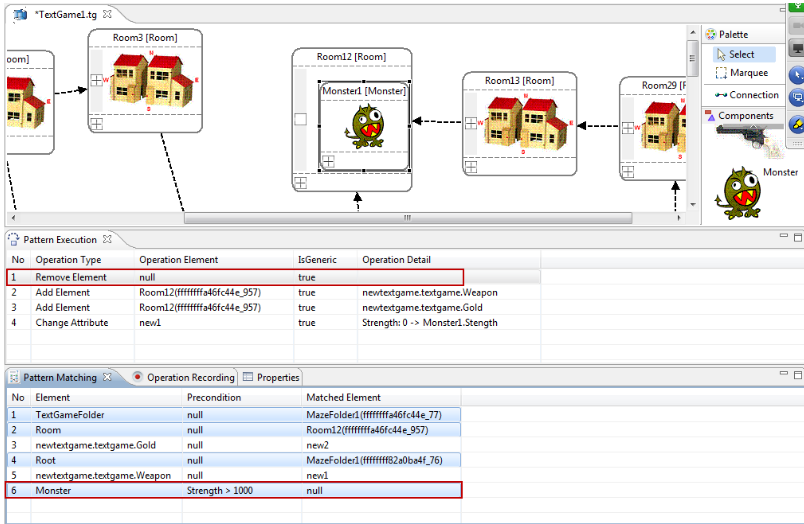


Fig. 6. Debugging the transformation pattern of Example 2

Debugging Example 3. Using MTBD, one of the scenarios that may cause an error is the refinement on the transformation actions in order to identify generic repeatable operations. The third example is based on the scenario that we want to remove all the pieces of *Gold* in all the *Rooms*, no matter how many pieces there are in the *Room* (see Figure 7).

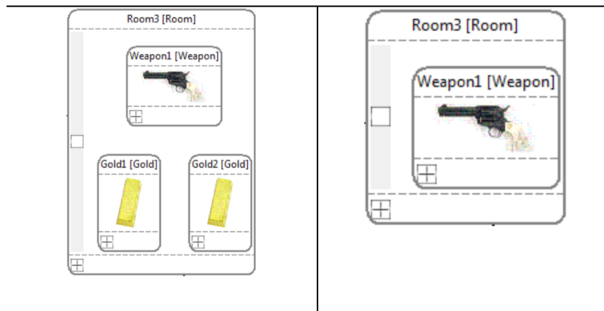


Fig. 7. The excerpt of a MazeGame model before and after removing all Gold

To specify the transformation pattern, a user performs a demonstration on a *Room* that contains two pieces of *Gold* (two operations performed - see List 2).

List 2 – Operations for demonstrating removing all pieces of Gold

Sequence	Operation Performed
1	Remove *Gold1* in *Root.TextGameFolder.Room3*
2	Remove *Gold2* in *Root.TextGameFolder.Room3*

Fig. 8. Debugging the transformation pattern of Example 3

Without giving further refinement on the transformation actions, the user may complete the demonstration. When executing the generated transformation pattern on the model, however, it is found that the Rooms that contain only one piece of *Gold* were not transformed as expected. To track the error, the pattern can be re-executed step-by-step using MTBD Debugger. As listed in the Pattern Execution view, we can see that there are two operations in this pattern, and each operation requires a different target element (i.e., the *Gold* to remove). When the *Room* contains only one piece of *Gold*, the second operation cannot be provided with a correct operand as shown in Figure 8. Thus, the problem of this bug comes from the fact that the transformation actions are not generic so that it always requires a fixed number of model elements to enable the correct transformation. The correct way to use MTBD is to make the demonstration concise, such that users should only demonstrate a single case followed by identifying the necessary generic operations. Thus, the correct demonstration should be done by removing only one piece of *Gold* and then marking it as generic.

Debugging Example 4. Following Example 3, the user may re-demonstrate the removal of *Gold* pieces by only performing a single removal operation. However, the wrong transformation pattern will be generated again due to the user forgetting to mark the operation as generic. This time, when the pattern is executed, only one piece of *Gold* can be removed in each *Room*. To track the error, the MTBD Debugger can reveal whether each operation is generic. When stepping through the execution in *Room3* (Figure 9, which contains two pieces of *Gold*), the user finds that another *Room* is matched after removing only one piece of *Gold*. The user may think that the problem is caused by the generic operations, so by double-checking the generic status, it can be seen from the Pattern Execution view that the removal operation is not generic (the highlighted box marked as false in the middle of the figure).

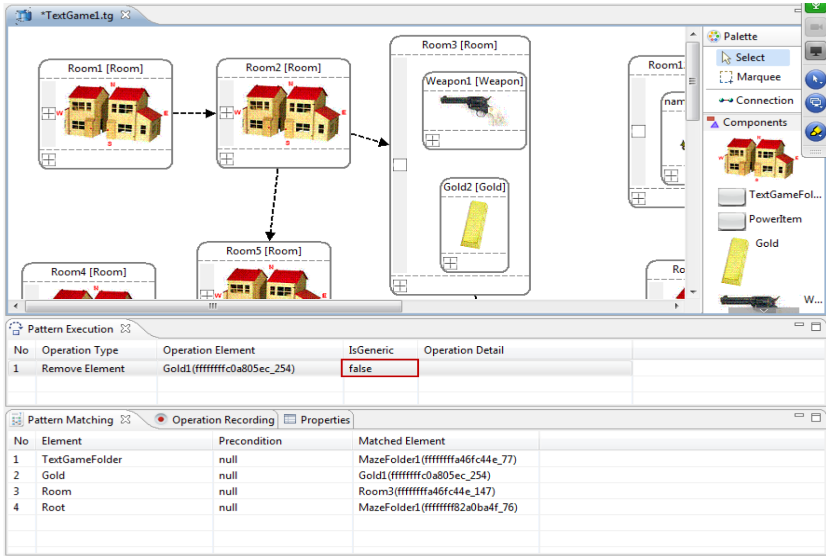


Fig. 9. Debugging the transformation pattern of Example 4

Debugging Example 5. Another common error that occurs when using MTBD is choosing the wrong element in the demonstration process, particularly in the attribute editing demonstration. For example, the user may want to replace all the *Monsters* with *Weapons*, as well as doubling the *strength* of the new *Weapons*, as shown in Figure 10.

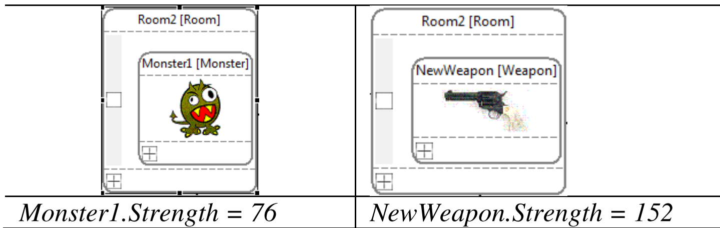


Fig. 10. The excerpt of a MazeGame model before and after doubling the new weapon

The recorded operations are in List 3. An attribute transformation is demonstrated using the attribute refactoring editor. The expected computation of the *strength* is to use the removed *Monster* and double its *strength* value. However, operation 3 in the list mistakenly selects the wrong *Monster* (i.e., *Monster1* in *Room1*) which is not the *Monster* that has just been removed (i.e., *Monster1* in *Room2*). The wrong execution result triggered by this bug is that the new *Weapon* being added in the *Room* uses the strength value of the *Monster* in a different *Room*, which is not what user expects to double.

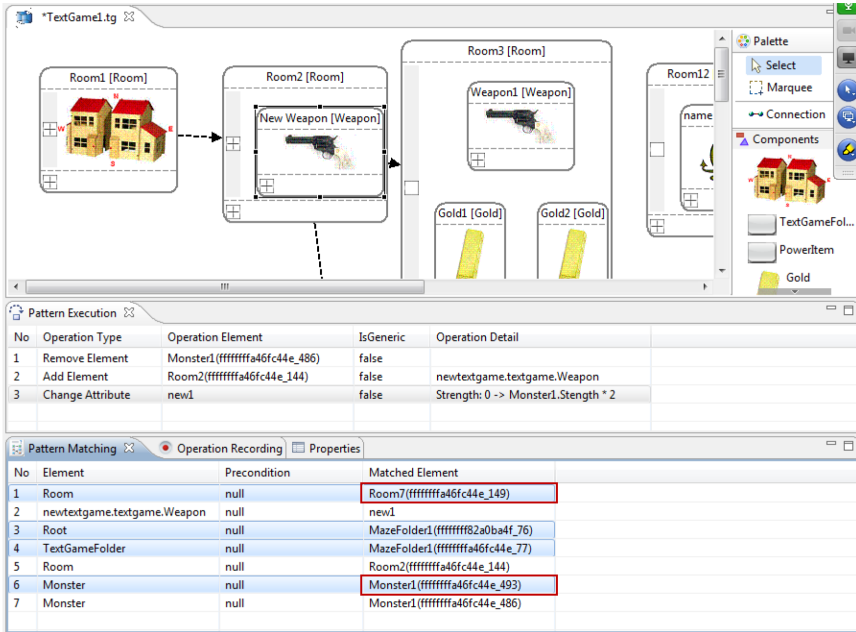


Fig. 11. Debugging the transformation pattern of Example 5

List 3 – Operations for demonstrating replacing a Monster and doubling the strength

Sequence	Operation Performed
1	Remove *Monster1* in *Root.TextGameFolder.Room2*
2	Add a *Weapon* in *Root.TextGameFolder.Room2*
3	Set *Root.TextGameFolder.Room2.Weapon.strength* = *Root.TextGameFolder.Room1.Monster1.strength* * 2 = *152*

This type of bug can be located easily using MTBD Debugger, as shown in Figure 11. When we step through each operation, the used elements in the Pattern Matching view cab be observed. In this case, the remove element operation is done on *Monster1* in *Room2*, while the change attribute operation uses *Monster1* in *Room7*, which means that we probably selected the wrong element in the demonstration of the attribute change process.

5 Related Works

As one of the most popular MTLs, ATL has an associated debugger [9] to provide the basic debugging options similar to general-purpose programming languages, such as step-by-step execution, setting up breakpoints, and watching current variables. Additionally, simple navigation in source and target models is supported. However, all these debugging options are closely related with the language constructs, so it is inappropriate for general end-users who do not have the knowledge of ATL.

Similarly, in the Fujaba modeling environment, Triple Graphical Grammar (TGG) rules [10] can be compiled into Fujaba diagrams implemented in Java, which allows debugging TGG rules directly [11].

Schoenboeck et al. applied a model transformation debugging approach [12] using Transformation Nets (TNs), which is a type of colored Petri Net. The original source and target metamodels are used as the input to derive places in TNs, while model instances are represented as tokens with the places. The actual transformation logic is reflected by the transitions. The derived transformation provides a formalism to describe the runtime semantics and enable the execution of model transformations. An interactive OCL console has been provided to enable users to debug the execution process. TNs are at a higher level of abstraction than MTLs (e.g., QVT is used as the base in this approach), so this approach helps to isolate users from knowing the low-level execution details. Although TNs can be considered as a domain-specific modeling language (DSML) to assist debugging model transformations, it is a different formalism from the specific model transformation area and can be used as a general-purpose specification in many domains, which inevitably limits its end-user friendliness. Most users may find it challenging to switch their model transformation tasks to colored Petri Net transition processes. TNs also aim at defining the underlying operational semantics that are hidden in the model transformation rules, and this exerts an extra burden in its understandability to general end-users and domain experts.

A similar work has been done by Hillberd [13] which presents forensic debugging techniques to model transformation by using the trace information between source and target model instances. The trace information can be used to answer debugging questions in the form of queries that help localize the bugs. In addition, a technique using program slicing to further narrow the area of a potential bug is also shown. Compared with MTBD Debugger, which is a live debugging tool, this work of Hillberd et al. focuses on a different context – forensic debugging. Similar to the ATL debugger, their work aims at providing debugging support to general MTLs used in MDE.

Another related work is focused on debugging a different type of model transformation – Model-to-text (M2T). Dhoolia et al. present an approach for assisting with fault localization in M2T transformations [14]. The basic idea is to create marks in the input-model elements, followed by propagating the marks to the output text during the whole transformation, so that a dynamic process to trace the flow of data from the transform input to the transform output can be realized. Using the generated mark logs and a location where a missing or incorrect string occurs in the output, the user can examine the fault space incrementally.

6 Conclusions and Future Work

Our recent work has focused on tools and concepts that allow end-users to participate in the model transformation process by allowing them to record a desired transformation directly on instance models, rather than applying transformation

languages that may be unfamiliar to them. This paper extends end-user participation in model transformation by presenting a technique that supports end-user debugging of model transformation patterns that were initially recorded through user demonstration. The MTBD Debugger allows users to step through each action in the transformation pattern and check all the relevant information through two views. The MTBD Debugger has been implemented as an extension to the MTBD execution engine and integrated with the MTBD framework.

The MTBD debugger can be applied to the core elements specified in a model transformation pattern. However, one drawback of the current views used in the debugger is that they are textual and not visual. For instance, the Pattern Matching View shows all the needed elements for each action. However, the containment relationship among these elements cannot be seen clearly. It would be very helpful to have another view that shows all the currently involved model elements and their relationships visually. Future work will provide a view that can capture the specific part of the current model that is used for the next transformation action. This can enable users to catch and check the matched elements more easily.

Another option that is useful in the general debugging process, but missing in the MTBD debugger, is the concept of setting a breakpoint. In some large model transformation scenarios (e.g., scaling up a base model to a large and complex state), it is not necessary to watch all the actions being executed one-by-one, so setting a breakpoint would make debugging more useful in this case. Thus, in the Pattern Execution View, it would be helpful to enable the breakpoint setup in the action execution list.

Acknowledgement. This work is supported by NSF CAREER award CCF-1052616.

References

1. Czarnecki, K., Helsen, S.: Feature-based Survey of Model Transformation Approaches. IBM Systems Journal 45(3), 621–645 (2006)
2. Sendall, S., Kozaczynski, W.: Model Transformation - The Heart and Soul of Model-Driven Software Development. IEEE Software, Special Issue on Model Driven Software Development 20(5), 42–45 (2003)
3. Wimmer, M., Strommer, M., Kargl, H., Kramler, G.: Towards Model Transformation Generation By-Example. In: The 40th Hawaii International Conference on Systems Science, Big Island, HI, p. 285 (January 2007)
4. Varró, D.: Model Transformation By Example. In: Wang, J., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 410–424. Springer, Heidelberg (2006)
5. Sun, Y., White, J., Gray, J.: Model Transformation By Demonstration. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 712–726. Springer, Heidelberg (2009)
6. Sun, Y.: Model Scalability Using a Model Recording and Inference Engine. In: International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion (SPLASH 2010), Reno, NV, pp. 211–212 (October 2010)

7. Balogh, Z., Varró, D.: Model Transformation by Example using Inductive Logic Programming. Software and Systems Modeling 8(3), 347–364 (2009)
8. White, J., Schmidt, D., Mulligan, S.: The Generic Eclipse Modeling System. In: Model-Driven Development Tool Implementer's Forum at the 45th International Conference on Objects, Models, Components and Patterns, Zurich Switzerland (June 2007)
9. Allilaire, F., Bézivin, J., Jouault, F., Kurtev, I.: ATL: Eclipse Support for Model Transformation. In: The Eclipse Technology eXchange Workshop (eTX) of the European Conference on Object-Oriented Programming (ECOOP), Nantes, France (July 2006)
10. Königs, A.: Model Transformation with TGGs. In: Model Transformations in Practice Workshop of MoDELS 2005, Montego Bay, Jamaica (September 2005)
11. Wagner, R.: Developing Model Transformations with Fujaba. International Fujaba Days, Bayreuth, Germany, pp. 79–82 (September 2006)
12. Schoenboeck, J., Kappel, G., Kusel, A., Retschitzegger, W., Schwinger, W., Wimmer, M.: Catch Me If You Can – Debugging Support for Model Transformations. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 5–20. Springer, Heidelberg (2010)
13. Hibberd, M., Lawley, M., Raymond, K.: Forensic Debugging of Model Transformations. In: International Conference on Model Driven Engineering Languages and Systems, Nashville, TN, pp. 589–604 (October 2007)
14. Dhoolia, P., Mani, S., Sinha, V.S., Sinha, S.: Debugging Model-Transformation Failures Using Dynamic Tainting. In: D'Hondt, T. (ed.) ECOOP 2010. LNCS, vol. 6183, pp. 26–51. Springer, Heidelberg (2010)
15. Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I.: ATL: A model transformation tool. Science of Computer Programming 72(1/2), 31–39 (2008)
16. Kappel, G., Langer, P., Retschitzegger, W., Schwinger, W., Wimmer, M.: Model Transformation By-Example: A Survey of the First Wave. In: Düsterhöft, A., Klettke, M., Schewe, K.-D. (eds.) Thalheim Festschrift. LNCS, vol. 7260, pp. 197–215. Springer, Heidelberg (2012)
17. Sun, Y., Gray, J., Langer, P., Kappel, G., Wimmer, M., White, J.: A WYSIWYG Approach to Support Layout Configuration in Model Evolution. In: Emerging Technologies for the Evolution and Maintenance of Software Models. Idea Group (2011)
18. Sun, Y., White, J., Gray, J., Gokhale, A.: Model-Driven Automated Error Recovery in Cloud Computing. In: Model-driven Analysis and Software Development: Architectures and Functions, Hershey, PA, pp. 136–155. IGI Global (2009)
19. 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)

DPMP: A Software Pattern for Real-Time Tasks Merge

Rania Mzid[1,2,3], Chokri Mraidha[1], Asma Mehiaoui[1,2],
Sara Tucci-Piergiovanni[1], Jean-Philippe Babau[2], and Mohamed Abid[3]

[1] CEA, LIST, Laboratory of Model Driven Engineering for Embedded Systems
Point Courrier 174, Gif-sur-Yvette, 91191, France
{rania.mzid,chokri.mraidha,asma.mehiaoui,sara.tucci@}@cea.fr

[2] Lab-STICC, UBO, UEB, Brest, France
Jean-Philippe.Babau@univ-brest.fr

[3] CES Laboratory, National School of Engineers of Sfax, Sfax, Tunisia
Mohamed.Abid@enis.rnu.tn

Abstract. In a model-driven development context, the refinement of the architectural model of a real-time application to a Real Time Operating System (RTOS) specific model is a challenging task. Indeed, the different design choices made to guarantee the application timing properties are not always implementable on the target RTOS. In particular, when the number of distinct priority levels used at the design level exceeds the number allowed by the RTOS for the considered application, this refinement becomes not possible. In this paper, we propose a software pattern called Distinct Priority Merge Pattern (DPMP) that automatically perform the re-factoring of the architectural model when this problem occurs. First, we give an heuristic algorithm describing this pattern and we show that this method is not always effective. Then, to address the limitations of the first method, we propose a MILP formulation of the DPMP pattern that allows to check whether a solution exists and gives the optimal one. The evaluation of the second method, allows to estimate a cost in terms of processor utilization increase during the deployment of an application on a given RTOS family characterized by the number of distinct priority levels that it offers.

Keywords: Real-Time Validation, Architectural Model, RTOS-Specific Model, Software Pattern, Re-factoring, MILP Formulation.

1 Introduction

In order to increase productivity and reduce the time-to-market during the development of Real-Time Embedded Systems (RTES), Model-Driven Development(MDD) proposes solutions by introducing intermediate models, from requirements specification to the binary code, that allows verification activities at each level of abstraction. In a software development context of such systems, the designer makes different architectural choices, at the design level, to describe the realization of the application. Then, a verification of timing properties is

P. Van Gorp, T. Ritter, and L.M. Rose (Eds.): ECMFA 2013, LNCS 7949, pp. 101–117, 2013.
© Springer-Verlag Berlin Heidelberg 2013

performed to assess these choices. This verification step requires an abstraction of some information related to the underlying Real-Time Operating System (RTOS) such as scheduling policy, communication mechanisms, etc. In fact, the design model is a Platform-Independent Model (PIM), thus most of the verification tools [1][2] used to validate this model make assumptions about the target RTOS and consider that is an ideal one offering thus unlimited (software) resources without any limitation. In that case, the refinement of the design model to an RTOS-specific model, which corresponds to a deployment phase, is a non trivial transformation because the assumptions made may be not verified for the selected RTOS.

In previous works [3][4], we have proposed a model-driven approach to guide the transition from real-time design model to an RTOS-specific model and to verify the correctness of the resulting model in terms of timing properties. This approach integrates two steps; a deployment feasibility tests step and a mapping step. The approach is based on explicit description of the abstract platform used to verify the design model and the concrete one corresponding to the RTOS. The different platform models are created using UML enriched with the Software Resource Modelling (SRM) sub-profile of MARTE [5]. Indeed, the deployment feasibility tests step defines a set of tests to verify whether the real-time design model is *implementable* on the target RTOS. When a problem is detected an error is generated to inform the designer about the source and the rationale of the problem.

In the present paper, we extend the proposed approach by introducing an *automatic* pattern-based re-factoring of the design model when a deployment problem is detected. Indeed, in this paper, we are interested in a particular one that occurs when the number of distinct priority levels used to validate the real-time application is greater than the number authorized by the RTOS. Indeed, at the design level, this number is supposed to be unbounded which is not the case for the majority of RTOSs that offer a limited number of distinct priority levels or when for extensibility concerns this number is bounded for a particular application in order to conserve spare priorities for additional future functionalities.

To address this issue, we propose a software pattern that we call Distinct Priority Merge Pattern (DPMP). For a particular application, this pattern looks at reducing the number of used priority levels by merging harmonic tasks having distinct priorities while ensuring the respect of timing properties. In this paper, we show that using a heuristic method to formulate this problem is not always effective and we propose a Mixed Integer Linear Programming (MILP) formulation of this pattern. Given a design model as input and a RTOS as target, our linear program checks whether a solution exists and finds the optimized one in terms of processor utilization. An evaluation of this pattern allows to estimate the performance loss when deploying a real-time application on a given RTOS family characterized by the number of distinct priority levels that it offers.

This paper is organized as follows. The first section briefly describes a design refinement (toward implementation) method and specifies the assumptions that

must be fulfilled by the considered design models. In section 2 we describe the context, the problem and the solution of the proposed pattern. In section 3, two formulations of the DPMP pattern are given; algorithmic description and MILP formulation. Some experimental results are given in section 4 to evaluate our proposal. Section 5 presents some related work and section 6 concludes the paper.

2 A Method for Design Refinement

The objective of the proposed method is to reduce the gap between the design and the implementation models during real-time application development process. In this section we briefly describe the proposed method. Then, we give a formal description of the design model.

2.1 Method Overview

Fig.1 gives an overview of the proposed refinement method. The entry point is a design model that is generated following the methodology given in [1]. In fact, this methodology introduces timing verification from the functional level in order to ensure that the constructed design model satisfies the application timing requirements.

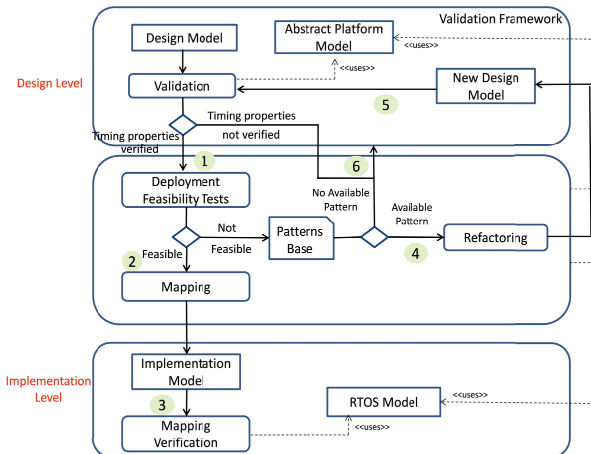


Fig. 1. Design Refinement Method Overview

Our objective is to ensure a correct transition from a correct design model, to the implementation model while preserving its timing properties. Indeed, we are interested in the semantics of the software platform resources involved during the refinement. To this end, in previous work [3], we have proposed to integrate two steps; deployment feasibility tests (1) step and mapping step (2). The first step defines a set of feasibility tests to verify whether the design choices are

implementable on the target RTOS. When no feasibility concerns are raised, the mapping step generates the appropriate RTOS-specific model. These two steps are based on an explicit description of an abstract platform used for validation [3] and a concrete platform which corresponds to the RTOS. The mapping verification step (3) previously introduced in [4], defines the set of properties that must be verified to confirm the correctness of the refinement.

In this paper, we are addressing the case where the design model is not implementable on the target RTOS. In that situation, the deployment feasibility tests step generates a warning to highlight that the input model is not implementable for a particular reason. One objective of our work is to guide the designer by proposing solutions whenever the refinement is not feasible. To this end, we create a *pattern base* which collects a set of predefined patterns. Each pattern aims at solving a particular deployment problem in the case where some particular assumptions are fulfilled by the considered design model. Therefore, when a problem is detected, we verify if a pattern corresponding to this problem exists in the pattern base. If it is not the case, our framework generates an error to inform the designer that the design model is not implementable on the selected RTOS and that no solution is available to solve the problem. Otherwise, when a pattern is available (4), we perform the re-factoring of the design model by applying this pattern. This re-factoring must guarantee two points: (1) the portability and (2) the preservation of timing properties. Regarding the first point, even if the re-factoring of the design model is performed to handle the deployment problems related to the target RTOS, it must still independent from the latter: the resulting design model (denoted new design model in Fig.1) is also a PIM. In order to ensure the second point, the new design model must be revalidated (5). After performing the revalidation, if the timing properties are not verified, an error is generated to mention that the model is not implementable and no solution is available (6).

2.2 Design Model Formalization

We assume that the considered real-time design model consists of m *periodic* tasks that we denote by $M = \{T_1, T_2, \ldots, T_m\}$ running in a single-processor system. Each task T_i is defined by a set of parameters deducted from the high-level model (functional model) and the architectural choices enabling thus the timing validation. Indeed, a task T_i is characterized by its priority p_i, its execution time c_i which is considered as input in our case, its activation period P_i supposed to be an integer in this paper and its deadline D_i that represents the time limit in which a task must complete its execution. We assume that 0 is the highest priority level and that tasks may share the same priority level. Let us denotes as n the number of distinct priority levels used in the architectural model for validation ($n \leq m$) and with N the number of distinct priority levels allowed by the platform for the considered application.

The architectural model consists also of a set of software resources $R = \{R_1, R_2, \ldots, R_\ell\}$ that can be shared between one or several tasks (e.g. a mutex to access a critical section). We denote c_{R_i,T_j} the worst-case time for the task

T_j to acquire and release the lock of the resource R_i in case of no contention. Let us remark that c_{R_i,T_j} is considered as an input and that $c_{R_i,T_j} \leq c_j$. Due to the presence of shared resources, a task is also characterized by a blocking time B_i. The blocking time accounts for the time a higher-priority task has to wait, before acquiring the lock, since a lower-priority task owns this lock. The computation of this term depends on the synchronization protocol used to implement the access to the shared resource. In this paper, we suppose that Priority Ceiling protocol(PCP) [6] is used as a synchronization protocol to avoid unbounded priority inversion and mutual deadlock due to wrong nesting of critical sections. In this protocol each resource R_i is assigned a priority ceiling π_i, which is a priority equal to the highest priority of any task which may lock the resource. The expression used to compute the blocking time for the PCP protocol is given just below:

$$B_i = \max_{T_j \in HP, R_k \in R} \{c_{R_k,T_j} : p_j < p_i \ and \ \pi_k \geq p_i\} \tag{1}$$

We perform Rate-Monotonic (RM) response time analysis [7]. The analysis results correspond to the computation of the processor utilization U and the response time Rep_i of the different tasks in the model. The model satisfies its timing constraints if and only if $U \leq 1$ and $\forall i \in \{1..m\}$ $Rep_i \leq D_i$. The expressions used to compute U and Rep_i are given just below, where HP_j represents the set of tasks with priority higher than T_j.

$$U = \sum_{T_i \in M} \frac{c_i}{P_i} \tag{2}$$

$$Rep_i = c_i + B_i + \sum_{T_j \in HP_j} \left\lceil \frac{Rep_i}{P_j} \right\rceil * c_j \tag{3}$$

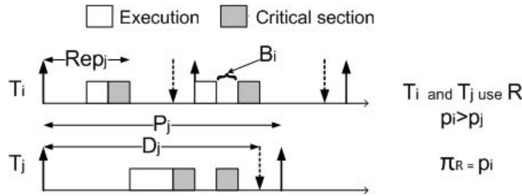


Fig. 2. Real-Time Concepts

Fig.2 shows an example of execution of two periodic tasks (T_i and T_j) sharing the resource R. The priority ceiling of R is equal to the priority of T_i as it is the highest.Up-raising arrows represent the instants of tasks activation, for their part, down-raising arrows determine the deadline for each task activation. The response time to an activation is defined as the time between the activation and the completion instants. We also show the blocking time B_i of the task T_i resulting from the utilization of R by T_j.

In addition, each task implements a set of functions that we denote by $f \subset F/card(f) \geq 1$ such as F is the set of functions defined by the application (from the functional model).

3 Problem Statement and Solution

In this section, we identify the case where the Distinct Priority Merge Pattern (DPMP) must be applied. Then the proposed solution is detailed.

3.1 Problem Statement

This pattern is automatically applied on the design model when the *Deployment Feasibility Tests* step detects that the number of distinct priority levels used in the architectural model exceeds the number allowed by the RTOS for the considered application (i.e. $n > N$). The resulting design model after applying this pattern must still verifying the timing requirements as specified in section 2.2.

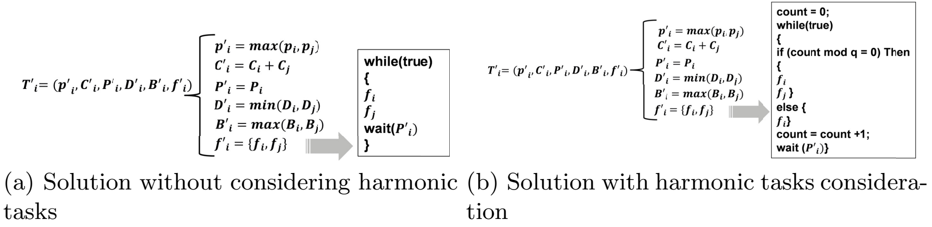
3.2 Solution Description

In order to solve the problem (i.e. $n > N$), we propose to reduce n to be equal to N by merging tasks having distinct priority levels. This operation is repeated until the number of distinct priority levels becomes equal to N. However, the proposed solution must preserve:

1. The high level specification i.e. the activation rate of the different functions defined in the specification must be preserved.
2. The real-time constraints i.e. the response time of the all considered tasks is lower than their deadline.

Let us consider an initial model $M = \{T_1, T_2, ., T_m\}$ defining m tasks and n distinct priority levels ($n \leq m$). Let us consider also two tasks T_i and $T_j \in M$, each task is defined by a set of parameters; $T_i = (p_i, C_i, P_i, D_i, B_i, f_i)$ and $T_j = (p_j, C_j, P_j, D_j, B_j, f_j)$ such as $p_i \neq p_j$, $P_j \geq P_i$ and f_i, f_j corresponds to the functions implemented respectively by T_i and T_j. We denote by T_i' the task resulting from merging these two tasks such as $T_i' = (p_i', C_i', P_i', D_i', B_i', f_i')$. Consequently, the resulting model M' consists of *m-1* tasks and *n-1* distinct priority levels.The obtained task T_i' is described in Fig.3(a).

The problem with the resulting model described in Fig.3(a) where one of the two merged tasks will be executed with a rate different from the one defined in the high level specification and thus the first constraint (1) previously defined will be violated. In order to avoid this problem, we consider also in the solution that only *harmonic tasks* may be merged (i.e. two tasks T_i and T_j are harmonic if and only if ($P_j \mod P_i = 0$). By considering this additional assumption ($\frac{P_j}{P_i} = q$ with q in an integer), the period of the resulting task which corresponds to minimum of the two periods will be equal to P_i and the implementation of



(a) Solution without considering harmonic tasks (b) Solution with harmonic tasks consideration

Fig. 3. Solution Description

f'_i will be modified in such a way that the execution rate of the two functions is preserved. The new solution is presented in Fig. 3(b).

In order to guarantee the second constraint (2), we have to re-validate the model after merging the tasks in order to verify whether the new design model still satisfy the timing constraints.

4 DPMP Formulation

This section presents an algorithmic description of the previously proposed solution. Then, we show the limitations of this method and we propose a MILP formulation of the DPMP pattern.

4.1 Heuristic Method

Algorithm 1 just below corresponds to an algorithmic description of the DPMP pattern. This algorithm merges recursively tasks in pairs. After each merge, this algorithm performs a re-validation to verify the timing properties. The algorithm ends, when the number of distinct priority levels used in the resulting model is equal to the number authorized by the target RTOS or when there is no harmonic tasks in the model. The complexity of this algorithm is linear and it depends on the number of tasks in the initial model.

Heuristic Method Limitations. The problem of merging tasks with the objective to reduce the number of distinct applicative priority levels is *a combinatorial problem.* In fact, the solution depends on the application (i.e. n and the period of the different tasks) and the target RTOS. Consequently, the heuristic method presented in previous section is not always able to find the solution.

Let's consider an initial model $M = \{T_1, T_2, T_3, T_4\}$. Each task is characterized by a set of parameters such as $T_1 = (1, 4, 10, 10, 0, f_1)$, $T_2 = (2, 5, 20, 20, 0, f_2)$, $T_3 = (3, 2, 30, 30, 0, f_3)$ and $T_4 = (4, 1, 60, 60, 0, f_4)$.The initial processor utilization is evaluated to 73,33 %. All the tasks in this model are independent and have distinct priority levels (n=4). We can notice that task T_1 is harmonic with all the other tasks and also T_2, T_3 are harmonic with T_4. In addition, we suppose that the target RTOS authorizes only *two* priority levels for this application (N=2).

Algorithm 1. DistinctPriorityMergePattern

```
Input:
M_a: Design Model describing the application
N : The number of distinct priority levels allowed by the RTOS for the considered
application
Ref-Period : A reference period used to detect harmonic tasks
Output:
M_Res: New design model after reducing the number of priority levels
Notations:
Ref-Task: The reference task with the lowest value of period
H-Tasks: The set of tasks which are harmonic with Ref-Task
n: The number of distinct priority levels used in the design model
begin
    M_Res ⟵ M_a
    n ⟵ getPriorityLevelsNumber(M_a)
    Ref-Task ⟵ getMinPeriod(M_a, Ref-Period)
    if (Ref-Task ≠ null) then
        for (i_s ∈ M_a / i_s is a periodic task) do
            if (IsInteger(period(i_s), period(Ref-Task)) then
                if (priority(i_s) ≠ priority(Ref-Task)) then
                    add(i_s, H-Tasks)

        if (SizeOf(H-Tasks) ≤ 1) then
            Ref-Period ⟵ period(Ref-Task)
            DistinctPriorityMergePattern(M_a,N,Ref-Period)

        else
            if (n ≻ N) then
                M'_a ⟵ Merge(H-Tasks[1],H-Tasks[2])
                OK ⟵ Re-Validate (M'_a)
                if (Ok = true) then
                    M_Res ⟵ M'_a
                    DistinctPriorityMergePattern(M_Res,N,Ref-Period)

    return M_Res
```

Table 1. Example: Possible Solutions and Utilization Estimation

Possible Solutions	Utilization
$M_1 = (\{T_1, T_2\}, \{T_3, T_4\})$	100%
$M_2 = (\{T_1, T_3\}, \{T_2, T_4\})$	90%
$M_3 = (\{T_1, T_2, T_3\}, T_4)$	111,67%
$M_4 = (T_2, \{T_1, T_3, T_4\})$	95%
$M_5 = (T_3, \{T_1, T_2, T_4\})$	106,66%

As a result, for this particular example, 5 solutions are possible. These solutions are presented in Table 1. For the first solution for example, we choose to merge T_1 and T_2 from one side and T_3 and T_4 from the other side in order to obtain the model M_1 consisting of just two tasks and thus two distinct priority levels. For this particular example, the heuristic method didn't find a feasible solution as it found M_3 which is not feasible due to analysis issues (utilization 111,67%$\succ$ 100%). Therefore, this method is not always effective. From these considerations, an appropriate method must be considered to solve this problem. This method must be able to confirm whether a solution for each particular problem (application and RTOS) exists. In addition, when many solutions are available, this method should find one which costs less performance degradation. In next section, we propose a MILP formulation of this problem.

4.2 MILP Formulation

In order to ensure a reliable implementation of the problem taking into consideration the different constraints already mentioned (timing requirements, application), we propose in this section a MILP formulation of our problem. MILP techniques define an objective function which corresponds to a formulation of the considered problem. This formulation is interpretable by a solver that seeks to find a solution for this problem under a set of defined constraints.

Objective Function. Expression (4) defines the objective function for our problem. We denote by m the number of tasks in the initial model. $Merge$ is a boolean variable used to mention whether two tasks are merged. More in detail, if $Merge_{i,j}$ is equal to 1, the merge corresponds to the situtation in which T_i absorbs T_j, then T_i augments its worst-case execution time by adding the worst-case execution time of T_j, while T_j is deleted from the model. Let us note that more than one task can be absorbed by another task. The objective function aims at maximizing the number of merge while minimizing the processor utilization.

$$\text{maximize}: \sum_{\mathrm{i,j} \in \{1..\mathrm{m}\}} \mathrm{Merge_{i,j}} - \mathrm{Utilization} \tag{4}$$

Merging Situations Constraints. The objective function aims at maximizing the number of merge, however this function should be aware of some constraints that limit the exploration space and eliminate non meaningful merging situations. These constraints are presented just below:

$$\mathrm{n} - \sum_{\mathrm{i,j} \in \{1..\mathrm{m}\}} \mathrm{Merge_{i,j}} = \mathrm{N} \tag{5}$$

$$\forall \mathrm{i,j} \in \{1..\mathrm{m}\}, \mathrm{Merge_{i,j}} = 0 \quad \text{if} \quad (\mathrm{isHarmonic_{i,j}} = 0) \quad \text{or} \quad (\mathrm{p_i} = \mathrm{p_j}) \tag{6}$$

$$\forall \mathrm{j} \in \{1..\mathrm{m}\}, \sum_{\mathrm{i} \in \{1..\mathrm{m}\} \wedge \mathrm{i} \neq \mathrm{j}} \mathrm{Merge_{i,j}} \leq 1 \;;\; \forall \mathrm{i,j,k} \in \{1..\mathrm{m}\} \wedge \mathrm{j,k} \neq \mathrm{i}, \mathrm{Merge_{i,j}} + \mathrm{Merge_{k,i}} \leq 1 \tag{7}$$

In constraint (5), n and N represent two input parameters defined previously in section 2.2. This constraint means that we have to maximize the number of merged tasks and thus minimize the number of distinct priority levels used in the design model until the number authorized by the RTOS. Indeed, this Equation serves as a bound for the objective function (i.e. the number of merge). Constraint (6) defines a new input parameter which is *isHarmonic*, this parameter is used to mention if two tasks are harmonic. Thus if the value of $isHarmonic_{i,j}$ is equal to 1, then the corresponding tasks T_i and T_j have harmonic rates. Consequently, this constraint avoids the merge of non-harmonic tasks and avoids also the merge of tasks having equal priority levels($p_i = p_j$). Finally, the constraints in (7) are used to avoid a non-meaningful situations which corresponds to the merge of a task already merged. In particular, the first contraint assures that a

task T_j can be absorbed by at most one other task, and the second constraint states that either a task absorbs another task or it is absorbed by another task. We define also a new boolean variable that we denote by TASKS and which refers to the resulting task model after merging the different tasks. Therefore, constraint (8) is defined to create the new obtained model. In fact, when $Merge_{i,j}$ is equal to 1, $TASKS_j$ will be equal to 0 and $TASKS_i$ will be equal to 1 (thanks to constraints 7). This constraint is defined as follows:

$$\forall \mathrm{j} \in \{1..\mathrm{m}\}, \mathrm{TASKS_j} = 1 - \sum_{\mathrm{i}\in\{1..\mathrm{m}\}} \mathrm{Merge_{i,j}} \tag{8}$$

Real-Time Constraints. The constraints defined in this section are related to real-time requirements. Indeed, the model obtained after applying the merge pattern should satisfy the timing constraints which are expressed in constraints (9) and (10).

$$\forall \mathrm{i} \in \{1..\mathrm{m}\}, \mathrm{Rep_i} \leq \mathrm{D_i} \tag{9}$$

$$\mathrm{utilization} \leq \mathrm{Max_Utilization} \tag{10}$$

Constraint (9) ensures that the response times Rep_i of the different tasks in the resulting model are lower or equal than their deadlines. Constraint (10) verifies whether the processor utilization is lower or equal than the maximum authorized utilization. Constraint (11) gives the computation formula of T_i response time while taking into consideration the different decisions of merge.

$$\forall \mathrm{i} \in \{1..\mathrm{m}\}, \mathrm{Rep_i} = \delta_\mathrm{i} + \theta_\mathrm{i} + \beta_\mathrm{i} \tag{11}$$

The first term of the expression (11) is δ_i which corresponds to the worst case execution time of the task T_i. This term is computed as follows:

$$\forall i \in \{1..m\}, \delta_i = TASKS_i * C_i + \sum_{j\in\{1..m\}} Merge_{i,j} * C_j \tag{12}$$

The execution time of a deleted task will be equal to 0 since the term $TASKS_i$ is equal to 0 and $\forall j \in \{1..m\}, Merge_{i,j} = 0$. However, the execution time of a task resulting from the merge of different tasks will be equal to the sum of the execution times of these tasks.

The second term in the expression is θ_i representing the overhead induced by the interferences of the task T_i with the different tasks in the model having higher priorities. This variable is defined $\forall i \in \{1..m\}$ as the sum of two terms ζ_i, γ_i and it is defined just below:

$$\theta_\mathrm{i} = \zeta_\mathrm{i} + \gamma_\mathrm{i} \tag{13}$$

$$\zeta_\mathrm{i} = \mathrm{TASKS_i} * \sum_{\substack{\mathrm{j}\in\mathrm{HP_i} \\ \mathrm{j}\in\{1..\mathrm{m}\}}} \mathrm{TASKS_j} * (\lceil \frac{\mathrm{Rep_i}}{\mathrm{P_j}} \rceil * \mathrm{C_j}) \tag{14}$$

$$\gamma_\mathrm{i} = \mathrm{TASKS_i} * [\sum_{\substack{\mathrm{j}\in\mathrm{HP_i} \\ \mathrm{j}\in\{1..\mathrm{m}\}}} \mathrm{TASKS_j} * (\sum_{\mathrm{k}\in\{1..\mathrm{m}\}} \mathrm{Merge_{j,k}} * \lceil \frac{\mathrm{Rep_i}}{\mathrm{P_j}} \rceil * \mathrm{C_k})] \tag{15}$$

The interference term is equal to 0 if the corresponding task is a deleted one ($TASKS_i$). Otherwise, this term computes the overhead resulting from the interferences of tasks $T_j/j \in HP_i$. This expression takes into consideration the different situations when higher priority tasks correspond to deleted ones ($TASKS_j$ in the expression) or tasks resulting from merging decision ($Merge_{j,k}$ in the expression). We notice that the expressions (14) and (15) are not linear and thus in order to be interpretable by the solver these expression must be linearized. For instance, the linearization of the expression (14) is given by the following constraints:

$$\forall \mathrm{i,j} \in \{1..\mathrm{m}\}, 0 \leq \mathrm{X_{i,j}} - (\frac{\mathrm{Rep_i}}{\mathrm{P_j}}) < 1 \tag{16}$$

$$\forall \mathrm{i,j} \in \{1..\mathrm{m}\}, \mathrm{Y_{i,j}} \leq \mathrm{X_{i,j}} \ ; \ \mathrm{Y_{i,j}} \leq \mathrm{M} * \mathrm{TASKS_j} \ ; \ \mathrm{X_{i,j}} - \mathrm{M} * (1 - \mathrm{TASKS_j}) \leq \mathrm{Y_{i,j}} \tag{17}$$

In order to linearize the expression (14), we define new constraints (16) (17) and 2 additional variables X and Y. The constraint (16) permits to compute the term $\lceil \frac{Rep_i}{P_j} \rceil$, however the constraints in (17) are defined to determine the value of $(TASKS_j) * \lceil \frac{Rep_i}{P_j} \rceil$. Eventually, the constraints in (18) and (19) are used to compute the final value of ζ_i, $\forall i \in \{1..m\}$.

$$\forall \mathrm{i} \in \{1..\mathrm{m}\}, \zeta_\mathrm{i} \leq \sum_{\substack{\mathrm{j} \in \mathrm{HP_i} \\ \mathrm{j} \in \{1..\mathrm{m}\}}} \mathrm{Y_{i,j}} * \mathrm{C_j} \ ; \ \zeta_\mathrm{i} \leq \mathrm{M} * \mathrm{TASKS_i} \tag{18}$$

$$\forall \mathrm{i} \in \{1..\mathrm{m}\}, [\sum_{\substack{\mathrm{j} \in \mathrm{HP_i} \\ \mathrm{j} \in \{1..\mathrm{m}\}}} \mathrm{Y_{i,j}} * \mathrm{C_j}] - \mathrm{M} * (1 - \mathrm{TASKS_i}) \leq \zeta_\mathrm{i} \tag{19}$$

Finally the third term in the expression of the response time β_i represents the blocking time. This variable is computed as follows:

$$\forall \mathrm{i} \in \{1..\mathrm{m}\}, \beta_\mathrm{i} = \mathrm{TASKS_i} * \mathrm{BT_i} \tag{20}$$

This term is equal to 0 if the task corresponds to a deleted task. Otherwise, the blocking time of the considered task is equal to BT which is defined as follows:

$$\forall \mathrm{i} \in \{1..\mathrm{m}\}, \quad \mathrm{BT_i} = \begin{cases} \mathrm{B_i} \quad \text{if} \quad \sum_{\mathrm{i,j} \in \{1..\mathrm{m}\}} \mathrm{Merge_{i,j}} = 0 \\ \max_{\mathrm{j} \in \{1..\mathrm{m}\}} \mathrm{Merge_{i,j}} * \mathrm{B_i} \quad \text{Otherwise} \end{cases} \tag{21}$$

The term B_i in expression (21) is an input parameter representing the blocking time of the task T_i. Consequently, if the considered task is not merged with other tasks in model ($\sum_{j \in \{1..m\}} Merge_{i,j} = 0$), the blocking time is kept the same. Otherwise, the blocking term corresponds to the maximum of the merged task blocking times. The processor utilization represents an important term in scheduling analysis. In fact, in order to confirm that the design model meets the timing constraints the following constraint must be verified:

$$\mathrm{Utilization} \leq 1 \tag{22}$$

We define the Utilization term by the constraints just below:

$$\mathrm{Utilization} = \mu_1 + \mu_2 \tag{23}$$

$$\mu_1 = \sum_{i \in \{1..m\}} \mathrm{TASKS_i} * (\frac{\mathrm{C_i}}{\mathrm{P_i}}) \ ; \ \mu_2 = \sum_{i \in \{1..m\}} \mathrm{TASKS_i} * \sum_{j \in \{1..m\}} \mathrm{Merge_{i,j}} * (\frac{\mathrm{C_j}}{\mathrm{P_i}}) \quad (24)$$

Under these constraints, the objective function will seek for the best way to merge tasks (i.e. the optimized solution in terms of utilization) in order to reduce the number of used priority levels while ensuring the respect of timing properties.

Let's consider the same example previously introduced in section 4.1. Considering this problem, our linear program confirms that a solution exists and generates the following $Merge$ matrix:

$$Merge = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

This matrix shows that the solution considered by the solver is the merge of T_1 and T_3 and the merge of T_2 and T_4. The processor utilization of the resulting model is 90%. Now if we compare this solution with the different possible solutions given in Table 1, we can conclude that the latter corresponds to M_2 which is the best one in terms of processor utilization.

The solution generated by the linear program will be interpreted by our framework in order to provide the information to the designer on how the design model must be re-factored.

5 Experimental Results

In this section, we present a set of experiments to test the effectiveness of the proposed pattern in terms of applicability and scalability. The experiments are carried-out on Intel Core i5-3360M processor running at 2.8 GHz with 4GB of cache memory. CPLEX is used as a MILP solver for the whole set of experiments.

We define also a new parameter that we denote *Cost*. This parameter references the *performance loss* and is defined as the difference between the utilization evaluated on the initial model and the utilization evaluated on the model resulting from the application of the merge pattern in order to avoid non-implementable design models. Expression 25 given just below defines this parameter:

$$\mathrm{Cost} = \mathrm{Current_{utilization}} - \mathrm{Initial_{utilization}} \quad (25)$$

Extensibility at the Implementation Level

We define the *extensibility* as the capacity to integrate additional applications (or functions) on the same platform. In this section, we suppose that the first focus of the designer is to maximize extensibility at the implementation level even at the expense of some performance loss. For that achievement, the designer should determine the processor utilization authorised for his application that we denote by *Max-Utilization* and the number of distinct priority levels

denoted by N. This task is not trivial because it strongly depends on the application. The previously described linear program provides a sort of guidance for the designer to help him determining one of these parameters by fixing the second. Therefore, two scenarios are considered: (1) the designer defines the maximum number of priority levels (N) that he wants to reserve for his application and asks the linear program to determine the minimum processor utilization necessary for this achievement.(2) the designer defines the maximum processor utilization (Max-Utilization) authorized for his application and asks the linear program to determine the minimum number of priority levels that are necessary to achieve such utilization. For scheduling issues, we suppose that the considered application is the first to be implemented on the platform and that the reserved priority levels are the higher ones.

In order to illustrate this idea, we consider an example of an architectural model describing an application; this model is given in table 2. The design model consists of 6 tasks; each task is characterized by a set of parameters. Besides, the model defines also two shared resources R_1 and R_2; the resource R_1 is shared between the two tasks T_1 and T_3, however R_2 is shared between T_2 and T_5. After, the different design choices, the designer performs validation to verify the timing constraints. Validation results are also presented in table 2; all the tasks meet their deadlines since their response times are lower than their deadlines. The processor utilization for this model is evaluated to 39, 69%.

Table 2. Example of an Architectural Model

Task	Period	Deadline	Wcet	Priority	Blocking Time	Response Time
T_1	10	10	2	0	2	4
T_2	20	20	2	1	2	6
T_3	40	40	2	2	1	7
T_4	80	80	3	3	1	10
T_5	160	160	1	4	0	10
T_6	320	320	1	5	0	13

Now let's consider the first scenario. For extensibility issues the designer wants to reserve just 3 priority levels for this application at the implementation level. To this end, he fixes the number N to 3 and asks the pattern to determine minimum processor utilization that should be reserved in that case. Then, the pattern generates the corresponding value which is equal to 46, 25%. For the second scenario, if the designer fixes the Max-Utilization to 45%, the pattern determines that the minimum number of priority levels that should be reserved for such utilization is 4 (N=4). Fig.4 illustrates the variation of the *Cost* with regard to the *extensibility* for the considered application.

We have already mentioned that the extensibility is inversely proportional to the number of priority levels reserved at the implementation level. Hence, in the graph we evaluate the extensibility to be equal to $\frac{1}{N}$. We can conclude that the performance loss increases when the extensibility increases.

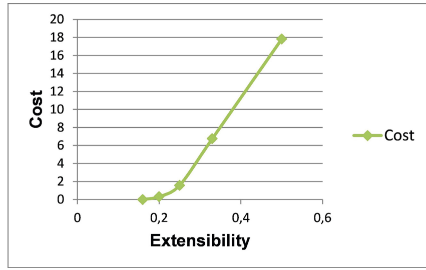
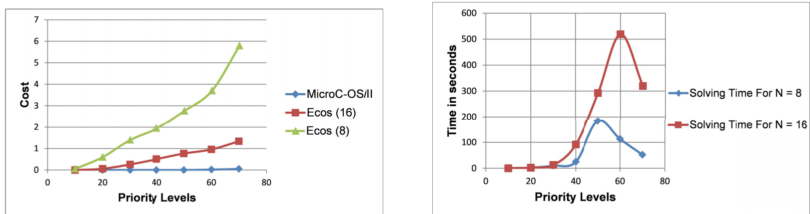


Fig. 4. Cost Variation Versus Extensibility

Insufficient Priority Levels for Large Scale Applications

In this section, we are interested on the evaluation of the merge pattern on large scale systems. In that case, during the deployment phase, the problem of insufficient number of priority levels authorized by the target RTOS will more likely occur. To this end, we consider different design models. Each model M consists of $\{T_1, T_2, , T_n\}$; n defines the number of distinct priority levels used in the model. We suppose also that each task $T_i \in M$ is defined by a set of parameters $(p_i, C_i, P_i, D_i, B_i)$. In addition, we assume that for each model $\forall i, j \in \{1..n\}$, T_i and T_j are harmonic (i.e. P_j mod P_i =0) and $\forall i, j \in \{1..n\}$ $p_i \neq p_j$. This brings us to identify different categories of RTOS depending on their number of distinct priority levels. In this paper, we consider two examples of RTOSs; MicroC/OS-II [8] and Ecos [9]. Indeed, MicroC/OS-II offers 56 applicative distinct priority levels, however, Ecos provides the possibility to configure the number of distinct priority levels from 1 to 32 (we consider two cases where N=16 and N=8). For each considered model, we evaluate the deployment cost when a particular RTOS is targeted. Fig.4 (a) shows the variation of the cost (in %) for the already mentioned RTOSs in function of the application (by increasing the priority levels number).



(a) Cost Evaluation For MicroC-OS/II and Ecos

(b) Evaluation of the Resolution Time in Seconds

Fig. 5. Evaluation of The Merge Pattern for Large Scale Applications

This evaluation shows that, in some cases, the deployment of a real time application requires some performance degradation, due the implementation constraints, to generate valid implementation models. This deployment cost is strongly influenced by the application and the target operating system. The merge pattern that we have proposed offers to the designer the possibility to estimate the performance loss for a particular application and a RTOS and thus provides a source of guidance for the selection of the appropriate operating system.

Fig.4 (b) illustrates the variation of the resolution time in seconds. In fact, in this graph we evaluate the execution time of the linear program using CPLEX for different applications and for the Ecos Operating system configured respectively with 8 and 16 authorized applicative priority levels. From these graphics (Fig.4 (b)) we can conclude that the time required by the linear program to make decision is bounded even for large scale applications. The second conclusion is that this time depends strongly on the application and the target RTOS. In fact, it increases when the number of distinct priority levels in the architectural model increases and the number of priority levels authorized by the RTOS decreases.

6 Related Work

Several approaches have been proposed to provide guidelines for the software development of RTES in a MDD context. In [10], the authors propose a generative process to transform an application deployed on one RTOS to another based on an explicit description of the involved RTOSs using SRM. This approach focuses especially on the portability requirement by proposing generic transformations enabling the deployment of the same application on several RTOSs. This work focuses on the structural aspect but makes the assumption that the deployment is always possible without any consideration of the potential difference between the semantics of RTOSs resources. The authors in [11] extend the previous work by introducing behavioural information in platforms description. This approach focuses on the separation of concerns and portability while ensuring an automatic full code generation. To achieve that, the authors introduce behavioural patterns in platform models for a detailed description of the different services offered by the target platform. Indeed, these previously mentioned works do not consider real-time validation.

In order to address real-time concerns, several works focus a specific standard and do not address the portability issue. In [13], the author extends the RT-UML profile to support the creation and validation of OSEK-compliant models. In [14], the authors use an OSEK-compliant abstract platform called SmartOSEK [15] and define a set of transformation rules to create OSEK-compliant models from UML models. In addition, this approach enables the simulation of the resulting OSEK-compliant models and provides the designer with the results to optimize this model at design level. In [16], the authors use RT-UML to annotate UML models describing real-time applications with timing properties. Then they identify the mapping rules between the resulting model and RT-Java

as a target platform. The objective of this work is to properly propagate the real-time constraints into the RT-java specific model in order to validate them. From the other side, many existing works define MDD approaches to guide the design choices and generate architectural models satisfying timing properties. In [12] authors provides an approach to automatically generate the architectural model from the functional blocks. The focus of this work is to automate this generation and ensure optimized architectural models in terms of timing properties. In [1] authors propose a MARTE-based methodology by introducing analysis from the functional level to guide the generation of a valid design model in terms of timing requirements. These works still keeping portability by ensuring platform-independent architectural models. However they in general end at the design level and do not focus on deployment issues. Hence, our approach aims at extending the latter methodology [1] by focusing on the refinement toward implementation of the resulting design model. This work is a step toward providing portability and separation of concerns from one side and early verification of timing properties from the other side during the deployment process of a real-time application on a several RTOSs.

7 Conclusion and Perspectives

In this paper we have proposed a model-driven approach to guide the transition from the design to the implementation model during the development of real-time applications. We have especially addressed the problem where the number of distinct priority levels used to validate the design model exceeds the number authorized by the RTOS. In that case, we have proposed a software pattern that we have called Distinct Priority Merge Pattern(DPMP) that automatically perform the re-factoring of the architectural model with the objective of solving the problem. The application of this pattern preserves the high level specification and the timing requirements while reducing the number of used distinct priority levels. Due to the complexity of this treatment, a MILP formulation of this pattern have been proposed. This formulation permits to confirm whether a solution exists for the problem and finds the better one in terms of processor utilization.

As perspective of this work, we aim at considering other problems such as timer granularity, equal priority levels, etc and proposing for each particular problem a software pattern to enrich our pattern base. In addition, we can extend this work by considering the behavioural aspect and thus other problems must be considered and consequently additional software pattern must be defined.

References

1. Mraidha, C., Tucci-Piergiovanni, S., Gerard, S.: Optimum: A MARTE-based methodology for Schedulability Analysis at Early Design Stages. In: Proceeding of the Third IEEE International Workshop UML and Formal Methods (UML&FM 2010), Shanghai, China (2010)

2. Singhoff, F., Legrand, J., Nana, L., Marc, L.: Cheddar: a Flexible Real Time Scheduling Framework. In: International ACM SIGADA Conference, Atlanta (November 2004)
3. Mzid, R., Mraidha, C., Babau, J.-P., Abid, M.: A MDD Approach for RTOS Integration on Valid Real-Time Design Model. In: The 38th Euromicro Conference On software Engineering and Advanced Applications (SEAA 2012), Cesme, Izmir, Turkey (September 2012)
4. Mzid, R., Mraidha, C., Babau, J.-P., Abid, M.: Real-Time Design Models to RTOS-Specific Models Refinement Verification. In: The 5th International Workshop on Model Based Architecting and construction of Embedded Systems ACES-MB 2012 in Conjunction with the 15th International Conference on Model Driven Engineering Languages & Systems MODELS 2012, Innsbruck, Austria (September 2012)
5. Object Management Group, UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems, Object Management Group, Inc. (September 2010) OMG document number: ptc/2010-08-32
6. Goodenough, J.B., Sha, L.: The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks, vol. 8. ACM (1988)
7. Klein, M.H., Ralya, T., Pollak, B., Obenza, R., Harbour, M.G.: A practitioners handbook for real-time analysis. Kluwer Academic Publishers (1993)
8. Labrosse, J.J.: MicroC/OS-II The Real-Time Kernel
9. Anthony. J. MASSA Embedded Software Development with Ecos
10. Thomas, F., Delatour, J., Terrier, F., Gerad, S.: Toward a framework for explicit platform-based transformations. In: Proceeding of the 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC), Orlondo, Florida, USA (May 2008)
11. Chehade, W.E.H., Radermacher, A., Terrier, F., Selic, B., Gerard, S.: A model-driven framework for the development of portable real-time embedded systems. In: Proceedings of the 16th IEEE International Conference on Engineering of Complex Computer Systems, pp. 45–54. IEEE Computer Society, Las Vegas (2011)
12. Bartolini, C., Lipari, G., Natale, M.D.: From functional blocks to the synthesis of the architectural model in embedded real-time applications. In: Proc. IEEE Real Time and Embedded Technology and Applications Symposium (RTAS), pp. 458–467 (2005)
13. Moore, A.: Extending the RT-prole le to support the OSEK infrastructure. In: Proceedings of the 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Washington, DC, pp. 341–347 (April 2002)
14. Yang, G., Zhao, M., Wang, L., Wu, Z.: Model-based Design and Verification of Automotive Electronics Compliant with OSEK/VDX. presented at The Secend International Conference on Embedded Software and System (ICESS), Xi'an (2005)
15. Zhao, M., Wu, Z., Yang, G., Wang, L., Chen, W.: SmartOSEK: A Dependable Platform for Automobile Electronics. In: Wu, Z., Chen, C., Guo, M., Bu, J. (eds.) ICESS 2004. LNCS, vol. 3605, pp. 437–442. Springer, Heidelberg (2005)
16. Becker, L.B., Holtz, R., Pereira, C.E.: On Mapping RTUML Specifications to RT-Java API: Bridging the Gap. In: 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, Washington, USA, pp. 348-355 (2002)

Using Model Types to Support Contract-Aware Model Substitutability

Wuliang Sun[1], Benoit Combemale[2], Steven Derrien[2], and Robert B. France[1]

[1] Colorado State University, Fort Collins, USA
[2] University of Rennes 1, IRISA, France

Abstract. Model typing brings the benefit associated with well-defined type systems to model-driven development (MDD) through the assignment of specific types to models. In particular, model type systems enable reuse of model manipulation operations (e.g., model transformations), where manipulations defined for models typed by a supertype can be used to manipulate models typed by subtypes. Existing model typing approaches are limited to structural typing defined in terms of object-oriented metamodels (e.g., MOF), in which the only structural (well-formedness) constraints are those that can be expressed directly in metamodeling notations (e.g., multiplicity and element containment constraints). In this paper we describe an extension to model typing that takes into consideration structural invariants, other than those that can be expressed directly in a metamodeling notation, and specifications of behaviors associated with model types. The approach supports contract-aware substitutability, where contracts are defined in terms of invariants and pre-/post-conditions expressed using OCL. Support for behavioral typing paves the way for behavioral substitutability. We also describe a technique to rigorously reason about model type substitutability as supported by contracts, and apply the technique in a usage scenario from the optimizing compiler community.

Keywords: SLE, Modeling Languages, Model Typing, Contract Matching, Model Substitutability.

1 Introduction

In Model Driven Engineering (MDE), developers of complex software systems create and transform models using model authoring and transformation technologies. The rise in the number of new modeling languages, however, presents a challenge because it requires software engineers to create complex transformations that manipulate models expressed in the new languages. Building these transformations from scratch requires significant effort. To address this problem, various approaches [1][2][3][4][5] have recently been proposed to facilitate the reuse of model transformation across different languages.

Model substitutability rules that are based on model typing [1] can be used to support model transformation reuse. For example, a subtyping relation that supports model substitutability allows a model typed by A to be safely used where a model typed by B is expected, where B is the supertype of A. The transformation used for models typed by B can thus be reused on models typed by A.

P. Van Gorp, T. Ritter, and L.M. Rose (Eds.): ECMFA 2013, LNCS 7949, pp. 118–133, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Current approaches to model type definition and implementation, however, only consider MOF-based metamodels as model types. In MOF, contracts (e.g., pre-conditio-ns, post-conditions and invariants) are externally defined, that is, they are defined in another language, for example, the Object Constraint Language (OCL) [6]. Neither the original paper on model typing [1] nor the follow-up paper [7] considers externally defined contracts in subtyping relations. This limits the utility of model subtyping in model-based software development approaches that are contract based (e.g., design by contract [8]). There is thus a need for model typing that provides support for typing models with contracts.

In this paper we propose a form of model typing that supports contract-aware substitutability, where contracts are defined in terms of invariants and pre-/post-conditions expressed using OCL. We add invariants to model types that specify additional structural properties, and use operation pre-/post-conditions to specify the transformation rules on model types. We also describe a technique for rigorously reasoning about the substitutability of models with contracts.

The rest of the paper is organized as follows. Section 2 illustrates the need for contract-aware substitutability using motivating examples from the high-performance embedded system design domain. Section 3 presents background material needed to understand the work described in this paper. Section 4 presents a formal definition of the subtyping relation between two model types that include contracts, and describes tool support for reasoning about substitutability on model types. Section 5 describes limitations of the approach. Section 6 discusses related work, and Section 7 concludes the paper with a discussion of planned future work.

2 Motivating Examples

In this section we describe two motivating examples from the high-performance embedded system design domain. Modern heterogeneous embedded hardware platforms are notoriously difficult to design and to program. In this context, tool-supported model based approaches (e.g., Simulink, Ptolemy) are now widely acknowledged as some of the most effective approaches to designing embedded systems.

Typically, these model-based approaches use tool chains that manipulate many different types of models. For example, structural platform description models range from system level models that abstract over processing and storage resource with their interconnections, to very low level Register-to-Logic level circuit models that are used to describe the structure of hardware accelerators within the platforms.

Similarly, behavioral description models range from application level modeling of the application using Models of Computation such as Synchronous Data Flow Graphs or Kahn Process Networks, to fine grain scalar operation level representations such as the basic-block level instruction dependence graph used in an optimizing compiler back-end.

Most of these tool chains share a common goal: They aim to produce highly optimized implementations. This requires the use of advanced algorithms that implement very complex model manipulations. It is also the case that these manipulations often have similar algorithmic patterns. These patterns can be used as the basis for developing reusable model transformations.

2.1 Example 1: Using Model Types to Support Structural Substitutability

In the optimizing compiler domain, a variety of models describing different aspects of languages are manipulated (i.e., analyzed and transformed) at different stages of the compilation process. While the analyses and transformations are different, they also share many common characteristics. For example, consider algorithms for schedule optimization. Obtaining an optimized implementation of an application on a target platform involves performing several static scheduling optimizations. Many of these algorithms have common characteristics, for example, they are often expressed as an acyclic graph resource constrained scheduling problem, for which many techniques (heuristics or MILP-Mixed Integer Linear Programming-solver based) have been proposed. Because these scheduling algorithms involve very sophisticated algorithms, reusable algorithms that can be tailored to the different types of representations (models) are highly desirable. For example, it would be useful to have a reusable scheduling algorithm that can be used to derive a schedule for an Application level Synchronous Data-flow graph on a multi-processor based implementation, as well as for generating efficient code for a customized VLIW (Very Long Instruction Word) embedded processor.

However in this case, structural substitutability based only on constraints that can be expressed directly in a metamodel (e.g., multiplicity or element containment constraint) is not sufficient; other structural constraints need to be specified. For example, a classical static scheduling toolset can only operate on acyclic dependence graphs and the acyclicity property cannot be expressed directly in a metamodel. A language such as the Object Constraint Language (OCL) is needed to specify properties of acyclic graphs. In this case, model substitutability requires that a substitute model enforces the acyclicity constraint expressed in OCL. Model typing based on metamodels with OCL constraints can be used to enable such structural substitutability.

2.2 Example 2: Using Model Types to Support Contract-Based Behavioral Substitutability

Behavioral Substitutability for Model Transformations: A consistent scheduling tran-sformation must ensure that every node in the dependence graph is scheduled *at least* once. This property can be expressed as a post-condition on the scheduling transformation and thus any scheduler implementation should enforce this post-condition. The effective post-condition could even be stricter; in our case we could consider a post-condition that restricts a node to be scheduled *exactly* once.

The same holds for the pre-condition. For example, most schedulers operate on acyclic graphs and this can be translated as a pre-condition for the transformation. However, there also exists a class of pipelined schedulers that operate on cyclic graphs, in which cycles implement delays to preserve causality. For such pipelined schedulers, the pre-condition would not forbid cycles in the dependence graph. That would, however, prevent a pipelined scheduler from being used to schedule acyclic graphs in a design flow.

Contract Based Tool Chain Validation: An optimizing compiler custom tool chain consists of a sequence of analyses and transformations (called compiler passes)

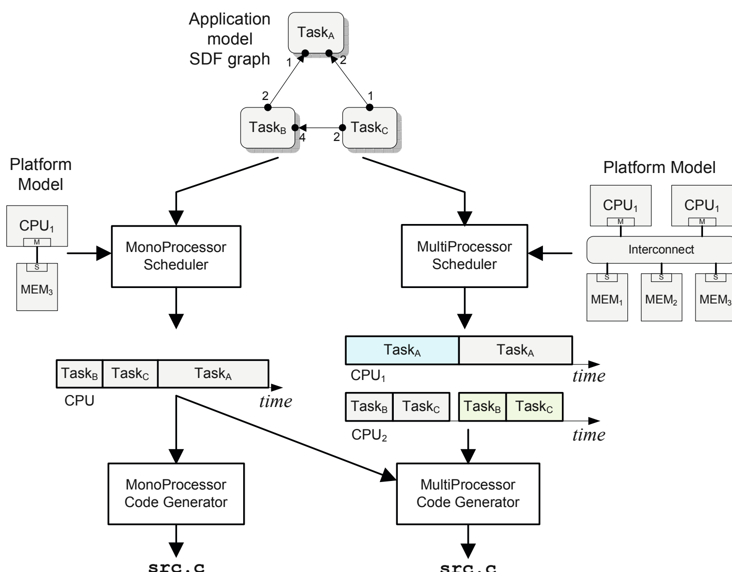


Fig. 1. A Model based compiler tool chain for embedded multiprocessors

executed in a very carefully chosen order. They can hence be seen as a model transformation chain. Compiler passes cannot be combined arbitrarily, as each pass usually assumes that the program representation at hand has very specific properties.

For example, consider a compiler tool chain for generating software code from Synchronous Data Flow Graph (SDF) model specifications on an embedded platform. Such a tool chain is illustrated in Figure 1. Before any code can be produced, the SDF first needs to be scheduled on this platform. Depending on whether the target system consists of a single or several processors, it is likely that different scheduling algorithms will be used. Similarly, different code generators (i.e. pretty printers) will have to be used depending on whether we target a mono-processor or multiprocessor. Two back-end code generators are shown in Figure 1. The mono-processor code generator can only be used after a mono-processor scheduling stage, whereas the second back-end is more general and can be used for both types of scheduling.

These constraints – that is targeting one or several processing resources – apply to the result of the scheduling stage and to the input on the code generation stage. They can hence be modeled as pre-conditions (resp. post-conditions) expressed using OCL. When chaining a given scheduling and code generation pass, we can ensure the consistency of the flow by checking if the pre-/post-conditions of two chained transformation are satisfiable.

3 Background

In this section, we describe the concepts underlying our use of model types to support model substitutability. We first present the MOF (Meta-Object Facility) meta-language,

the basis for metamodels, and thus model manipulation operators. We then give an overview of model types as currently defined and implemented [1,7], and describe the limitations addressed by the approach presented in this paper.

3.1 Metamodeling

The Meta-Object Facility (MOF) [9] 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 modeling languages and tools. A metamodel defines a set of models on which it is possible to apply common operators. The model substitutability approach presented in this paper is applicable to models expressed in languages with MOF metamodels.

MOF supports the definition of metamodels using `Classes` and `Properties`. `Classes` can be abstract (i.e., they cannot be instantiated) and have `Properties` and `Operations`, which respectively declare attributes and references, and the signatures of methods available to the modeled concept. 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 `Properties`) and read-only (i.e., cannot be modified). A `Property` can also have an opposite `Property` with which it forms a bidirectional association.

Metamodels can be viewed as class diagrams in which each metamodel element can be instantiated to obtain objects representing model elements. However, metamodel elements are themeslves 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., `Class` or `Property` classes).

3.2 Model Typing

Model Types were introduced by Steel *et al.* [1], as an extension of object typing to provide abstractions about the object type level and enable the reuse of model manipulation operators. Informally, a model type is a substructure (referred to as a *type group*) of the metamodel's class diagram. It is important to distinguish the usage of the term metamodel from model type. We use the term metamodel to refer to the class diagram used to define a language, and when the same class diagram is used to define the type of a model it is called an *exact type*. It is also important to note that a model has one and only one metamodel to which it must conform, but the same model can have several model types, where each model type is a substructure of the metamodel. Because model types and metamodels share the same structure, it is possible to extract the exact type of a model from its metamodel. Figure 2 represents a model m_1 that conforms to a metamodel MM_1 and is typed by model types MT_A and MT_B, where MT_B is the *exact type* of m_1 that is extracted from MM_1. Both metamodels and model types conform to MOF. Given the above, a model type can be defined as follows:

Definition 1. (Model type) *A model type is a substructure of a metamodel's class structure. A model does not have to include instantiations of each class in an associated model type, that is, the set of classes of elements in a model can be smaller than the classes in its model type.*

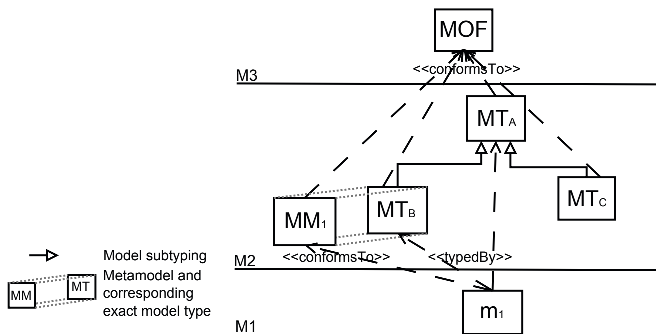


Fig. 2. Conformance, model typing and model subtyping relations

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 specializations of model substructures (or *type groups*)[1]. One way to safely reuse a model manipulation operation created for a model typed by MT_A on a model typed by MT_B is to ensure that MT_A contains elements that can be substituted by elements defined by MT_B. However, it is not possible to achieve model type substitutability through object subtyping. Thus, model typing uses an extended definition of *object type matching* introduced by Bruce *et al.* [11], namely *MOF Class Matching*.

Definition 2. (MOF class matching) *MOF class T' matches T (written $T' <\# T$) iff their names are equal, and for each property (respectively method) in T there is a corresponding property (respectively method) in T'.*

The *MOF class matching* relation can be seen as a kind of *object type matching* relation that is tailored to MOF concepts. Based on the *MOF class matching* relation, we can achieve model type substitutability by defining a subtyping relation as follows:

Definition 3. (Subtyping relationship for model types) *The model type* subtyping *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 $T_B <\#\ T_A$.*

We recently introduced four extended subtyping relations between model types that take into account two additional criteria: The presence of heterogeneities between two model types (using adaptation) and the considered subset of the model types (using model type pruning) [7].

The *subtyping* relation as currently defined has shortcomings. In particular, the current model typing definition and implementation only considers MOF-based metamodels as model types (through the *MOF class matching* relation). Unfortunately, MOF delegates the definitions of contracts (*e.g.*, pre and post-conditions or invariants) to other

[1] For further information on type groups see Ernst's paper [10].

languages (*e.g.*, OCL, the Object Constraint Language [6]). This limits the applicability of model typing for safely reusing model manipulations where OCL contracts are needed to precisely specify the applicability of the model transformation or the structure on which the model transformation can be applied (see motivating examples in Section 2). The approach described in this paper addresses this limitation.

4 Contribution

In this paper we extend the subtyping relation described in [7] by taking into account OCL contracts for a safe substitutability of models conforming to metamodels including contracts. This provides a safe reuse of model transformations expressed on metamodels that include contracts. Specifically, we extend the MOF class matching (cf. Def. 2 of Section 3) by considering contracts matching (Section 4.1) and provide a technique for analyzing the matching of OCL contracts associated with two classes with different model types (Section 4.2). In this section we describe the contract matching technique we developed to support contract-aware model substitutability. We also describe an Alloy-based prototype tool that supports contract matching (Section 4.3), and illustrate the use of contract-aware substitutability using the motivating examples (Section 4.4).

4.1 Contract-Aware MOF Class Matching

We consider the use of OCL invariants added to MOF classes to specify additional structural properties, and OCL pre-/post-conditions defined in the context of MOF class operations to specify the model manipulation rules (e.g., transformation) associated with model types. The MOF class matching relation is thus determined by two aspects: the structural features specified using MOF (e.g., classes, properties, operation signatures, etc.) and the contracts expressed using OCL (e.g., invariants and pre-/post-conditions).

The substitutability through model subtyping is a specialization of the Liskov Substitution Principle [12] on the model type system. Specifically the contract matching relation that enables contract-aware model substitutability must abide by the following rules: (1) invariants of the supermodel type cannot be weakened in a sub model type, (2) pre-conditions cannot be strengthened in a sub model type, and (3) post-conditions cannot be weakened in a sub model type. The extended MOF class matching relation is formalized as follows:

Definition 4 (Contract-aware MOF Class Matching). *Class T' matches T (written T' <# T) iff their structures match (cf. Def. 3 of [7]), their invariants match and their operation pre-/post-conditions match, where*

1. ***Invariants Match*** *is defined as follows:*
   let T.ownedInvariant = $\{inv_{T1}, inv_{T2}, ..., inv_{Tk}\}$ *be the invariants defined for T;*
   let $result_T = inv_{T1} \wedge inv_{T2} \wedge ... \wedge inv_{Tk}$;
   let SuperClass(T) = $\{cls_1, cls_2, ..., cls_n\}$ *where* cls_i *is a superclass of T;*
   let cls_i*.ownedInvariant =* $\{ inv_{i1}, inv_{i2}, ..., inv_{ik} \}$ *be the invariants defined for* cls_i*, for i = 1, .., n;*
   let $result_i = inv_{i1} \wedge inv_{i2} \wedge ... \wedge inv_{ik}$*, for i = 1, .., n;*
   let $invs = result_1 \wedge result_2 \wedge ... \wedge result_n \wedge result_T$;

let T'*.ownedInvariant* = $\{inv'_{T1}, inv'_{T2}, ..., inv'_{Tk}\}$ *be the invariants defined for* T'*;*
let $result'_T = inv'_{T1} \wedge inv'_{T2} \wedge ... \wedge inv'_{Tk}$*;*
let SuperClass(T'*)* = $\{cls'_1, cls'_2, ..., cls'_n\}$ *where* cls'_i *is a superclass of* T'*;*
let cls'_i*.ownedInvariant* = $\{inv'_{i1}, inv'_{i2}, ..., inv'_{ik}\}$ *be the invariants defined for* cls'_i*, for i = 1, .., n;*
let $result'_i = inv'_{i1} \wedge inv'_{i2} \wedge ... \wedge inv'_{ik}$*, for i = 1, .., n;*
let $invs' = result'_1 \wedge result'_2 \wedge ... \wedge result'_n \wedge result'_T$*;*

The invariants of T and T' *match if Models(invs)* $\supseteq$ *Models(*$invs'$*), where Models(invs) returns all models that satisfy invs and Models(*$invs'$*) returns all models that satisfy* $invs'$*.*

2 ***Pre-/post-conditions Match*** *is defined as follows:*
$\forall$ *op* $\in$ *T.ownedOperation,* $\exists$ S' $\in$ *SuperClasses(*T'*) such that* $\exists$ op' $\in$ S'*.ownedOperation and:*

2.1 *let op.ownedPrecondition* = $\{pre_1, pre_2, ..., pre_k\}$ *be the pre-conditions defined for op;*
let pres = $pre_1 \wedge pre_2 \wedge ... \wedge pre_k$*;*
let op'*.ownedPrecondition* = $\{pre'_1, pre'_2, ..., pre'_k\}$ *be the pre-conditions defined for* op'*;*
let $pres' = pre'_1 \wedge pre'_2 \wedge ... \wedge pre'_k$*;*

2.2 *let op.ownedPostcondition* = $\{post_1, post_2, ..., post_k\}$ *be the post-conditions defined for op;*
let posts = $post_1 \wedge post_2 \wedge ... \wedge post_k$*;*
let op'*.ownedPostcondition* = $\{post'_1, post'_2, ..., post'_k\}$ *be the post-conditions defined for* op'*;*
let $posts' = post'_1 \wedge post'_2 \wedge ... \wedge post'_k$*;*

The operation specifications of T and T' *match if Models(*$pres'$*)* $\supseteq$ *Models(pres) and Models(posts)* $\supseteq$ *Models(*$posts'$*)*

4.2 Analyzing the Matching of Contracts

Definition 4 can be used to formally reason about the matching relation between two MOF classes with contracts. The MOF class matching relation in Definition 4 includes the matching of the contracts from classes of two model types. Consequently, analyzing such relations requires one to formally analyze the relation between contracts (e.g., to check if the models satisfying one contract includes the models satisfying the other). To do this, a query function $Models(MT, C)$ is used to compute all models that both conform to a model type MT and satisfy an OCL contract C defined in MT. Thus given contract C_1 in a candidate supermodel type MT_1 and contract C_2 in a candidate sub

model type MT_2, C_1 matches C_2 iff (1) C_1, C_2 are invariants, and $Models(MT_1, C_1) \supseteq Models(MT_2, C_2)$, (2) C_1, C_2 are pre-conditions, and $Models(MT_2, C_2) \supseteq Models(MT_1, C_1)$, and (3) C_1, C_2 are post-conditions, and $Models(MT_1, C_1) \supseteq Models(MT_2, C_2)$.

Checking the contract matching requires a tool to implement the functionality of the query function $Models(MT, C)$. We use the Alloy Analyzer [13] for this purpose. The Alloy Analyzer is used to analyze Alloy specifications. It is supported by a SAT-based model finder. The Alloy Analyzer can generate models that conform to a model type expressed in Alloy in terms of signatures and fields that specify the model type structure and a predicate that expresses the contracts. In this paper we use the Alloy Analyzer at the back-end to check whether two contracts match.

For example, given a candidate supermodel type MT_1 and a candidate sub model type MT_2, with two OCL invariants respectively, C_1 and C_2, the procedure below can be used to check if C_1 matches C_2.

1. (preprocess) Since model subtyping requires each element in the supermodel type to be matched by an element in the sub model type (see Definition 4), the contract defined in the supermodel type refers to elements that also exist in the sub model type. Thus we can move C_1 to MT_2, and use only the sub model type (i.e., MT_2) to check whether C_1 and C_2 match.
2. Transform MT_2 to an Alloy model using the technique described in [14]. Convert C_1 and C_2 into two Alloy predicates, P_1 and P_2, respectively.
3. Run an empty predicate in the Alloy Analyzer to search for a model conforming to the model type MT_2. If the Analyzer returns no model satisfying the empty predicate (i.e., $Models(MT_2, \emptyset) = \emptyset$), $Models(MT_2, C_1) = \emptyset$ and $Models(MT_2, C_2) = \emptyset$. In this case C_1 matches C_2 since $\emptyset$ is a subset of $\emptyset$; otherwise, continue to the next step.
4. Run P_1 and P_2 respectively. If the Alloy Analyzer returns no model for each predicate (i.e., $Models(MT_2, C_1) = \emptyset$ and $Models(MT_2, C_2) = \emptyset$), then C_1 matches C_2; if the Alloy Analyzer returns a model (or models) for only P_1, then C_1 matches C_2; if the Alloy Analyzer returns a model (or models) for only P_2, then C_1 does not match C_2; otherwise, continue to the next step.
5. Run a predicate to search for a model satisfying both P_1 and P_2. If the Alloy Analyzer returns a model satisfying the predicate, continue to the next step; otherwise, C_1 does not match C_2.
6. Run a predicate P_3 to search for a model satisfying both P_1 and $\neg P_2$ (i.e., the negation of P_2), and another predicate P_4 to search for a model satisfying both P_2 and $\neg P_1$. If the Alloy Analyzer returns no model for both P_3 and P_4 (i.e., $Models(MT_2, C_1) = Models(MT_2, C_2)$), C_1 matches C_2; if the Alloy Analyzer returns a model (or models) satisfying only P_3, $Models(MT_2, C_1) \supset Models(MT_2, C_2)$ and C_1 matches C_2; otherwise, C_1 does not match C_2.

The approach uses the Alloy Analyzer at the back-end to analyze the relation between two contracts, and it thus requires a translation from OCL expressions to Alloy specifications. The OCL to Alloy translation used in the prototype tool we developed is based on translation rules described in work by Bordbar et al. [15].

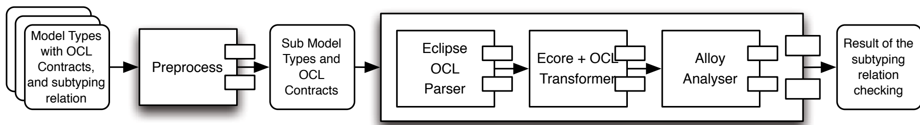


Fig. 3. Contract Matching Checking Tool Overview

4.3 Contract Matching Checking Tool

The contract matching approach described in the previous subsection has been implemented in a prototype tool. Figure 3 shows an overview of the prototype tool. It consists of an OCL parser, an Ecore[2]/OCL transformer and the use of the Alloy Analyzer. The Ecore/OCL transformer is developed using Kermeta [17], an aspect-oriented metamodeling tool. The inputs of the prototype are (1) an Ecore file that specifies two model types, and (2) a textual OCL file that specifies the contracts from each model type. The model types and contracts are automatically transformed to an Alloy model consisting of signatures and predicates.

The prototype provides several interfaces to check contract matching. For example, *matchInv(inv1: Constraint, inv2: Constraint)* is used to check whether $inv1$ matches $inv2$. In addition, *matchInvs(cls1: Class, cls2: Class)* can be used to check whether the invariants defined in $cls1$ and the invariants defined in $cls2$ match.

4.4 Case Study

In this section we illustrate how to use our approach to define model types and subtyping relations between them to ensure a safe reuse of model transformations.

A Simple Case Study of Structural Substitutability. Let us reconsider the scheduling example described in Section 2.1. A model transformation performs a static scheduling on an acyclic dependence graph. The model transformation needs a metamodel for "Acyclic Graph" (due to space limitation, the metamodel is not shown in the paper). The model type *AcyclicGraph* (see Figure 4) shows a simple example of model type definition for the dependency graph used in the example. Its definition consists of metaclasses that specify a graph structure, an invariant that specifies the acyclicity property, and a model transformation that takes as input an acyclic graph.

Suppose that in another context a colored graph is used as an intermediate representation and it extends the concept of nodes by introducing additional information. To reuse the transformation defined in *AcyclicGraph*, a colored graph must be a subtype of *AcyclicGraph*. The model type *ColoredAcyclicGraph* ensues the subtyping relation by adding an acyclicity invariant in its definition. However, the model type *ColoredGraph* does not specify any invariants. A compilation error will thus show that the $transfo$ operation cannot take as input an instance of *ColoredGraph* because *ColoredGraph* is not a subtype of *AcyclicGraph*.

[2] Ecore is an implementation aligned with MOF included in the Eclipse Modeling Framework [16].

```
modeltype AcyclicGraph{
        // Model structure (Graph, Node, Edge)
        ...
        // Invariant in the context of Graph
        // specifying that the graph is acyclic
        ...
        // Reusable model transformation
        transfo() :Void is
        do ··· end
}

modeltype ColoredAcyclicGraph{
        // Model structure (Graph, Node, Edge)
        ...
        // Invariant in the context of Graph
        // specifying that the graph is acyclic
        ...
}

modeltype ColoredGraph {
        // Model structure (Graph, Node, Edge)
        ...
}

modeltype Main
{
        main() : Void is do
                var m1 : ColoredGraph init
                        ColoredGraph.new
                var m2 : ColoredAcyclicGraph init
                        ColoredAcyclicGraph.new

                m1.transfo() // not OK
                m2.transfo() // OK
        end
}
```

Fig. 4. A Simple Example of Structural Substitutability in Kermeta

```
modeltype MT{
        // Model structure and invariant
        ...
        // Abstract transformation
        transfo() : MT is abstract
                post:   SSA // OCL expression
        // Reusable model transformation
        transfo2() : MT is do ··· end
                pre:    SSA
}

modeltype subMT1{
        // Model structure and invariant
        ...
        // Transformation specialization
        transfo() : subMT1 Is
        do ··· end
                post:   SSA
}

modeltype subMT2{
        // Model structure and invariant
        ...
        // Transformation specialization
        transfo() : subMT2 Is
        do ··· end
}

modeltype Main
{
        main() : Void is do
                var m1 : subMT1 init
                        subMT1.new
                var m2 : subMT2 init
                        subMT2.new

                m1.transfo().transfo2() // OK
                m2.transfo().transfo2() // not OK
        end
}
```

Fig. 5. A Simple Example of Behavioral Substitutability in Kermeta

A Simple Case Study of Behavioral Substitutability. In the optimizing compiler community, the daily task for software engineers is to design compilation chains in the right partial order, that is, scheduling the various passes (i.e., optimization, translation, code generation, analysis, etc.). Designing compilation chains would benefit from behavioral

substitutability by opening the way to describe "abstract" compilation chains, capitalizing a given knowledge in terms of constraints (pre-/post-conditions) to schedule a set of passes for a given purpose, where each pass would be then implemented in various ways, but conforming to the pre-/post-conditions defined in the abstract compilation chain.

Figure 5 shows a simple example of model types used for the compilation chain. Suppose that the abstract model transformation *transfo* defined in MT is used for optimization purpose and define a post condition stating that the model must conform to the Static Single Assignment (SSA) form. MT also contains *transfo2* as the next pass of the compilation chain and states as precondition that the model must conform to the SSA form. The two model types $subMT1$ and $subMT2$ implement the model transformation *transfo* but only $subMT1$ ensures as postcondition the SSA form. While $subMT2$ is not in this case a sub model type to MT, a compilation error for $m2.transfo().transfo2()$ will be returned. This shows that the model returned by *transfo* in $subMT2$ (typed by $subMT2$) is not of type MT, and can not reuse *transfo2*.

5 Discussion

In this section we discuss limitations of our work, and its scope of application. We first discuss the supported contracts in the subtyping relation of model typing (Section 5.1), and the corresponding model substitutability provided by our approach (Section 5.2).

5.1 On the Support of Contracts in Model Typing

In this paper we consider contracts in addition to the object oriented structure described in a metamodel. The object-oriented structure is usually defined using Ecore, an implementation aligned with OMG MOF. Contracts can then be invariants expressed in the context of the concepts (i.e., classes) defined in the MOF metamodel, and pre-/post-conditions expressed in the context of operations specified in concepts. While invariants restrict the structure of conforming models and their possible structural substitutability, pre- and post-conditions specify the behavior of the conforming models (i.e. manipulation by model operations) and their possible behavioral substitutability.

In our approach, we assume that the first order logic is used to express contracts in metamodels, and we have chosen OCL to express them. We rely on the provided binding between MOF and OCL as defined by OMG to link OCL expressions to a given MOF metamodel.

To test the feasibility of our approach, we implemented a prototype tool that is integrated into the Kermeta workbench. The tool checks OCL-based contract-aware subtyping relations between model types. While the substitutability related to the MOF structure is computed directly using Kermeta, the one related to the contracts is computed using Alloy through a translation from OCL expressions to Alloy specifications, and then an analysis of the output provided by the Alloy Analyzer. The tool only provides support for translating a subset of OCL to Alloy.

Most of the OCL operators have corresponding Alloy constructs. For example, OCL operator $forAll$ corresponds to Alloy construct *all*, $exists$ corresponds to *some*, $includes$ corresponds to *in*, $excludes$ corresponds to *!in*, *sum* corresponds to *sum*, and $closure$

corresponds to $*$. OCL contracts that involves such operators can be directly transformed into Alloy specifications.

However, as pointed out by Anastasakis et al. [15], the translation from OCL to Alloy is not seamless. There are some OCL operators that do not have corresponding Alloy constructs, and thus OCL contracts including such operators cannot be easily transformed into Alloy specifications. Some of them can be partially supported by the tool using the Alloy libraries. For instance, OCL operators like *select* and *collect* are translated by the tool described in the paper using Alloy functions that implement their semantics. Consequently, the operator *iterate* is partially supported by the transformation tool. The tool provides support for OCL contracts including *iterate* expressions that can be rewritten as *forall* with *select*/*collect* operators. However, the tool cannot be used to deal with *iterate* expressions that involve arithmetic accumulation since Alloy is a purely declarative language that does not provide support for imperative accumulators. Finally, the translation cannot deal with OCL casting operators like *oclAsType* since Alloy has a very simple type system that has little support for type casting.

5.2 On the Support of Modeling Language Substitutability

The research work described in the paper builds upon our previous work in [7], and paves the way for reasoning about the subtyping relation between two model types that include contracts. Specifically it can be used to reason about the contract-aware subtyping relation that involves structural subtyping (including not only MOF-based Object-Oriented structure but also OCL-based first order invariants) and behavioral subtyping (including a behavioral semantics in an axiomatic way using pre-/post-conditions on operations).

We implement our approach in a (Kermeta-based) tool included in the Kermeta language workbench to check advanced (i.e., including contracts) subtyping relations between modeling languages based on Ecore and OCL. These two meta-languages are supported by the Kermeta language workbench and are used for describing the abstract syntax and the static semantics respectively.

This approach and its corresponding implementation addresses the need illustrated by the motivating examples from the high-performance embedded system community used throughout the paper. The scope of the structural substitutability we offer is bounded by OCL and its translation to Alloy, and its applicability is well founded, e.g., in model transformation reuse in model-driven development [7].

The actual scope of behavioral substitutability is more difficult to define. The difficulty is twofold: while we support the motivating examples described in the paper, complex situations of OCL based typing could be considered, such as type propagation in model transformation chains. Such challenges will be addressed in future work. Moreover, the scope itself of behavioral substitutability is more difficult to delimit.

6 Related Work

The technical contribution of this paper is the integration of contract matching in the subtyping relation of model typing to enhance the substitutability supported between

modeling languages. As discuss in the previous section, we rely for that on the most established translation to Alloy. Then, the work related to our contribution discussed in this paper is the applicability of the substitutability as illustrated in the motivating examples, namely on model transformation reuse.

Substitutability is supported through subtyping in object-oriented languages, including the support of contracts (*e.g.*, Eiffel [18]). However, object subtyping does not handle type group specialization (*i.e.*, the possibility to specialize relations between several objects and thus groups of types)[3]. Such type group specialization have been explored by Kühne in the context of MDE [19]. 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.

Several approaches have been proposed during the last decade for model transformation reuse. Strict substitutability relation, such as the first version of model type matching presented in [1], offers the possibility to reuse model transformation through isomorphic metamodels, i.e., metamodels with MOF-based equivalent structures. Such possibility was first proposed in [2] where the authors introduce *variable entities* in patterns for declarative transformation rules. These entities express only the needed concepts (*e.g.*, types, attributes, etc.) to apply the rule, allowing any tokens with these concepts to match the pattern and thus to be processed by the rule. Latter, Cuccuru *et al.* introduced the notion of semantic variation points in metamodels [3]. Variation points are specified through abstract classes, defining a *template*, and metamodels can fix these variation points by binding them to classes extending the abstract classes. Patterns containing *variable entities* and *templates* can be seen as kinds of model types where the variability has to be explicitly expressed and thus anticipated. Sanchez Cuadrado *et al.* propose in [4] a notion of substitutability based on model typing and model type matching, but rather to use an automatic algorithm to check the matching between two model types, they propose a DSL that allows users to declare the matching by hand. Finally, De Lara *et al.* present in [5] the *concept* mechanism, along with *model templates* and *mixin layers* leveraged from generic programming to MDE. *Concepts* are really close to model types as they define the requirements that a metamodel must fulfill for its models to be processed by a transformation, under the form of a set of classes. The authors also propose a DSL to bind a metamodel to a *concept* and a mechanism to generate a specific transformation from the binding and the generic transformation defined on the *concept*.

In the context of model transformation chains, existing approaches deal with explicit relationships between model transformations. Vanhooff et al. [20] proposed a domain specific language to model and execute a transformation chain. Aranega et al. [21] used feature models to classify model transformations involved a transformation chain and specified the constraints between them. The user thus can design a new transformation chain by reusing the classified transformations. Yie et al. [22] advocated the use of several independently developed model transformation chains to convert a high-level model into a low-level model. The interoperability among model transformation chains

[3] We refer the reader interested in the type group specialization problem to the Ernst's paper [10].

is achieved by deriving correspondence relationships between the final models generated by each model transformation chain.

Unlike the above approaches, contract-aware model subtyping offers a unified and formal type theory to facilitate the safe reuse of model transformations involved in a transformation chain. It follows a declarative fashion to specify a model transformation chain in an abstract way using pre-/post-conditions on abstract model types. This promotes then the reuse of various implementations that match the conditions for a safe execution of the model transformation chain.

7 Conclusion and Perspective

We propose in this paper a model typing theory where model types include contracts. This includes a formally defined subtyping relation between model types, and a tool-supported approach supporting a safe contract-aware substitutability of models conforming to metamodels including contracts. This ensures a safe reuse of model transformations expressed on metamodels including contracts.

Contracts are defined in terms of invariants and pre-/post-conditions expressed using OCL on MOF-based metamodels. The invariants are added on the classes of a metamodel to specify additional structural properties of the metamodel, and pre-/post-conditions are added on the operations of classes to specify model transformations. Consequently, the support of invariants in the subtyping relation ensures a safe reuse of model transformations where OCL contracts are needed to precisely specify the structure on which the model transformation can be applied. The support of pre-/post-conditions paves the way for behavioral substitutability to safely reuse model transformations where OCL contracts are needed to precisely specify the applicability of the model transformation.

The subtyping relation is based on a matching relation between two MOF classes that include OCL contracts, and is checked thanks to a technique based on Alloy. The actual scope of the provided contract-aware substitutability is mainly determined by the OCL-to-Alloy translation.

We are currently extending the prototype by providing support for model types and contracts expressed using the Kermeta language workbench. We also explore how we can extend the approach by using SMT solvers at back-end to analyze the OCL contracts that include more complex arithmetic calculation.

Acknowledgment. This work was supported by the National Science Foundation grant (CCF-1018711), the ANR INS Project GEMOC (ANR-12-INSE-0011), and the CNRS PICS Project MBSAR.

References

1. Steel, J., Jézéquel, J.M.: On model typing. SoSyM 6(4) (2007)
2. 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)

3. 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)
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. de Lara, J., Guerra, E.: Generic meta-modelling with concepts, templates and mixin layers. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 16–30. Springer, Heidelberg (2010)
6. OMG: UML Object Constraint Language (OCL) 2.0 Specification (2003)
7. Guy, C., Combemale, B., Derrien, S., Steel, J.R.H., Jézéquel, J.-M.: On Model Subtyping. In: Vallecillo, A., Tolvanen, J.-P., Kindler, E., Störrle, H., Kolovos, D. (eds.) ECMFA 2012. LNCS, vol. 7349, pp. 400–415. Springer, Heidelberg (2012)
8. Meyer, B.: Applying design by contract. Computer 25(10), 40–51 (1992)
9. OMG: Meta Object Facility (MOF) 2.0 Core Specification (2006)
10. Ernst, E.: Family polymorphism. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 303–326. Springer, Heidelberg (2001)
11. Bruce, K.B., Schuett, A., van Gent, R., Fiech, A.: Polytoil: A type-safe polymorphic object-oriented language. ACM TOPLAS 25(2) (2003)
12. Liskov, B., Wing, J.: A behavioral notion of subtyping. ACM Transactions on Programming Languages and Systems (TOPLAS) 16(6), 1811–1841 (1994)
13. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Transactions on Software Engineering and Methodology (TOSEM) 11(2), 256–290 (2002)
14. Sun, W., France, R., Ray, I.: Rigorous analysis of uml access control policy models. In: IEEE POLICY, pp. 9–16 (2011)
15. Anastasakis, K., Bordbar, B., Georg, G., Ray, I.: On challenges of model transformation from uml to alloy. Software and Systems Modeling 9(1), 69–86 (2010)
16. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Addison-Wesley Professional (2008)
17. Muller, P.-A., Fleurey, F., Jézéquel, J.-M.: Weaving executability into object-oriented meta-languages. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 264–278. Springer, Heidelberg (2005)
18. Meyer, B.: Design by contract. the eiffel method. In: Proceedings of the Technology of Object-Oriented Languages, TOOLS 26, pp. 446–446 (1998)
19. Kühne, T.: On model compatibility with referees and contexts. Software & Systems Modeling, 1–14 (2012)
20. Vanhooff, B., Ayed, D., Van Baelen, S., Joosen, W., Berbers, Y.: Uniti: A unified transformation infrastructure. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 31–45. Springer, Heidelberg (2007)
21. Aranega, V., Etien, A., Mosser, S.: Using feature model to build model transformation chains. Model Driven Engineering Languages and Systems, 562–578 (2012)
22. Yie, A., Casallas, R., Deridder, D., Wagelaar, D.: Realizing model transformation chain interoperability. Software and Systems Modeling, 1–21 (2012)

Applying a Def-Use Approach on Signal Exchange to Implement SysML Model-Based Testing

Fabrice Ambert[1], Fabrice Bouquet[1], Jonathan Lasalle[1], Bruno Legeard[1,2], and Fabien Peureux[1,2]

[1] FEMTO-ST Institute, UMR CNRS 6174, Besancon, France
{fambert,fbouquet,jlasalle,blegeard,fpeureux}@femto-st.fr
[2] Smartesting R&D Center, Besancon, France
{legeard,peureux}@smartesting.com

Abstract. Model-Based Testing (MBT) uses a model of the System Under Test as reference to automatically derive test cases. Since it is often not reasonable to cover all the behaviours formalized in the model, coverage criteria are applied to select a relevant subset of model behaviours. In this paper, we propose a dedicated test coverage criterion, based on Def-Use criteria on signal exchange, to implement MBT approach from *Systems Modeling Language* (SysML) test models to validate mechatronic systems. This novel criterion is introduced and the relevance of the approach from SysML models is discussed regarding results obtained with a dedicated MBT toolchain implementing this criterion.

Keywords: Model-Based Testing, UML/SysML notations, coverage criteria, mechatronic systems, toolchain experimentation.

1 Introduction

Model-Based Testing (MBT) refers to the processes and techniques dealing with the automatic derivation of abstract test cases (including stimuli and expected outputs) from an abstract formal model, and the generation of executable tests from these abstract test cases [1]. MBT is usually performed to automate and rationalize functional black-box testing activities. The abstract model, called test model, formalizes the behavioural aspects of the System Under Test (SUT) in the context of its environment and at a given level of abstraction. It thus captures the control and observation points, the expected dynamic behaviour, the data associated with the tests, and finally the initial state of the SUT. The test model must be precise and formal enough to enable unambiguous interpretations to automate the derivation of test cases. UML4MBT approach [2] enables automated functional test generation from a UML test model written with a subset of UML language [3] and OCL constraints [4]. These UML and OCL fragments are respectively called UML4MBT and OCL4MBT [5]. Basically, class diagrams define the points of control and observation of the SUT, instance diagrams define

P. Van Gorp, T. Ritter, and L.M. Rose (Eds.): ECMFA 2013, LNCS 7949, pp. 134–151, 2013.
© Springer-Verlag Berlin Heidelberg 2013

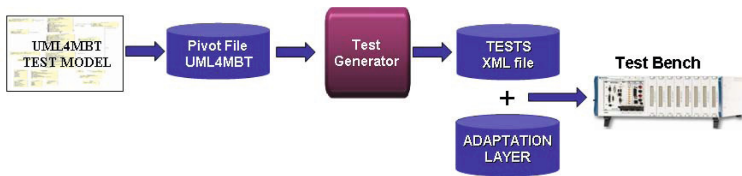


Fig. 1. UML4MBT existing toolchain

the initial state of the SUT and give the set of the test data, while Statemachines with OCL constraints define the expected behaviours in a formal way.

This MBT solution is implemented by the toolchain depicted in Fig. 1. It takes as input a test model specified by the UML4MBT/OCL4MBT language, which has a precise and unambiguous meaning. OCL4MBT expressions indeed provide the expected level of formalization necessary for model-based testing modeling. This precise meaning makes it possible to simulate the execution of the models and to automatically generate test cases. Such a test case takes the form of an abstract sequence (abstract because it is defined at the abstraction level of the test model) of the high-level actions modeled in the test model. These generated test cases contain the stimuli to be executed on the SUT, but also the expected results, obtained by resolving the associated OCL constraints. Finally, the test cases are concretized into executable scripts to be automatically executed on the targeted testing platform.

Since there is usually an infinite number of possible test cases that can be generated from a test model, some test selection criteria have to be applied to select a subset of appropriate test cases regarding the global purpose of the test campaign, and/or to ensure a given coverage of the system behaviours. Test selection criteria are usually based either on control-flow coverage [6] (such as all-states, all-transitions, all-k-paths, etc.), or data-flow coverage [7] (such as All-Defs, All-Uses, All-DU-Paths, etc.). Moreover, condition coverage criteria [8] (such as CC, DC, D/CC, MC/DC, etc.) may additionally be applied to enforce the structural coverage of the decisions of the test model. The test coverage strategy applied by the UML4MBT approach relies both on control-flow and condition coverage criteria: UML4MBT applies *All transitions* coverage, which ensures to cover each transition of the UML4MBT Statemachines, and Decision/Condition Coverage (D/CC) criterion, which ensures the coverage of all the conditions and all the decisions of the UML4OCL annotations.

In this paper, we propose to extend the UML4MBT solution to specifically address embedded mechatronic system domain. We thus propose to adapt this existing approach by taking *Systems Modeling Language* [9] (SysML) as input test models, and by introducing a dedicated coverage criterion, called `ComCover`, to select relevant test cases from such models. A fully automated toolchain, supporting this MBT process, is also introduced and experimentation results are provided. The paper is organized as follows. Section 2 introduces the motivation and the context of this research. Section 3 defines the subset of SysML notation supported to express the test model. Section 4 and 5 respectively describe and formalize the original `ComCover` criterion dedicated to such test models. Section 6

briefly presents the toolchain supporting this approach and discusses case study results. Section 7 finally concludes the paper.

2 Related Works

Mechatronic systems refer to systems that combine software, electronical systems and additional mechanical parts to perform a dedicated function. In this context, embedded softwares define a part of a larger system or product. Then, we can deduce that the embedded system exists because the larger system needs it. That is why testing an embedded system, without considering the system containing it, is often not efficient. In order to completely analyze and validate this kind of system, it appears necessary to take into account all parts which influences it.

Since 1990, the well-know simulation program PELOPS[1] is developed on the idea that, to specify a vehicle embedded systems in order to analyze it, it is necessary to represent three specific parts: the vehicle, the driver and the environment [10]. This realistic kind of modeling can then be validated by simulation: theoretical results calculated using such model framework are indeed compared with concrete results given by the physical corresponding system [11]. These three parts have to contain all elements that can influence the behaviour of the embedded system. This framework is nowadays still used in several works, and defines the preamble of the work presented in this paper. However, taking into account each environment part usually introduces combinatorial explosion problems, especially when each system part is tested independently. Moreover, in order to detect undesired behaviour, it is necessary to study interaction network between all the system components [12]. Consequently, our approach does not consist in verifying properties by proving local properties of each component, but in considering the global system components by focusing on their interactions.

A major challenge of such approaches concerns the heterogeneity of the different components and technology domains. To address this issue, it is needed to make uniform the representation of components at a given and adequate abstract level. [13] shows that using UML/SysML based models is an efficient way for automation engineering to handle the complexity of embedded systems. In this way, to avoid combinatorial problems, it is necessary to capture in the context model only the information required to simulate the behaviour of the system with regards to the evolution of its environment.

Several testing approaches for embedded systems are based on a model of the SUT environment. Typically, as proposed in [14], a test case is defined as a sequence of stimuli that are sent from the environment to the embedded system under test. In order to generate test cases, the authors apply Adaptive Random Testing and Search-Based Testing. These techniques allow to reduce combinatorial explosion during calculation, but gives poor information about the coverage of the test model and make difficult to assess a certain coverage.

Concerning the choice of the modelling language, two main strategies have been explored. As shown in [15], the first approach promotes the use of formal

[1] See `http://www.pelops.de/UK/index.html`

and mathematical languages such as Petri nets and VHDL-AMS codes. This kind of languages is powerful to specify mathematical and physical expressions, but they are not easy to acquire and does not often correspond to software engineer knowledge. Indeed, to be admitted by the software engineers, such approaches need to use a language widely used by software engineers. This point of view notably motivates the second approach that considers less formal, and often graphical, models. Moreover, in practice, starting the analysis of complex system using too concrete models is not convenient: it is usually necessary to begin the study using a more abstract notation in order to master the complexity of such systems. In this way, in [16], the authors propose to develop a domain model with UML class Diagram to represent the global structure of the environment (relationships, properties and constraints). Several behavioral models for each environment parts are also designed using UML Statemachines to model the dynamic part of the system. Other UML-based approaches use sequence diagrams, as [17], to model behavioural aspects or to represent test classification trees. In [18], the authors propose to use SysML to initially specify the system in a graphical manner. This language, where the object-oriented features are not visible, makes it possible to capture the mechatronic aspects of the SUT, and ease the interaction between different teams of multi-domain engineers.

In this paper, we also propose to use the Unified Modeling Language paradigm. This notation gives the advantage to be widely supported in terms of tools and training material. More precisely, we adopt SysML as specification language. Even if SysML is a recent modeling language, it is indeed on the rise in embedded system domain and some studies already use it to develop new industrial validation approaches (e.g. Model Checking and testing of on-board space applications [19]). Moreover, SysML, being defined as an OMG standard profile of UML, makes it possible to reuse existing testing approaches and tooling based on UML test models. In this way, it allows to adapt the existing UML4MBT approaches by focusing on the specific needs of testing mechatronic systems.

3 SysML4MBT Modeling

This section describes the subset of SysML notation supported to express the test model. This description is based on a simplified version of a realistic case study that will be used in the next sections to illustrate our approach.

3.1 Emergency-Stop Case Study

The emergency-stop case study describes a train emergency-stop system. This example will be used in the next sections to illustrate the proposed MBT approach. This system is defined by the following functionalities and rules:

- The train can either be stationing or moving on rail track.
- It is possible to set off the $emergency - stop$ by pulling the button $button1$.
- It is possible to set off the $emergency - stop$ by pushing the button $button2$.
- When one of these two buttons is activated, a signal is sent to the emergency-stop manager system, which automatically stops the train and sets off the alarm if the train is moving, or only advises the driver if the train is stopped.

To model this system, it is necessary to represent mechanical parts (buttons for example) and communications between subsystems (buttons and emergency-stop manager system) using signal. Mechatronic systems are indeed typically composed of some logical and some physical parts that communicate using mechanical or physical signals. But UML4MBT is not adapted to cover such aspects: a UML4MBT class diagram represents a logical entity of the system, and not at all a physical system; classes contain operations and attributes but signals are not allowed; only one Statemachine annotated by OCL expressions is allowed; neither parallel structures (parallel states, fork and join states) nor historic states are supported in the Statemachine; etc. That is why we decided to extend the expressiveness of UML4MBT to model parallel Statemachines and structures, and communication network between the subsystems of the SUT by taking into account the communication ports and links. Moreover, we also decided to use a subset of the SysML profile notation to capture the semantics of such embedded systems. It should be noted that the representation of time constraints, which is an other major aspect of embedded systems, will not be considered in the current approach, but will be studied in future work by using a dedicated UML extension such as MARTE profile [20].

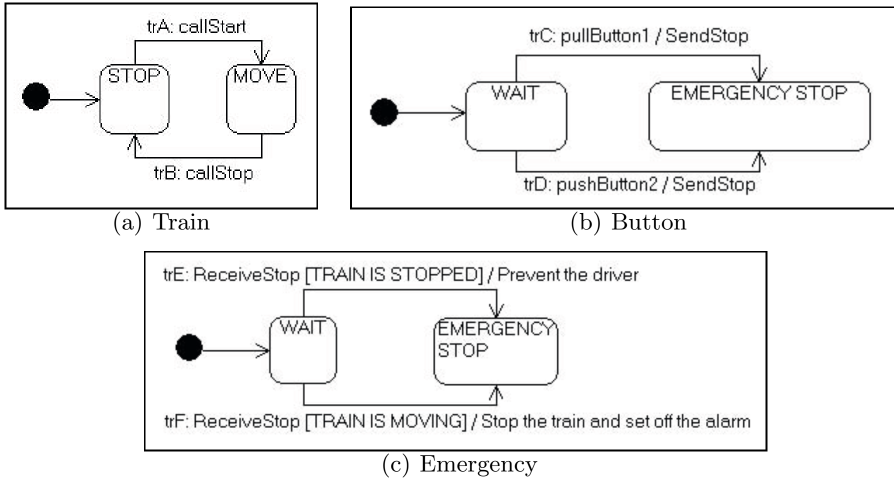
3.2 SysML4MBT Expressiveness

The test model, specified by SysML diagrams, defines the expected behavior of the SUT: it formalizes the control and observation points of the SUT, and its expected behaviour. However, SysML contains a large set of diagrams that defines a flexible notation and presents some freedom that can offer different semantical interpretations. For practical MBT, it is necessary to select a subset of SysML, and to clarify its semantics so that MBT tools can interpret the models in a unambiguous way. We thus define a precise subset of SysML for test generation purpose called SysML4MBT. A SysML4MBT model is composed of the following entities:

- A Block Definition Diagram (BDD) represents the static view of the system. It is defined as a stereotype of the UML class diagram. It can contain blocks, associations, compositions, signals, flow specifications and enumerations.
- An Internal Block Diagram (IBD) represents the internal view of the system, providing interconnections between its physical parts. The SysML IBD is defined as a stereotype of the UML Composite Structure Diagram.
- One or more Statemachines specify the dynamical view of the system. SysML Statemachines are directly inherited from UML Statemachines. Additionally to UML4MBT, SysML4MBT enables parallel structures (parallel states, fork/join states and multiple Statemachines) and historic states.
- In order to represent in a formal way the dynamical aspect of the system, OCL4MBT constraints are used to define the pre and post-conditions of the operations, and the guards and effects of the transitions in the Statemachine. The circumflex operator, which represents a sent signal in OCL, has been added to the OCL4MBT initial expressiveness.

3.3 SysML4MBT Modeling

To model the emergency-stop system, the train can be divided into three parts: the first part defines the general state of the train, the second one defines the button system, and the third one specifies the emergency-stop manager. It should be noted that, due to their simplicity and to simplify the presentation, only Statemachines are presented and depicted in Fig. 2 (BDD and IBD are not shown in this paper), and actions are all abstracted in the Statemachines. Finally, each transition is identified by a label `trX` to ease the explanation understanding.



(a) Train (b) Button

(c) Emergency

Fig. 2. Statemachines of the emergency-stop case study

The Statemachine defining the behaviour of the train, contains two states: $STOP$ (the train is stationing) and $MOVE$ (the train is moving). The expression *callStart* (resp. *callStop*) represents a call of *start* (resp. *stop*) operation that is a request to move (resp. stop) the train. At the initial state, the button system is waiting for an activation of one of the buttons. When one of them is activated (action $pullButton1$ if the button1 is pulled, and action $pushButton2$ if the button2 is pushed), a signal is sent to the emergency manager system. The sending of this signal is modeled by the action *SendStop*. Finally, the emergency-stop manager is initially positioned in the state $WAIT$. When it receives the emergency-stop signal sent by the button system (transition triggered by *ReceiveStop*), the train will stop and the alarm will be set off if the train is moving (guard of the transition); else, it only prevents the driver. This example will be used in the rest of the paper to illustrate the SysML4MBT testing approach.

4 Test Model Coverage Strategies

Some well-known criteria are usually used in Model-Based Testing techniques. A hierarchy of structural coverage criteria is notably defined in [21] as depicted

in Fig. 3(a). Criteria in the box are control-flow criteria and the others are data-flow criteria, while the (inheritance) arrows define that if the criterion at the start of the arrow is covered, the criterion pointed by this arrow is also covered.

The criterion *All states* consists in the coverage of each state of the model, while *All transitions* ensures that each transition is covered. It means that for each state (resp. transition), at least one test case executes it (if it is feasible). The criterion *All DU* (shortcut for *All Definition/Use*) deals with the coverage of each couple of definition (update) and use (reading) of each variable. It means that each time a variable is modified in the model, for each time it is read, a test, executing the definition before executing the use of the variable (without executing an other definition meantime) has to be generated. The criterion *All DU* is defined as an extension of the *All transitions* criterion: it ensures the coverage of all transitions and all definition/use pairs. The criterion $All\ DU-paths$ suggests the same level of coverage as *All DU*, but apply the approach to cover, for each variable, all the possible paths linking a definition and a use. Finally, the most constrained criterion of this hierarchy is the criterion *All paths*: it guarantees the coverage of all the possible paths in the system. The $All\ DU-paths$ and *All paths* criteria are infeasible in practice due to combinatorial explosion of reachable states, and can be usually applied only on very small models since it generates a large number (potentially infinite) of test cases.

4.1 Strategy Implemented within UML4MBT

The test coverage strategy implemented within UML4MBT relies on control-flow and condition coverage criteria. UML4MBT applies *All transitions* that ensures to cover each transition, and also implements Decision/Condition Coverage criterion (D/CC) for each decision branch of the model. The D/CC criterion deals with the coverage of all the conditions and all the decision of the model. It means that for each effect of each transition, the condition of decision structure and the decision itself have to be true and false in at least one test case.

These criteria do not take into account particularities of SysML models. Indeed, a major issue of SysML4MBT models in comparison with UML4MBT models concern the representation of communication links and exchanges (send and receive of signals) between components of the system. The next subsection underlines this lack of the UML4MBT approach on the emergency-stop example.

4.2 Illustration of the UML4MBT Strategy

The three Statemachines of the case study model contain six transitions. Each one contains only one behaviour without condition. Then, the application of the $All\ Transitions$ criterion is sufficient and the D/CC criterion is of no interest in this case. By performing UML4MBT approach to this model, the test cases represented in Table 1 are generated (in the representation of test cases, elements in parenthesis represent automatically fired transitions, while elements into brackets gives the name of the corresponding transition).

Table 1. Tests generated using UML4MBT strategy

Targets	Id	Tests
Train state Statemachine		
trA Start the train.	S1	$callStart[trA]$
trB Stop the train.	S2	$callStart[trA]$ $\rightarrow callStop[trB]$
Button system Statemachine		
trC Pull the button 1.	S3	$pullButton1[trC]$ $\rightarrow (ReceiveStop[trE])$
trD Push the button 2.	S4	$pushButton2[trD]$ $\rightarrow (ReceiveStop[trE])$
Emergency manager Statemachine		
trE Emergency stop called (train already stopped).		Already covered by S3 and S4.
trF Emergency stop called (train moving).	S5	$callStart[trA]$ $\rightarrow pullButton1[trC]$ $\rightarrow (ReceiveStop[trF])$

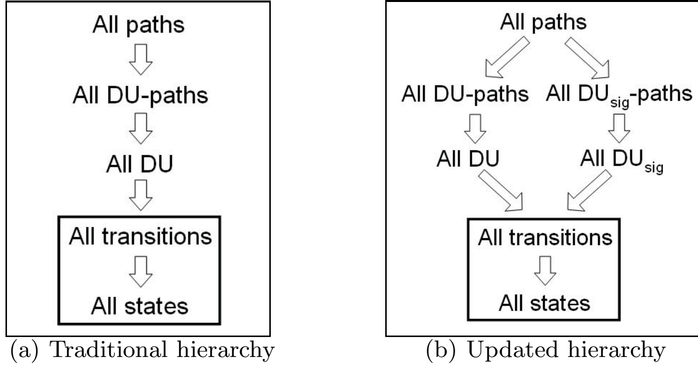
Since the sequence S1 is included in S2, S1 is not required. Thus, to satisfy the coverage criterion *All Transitions*, the four test cases represented by the sequences S2, S3, S4 and S5 are generated by the UML4MBT approach.

This example shows that there is a deficiency on the case study coverage because the scenario, consisting to push the button2 when the train is moving, is not required to satisfy the criterion. In critical system context, it appears to be necessary to test such case. Then, to avoid this lack, a more precise strategy should be applied: for this purpose, a dedicated data-flow test selection strategy, called `ComCover`, has been defined to cover all the configurations of signal exchange. The next subsection introduces this dedicated criterion and defines it with regards to the previously presented criteria.

4.3 ComCover Strategy

Within SysML4MBT test generation strategy, we are interested in the coverage of each signal received from each signal sending. In this way, we propose to adapt the *DU* approach, which concerns variables of the system, to address the exchange of signals in order to create a sensible test metric for reactive parallel Statemachines. This original criterion is called *All* DU_{sig}, and the corresponding strategy to select test cases, that satisfy this criterion, is called `ComCover`.

The *All* DU_{sig} criterion, based on *All DU*, deals with the coverage of send and receive events. The criterion *All* DU_{sig} guarantees the coverage of the succession of the sending event and the receive event: For each transition pair that is synchronized on the same event (one sender and one receiver), a test is required to show that the sender triggers the receiver. In analogy with the *All DU* criterion, the *All* DU_{sig} criterion can be seen as an extension of the *All transitions*



(a) Traditional hierarchy (b) Updated hierarchy

Fig. 3. Hierarchy of test coverage criteria

criterion. Thus, the $All\ DU_{sig}$ criterion ensures the coverage of all transitions and all send/receive couples. Finally, we also define the $All\ DU_{sig} - paths$ criterion, which guarantees, for each send/receive couple, the coverage of all possible paths containing them. The criteria hierarchy is updated as shown in Fig. 3.

The strategy, which consists in generating test cases in order to guarantee the $All\ DU_{sig}$ criterion, is called `ComCover`. The use of $All\ DU_{sig} - paths$ as selection criterion has not been implemented, and not be experimented, due to scalability issues. Indeed, $All\ DU - paths$ criteria is known to be infeasible in practice since it results in an infinite number of tests when the Statemachine contains loops [22].

The `ComCover` strategy is thus based on communications between parts: its purpose is to extract all send/receive couples of SysML4MBT Statemachines and to cover them by at least one test case that fires the concerned signal receiving behaviour after having fired its sending. Concretely, each behaviour $BhvA$ (of a transition in a Statemachine) that sends a signal to a specific port, and each behaviour $BhvB$ that can receive the signal sent by $BhvA$, are extracted from the model. Each couple $BhvA/BhvB$ then constitutes a test target to be covered by at least one test case to ensure $All\ DU_{sig}$.

4.4 Illustration of the ComCover Strategy

This section presents the results of the `ComCover` strategy using the SysML4MBT emergency-stop example. This model contains two signal sendings: $SendStop$ on the transition trC and $SendStop$ on trD. The receive of the signal can activate two transitions: trE and trF. Then, four test targets can be derived: the couples C1 (by firing trC before trE), C2 (by firing trC before trF), C3 (by firing trD before trE)and C4 (by firing trD before trF). The test cases of Table 2 are then generated using a classical breadth-search algorithm to cover each couple.

In comparison with the results obtained using UML4MBT approach, S6 and S3 are equals, like S8 and S4, S7 and S5, and S2 and S10. On this simple

Table 2. Test cases generated using ComCover

Targets	Id	Tests
C1 (trC/trE) Pull the button 1 (train already stopped).	S6	$pullButton1[trC]$ $\rightarrow (ReceiveStop[trE])$
C2 (trC/trF) Pull the button 1 (train moving).	S7	$callStart[trA]$ $\rightarrow pullButton1[trC]$ $\rightarrow (ReceiveStop[trF])$
C3 (trD/trE) Push the button 2 (train already stopped).	S8	$pushButton2[trD]$ $\rightarrow (ReceiveStop[trE])$
C4 (trD/trF) Push the button 2 (train moving).	S9	$callStart[trA]$ $\rightarrow pushButton2[trD]$ $\rightarrow (ReceiveStop[trF])$
Complement to guarantee *All transitions*		
trB Stop the train.	S10	$callStart[trA]$ $\rightarrow callStop[trB]$

example, all tests generated by the strategy of the UML4MBT approach are also generated with the `ComCover` strategy. Besides, the sequence that was missing using UML4MBT approach (activation of the emergency-stop using the button2 when the train is moving), is generated by `ComCover` with the sequence S9.

5 Formalization

This section introduces the formalization of the criteria D/CC and $All\ DU_{sig}$ on the basis of SysML4MBT expressiveness.

5.1 Formalization of a SysML4MBT Model

In this subsection, we introduce the subset of SysML4MBT notation that is required to formalize the coverage criteria. All elements annotated with $*$ are not detailed here, but can be found in [23], where the SysML4MBT modeling notation is completely formalized. A SysML4MBT model is composed of a Block Definition Diagram (BDD), Internal Block Diagram (IBD) and one or more Statemachines (SM). Internal Block Diagram, not required to formalize criteria, will be ignored in the rest of this section. We adopt the same restrictions for BDD in which only blocks and signals are relevant to define the criteria.

Definition 1 (Model). *A SysML4MBT model can be defined by the 2-tuple $\langle BDD, SMS \rangle$, where BDD represents the Block Definition Diagram and SMS is a set of Statemachine Diagrams (SM).*

Definition 2 (BDD). *A BDD is defined by the 2-tuple $\langle SIGS, BLOCKS \rangle$, where SIGS is the set of all signals and BLOCKS is the set of all blocks.*

Definition 3 (Block). *A Block* $BLOCK$ *is defined by the 3-tuple* $\langle OPS^*, PROPS^*, PORTS^* \rangle$*, where* OPS *is a set of all operations,* $PROPS$ *the set of all properties, and* $PORTS$ *the set of all ports contained in the block.*

To directly access to block elements of a model M, we define the accessors $M.allProps$, $M.allOps$ and $M.allPorts$ that respectively represent the set of all properties, operations and ports of the model M. We can now formalize the SysML4MBT Statemachine and its transitions. A transition starts from a state and reaches an other (which can be the same), and can be guarded and triggered by an event. When this event appends, if the guard of the transition holds, the transition is fired and one of its behaviours is executed.

Definition 4 (SM). *A Statemachine is represented by a 2-tuple* $\langle STATES^*, TRANS \rangle$*, where* $STATES$ *denotes all states of the Statemachine Diagram, and* $TRANS$ *is a set of all transitions of the Statemachine Diagram.*

Definition 5 (Transition). *A transition is defined by* $\langle TRstart^*, TRend^*, TRtrig, TRguard^*, TRbhvs \rangle$ *where:*
- $TRstart$ *is the initial state of the transition.*
- $TRend$ *is the final state of the transition.*
- $TRtrig$ *corresponds to the trigger of the transition* $TRtrig \in ((BDD.SIGS \; * \; allPorts) \cup allOps)$.
- $TRguard$ *defines the guard of the transition.*
- $TRbhvs$ *contains all behaviours of the transition.*

The behaviours of the transition are defined by an effect and a guard, which is a boolean expression on states that must hold to execute the action. It is formalized in the following way.

Definition 6 (Behaviour). *A behaviour is defined by a 2-tuple* $\langle BHVdecision^*, BHVaction \rangle$*, where:*
- $BHVdecision$ *defines the guard.*
- $BHVaction$ *is the set of all effects that can be executed when the behaviour is activated. An effect takes the form of a signal sending on a specific port or an update of a property value:* $(BDD.SIGS \; * \; allPorts) \cup (allProps \; * \; newValue)$ *(*$newValue$ *represents the new value to be associated to the property).*

5.2 Formalization of a Test Case

Within test generation from SysML4MBT models, we define a test case as a trace (sequence) of steps (operation calls).

Definition 7 (Trace). $TRACES$ *defines the set of all possible traces of the SysML4MBT model. A trace tr, such that* $tr \in TRACES$*, contains an ordered set of steps* $\langle StepOP^*, StepBhv^*, AllBhvs^* \rangle$*, where:*
- $StepOP$ *defines the operation triggering the behaviour.*

- *StepBhv is the executed behaviour if the trigger holds.*
- *AllBhvs is an ordered set containing all the behaviours (including StepBhv) triggered by StepOP.*

The set of generated test cases $TESTS$ is thus a subset of $TRACES$ that contains all the traces selected by the test generation strategy: $TESTS \subseteq TRACES$. All the elements, needed to formalize the coverage criteria D/CC and *All* DU_{sig} have been introduced.

5.3 Formalization of the Criteria

Using the definitions introduced in the previous subsection, we firstly propose in Fig. 4, the formalization of the criterion *all transitions* applied on UML4MBT model, which is refined using SysML4MBT model by *All* DU_{sig} criterion.

$$\begin{array}{l}\forall\ trans.(trans \in \{t | \exists\ sm.(sm \in M.SMS\ \wedge \\ \qquad\qquad\qquad t \in sm.TRANS)\} \Rightarrow \\ \quad \exists\ bhvTest.(bhvTest \in \{b | \exists\ (step, t).(t \in TESTS\ \wedge\ step \in t\ \wedge \\ \qquad\qquad\qquad b \in step.AllBhvs)\}\ \wedge \\ \qquad\qquad bhvTest \in trans.TRbhvs))\end{array}$$

Fig. 4. Formalization of *All transitions* criterion

The *All* DU_{sig} criterion, applied to SysML4MBT model to improve its coverage regarding communication exchange, is defined in Fig. 5 (in this formalization, the formula $bhvSend\ <_{step.AllBhvs}\ bhvRec$ means that $bhvSend$ is before $bhvRec$ in the $step.AllBhvs$ ordered set). Informally, this formalization establishes that a test set satisfies this criterion if all pairs signal send/receive are covered by at least one test case. The criterion *All* $DU_{sig} - paths$ enforces this criterion by ensuring the coverage of all paths that can be used to provide the *All* DU_{sig} criterion (this criterion has not been experimented due to scalability issues and is thus not formalized in this paper).

$$\begin{array}{l}\forall\ (sig, port, bhvSend, trRec). \\ \quad ((sig \in M.BDD.SIGS\ \wedge\ port \in M.allPorts()\ \wedge \\ \quad bhvSend \in \{b | \exists\ (sm, t).(sm \in M.SMS\ \wedge \\ \qquad\qquad t \in sm.TRANS\ \wedge\ b \in t.TRbhvs)\}\ \wedge \\ \quad trRec \in \{t | \exists\ sm.(sm \in M.SMS\ \wedge \\ \qquad\qquad t \in sm.TRANS)\}\ \wedge \\ \quad \langle sig, port\rangle \in bhvSend.BHVaction\ \wedge \\ \quad trRec.TRtrig\ =\ \langle sig, port\rangle) \\ \quad \Rightarrow \exists\ (step, bhvRec). \\ \qquad (step \in \{s | \exists\ t.(t \in TESTS\ \wedge\ s \in t)\}\ \wedge \\ \qquad bhvSend \in step.AllBhvs\ \wedge \\ \qquad bhvRec \in step.AllBhvs\ \wedge \\ \qquad bhvRec \in trRec.TRbhvs\ \wedge \\ \qquad bhvSend\ <_{step.AllBhvs}\ bhvRec))\end{array}$$

Fig. 5. Formalization of *All* DU_{sig} criterion

Finally, we can use the formalization of a SysML4MBT model to formalize the D/CC criterion, which is applied in our approach to complete the previous data-flow strategies. This formalization is expressed in Fig. 6.

$$\forall\, bhv.(bhv \in \{b | \exists\, (sm, t).(sm \in M.SMS \wedge t \in sm.TRANS \wedge b \in t.TRbhvs)\} \Rightarrow \exists\, bhvTest.(bhvTest \in \{b | \exists\, (step, t).(t \in TESTS \wedge step \in t \wedge b \in step.AllBhvs)\} \wedge bhvTest = bhv))$$

Fig. 6. Formalization of D/CC criterion

6 Toolchain and Experimentation Results

The `ComCover` approach consists to automatically derive, from a SysML4MBT model, test cases that satisfy *All* DU_{sig}. Moreover, we decide to ensure the D/CC criterion to cover the conditional branches specified in the model. The toolchain implementing this approach from SysML4MBT models is an extension of the existing UML4MBT toolchain (Fig. 1 in the introduction of this paper), which derives test cases from UML4MBT model by computing both *All transitions* and D/CC test selection strategies.

The obtained toolchain, depicted in Fig. 7, translates the entities of the SysML4MBT model into an equivalent UML4MBT model, and allows to re-use the test generation algorithms initially developed for UML4MBT models.

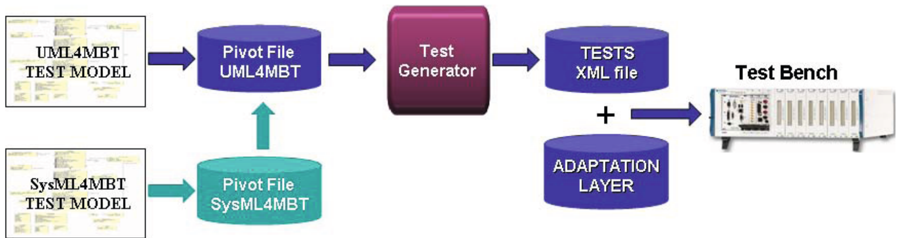


Fig. 7. SysML4MBT toolchain

The implementation of the `ComCover` approach is then performed during the translation of the SysML4MBT model into the corresponding UML4MBT model, as suggested in other Model-Based approaches such as [22]. More precisely, it consists to specialize the translation rules of SysML4MBT into UML4MBT model such that applying *All transitions* and D/CC strategies on UML4MBT resulting model implies the coverage of the initial SysML4MBT model by *All* DU_{sig} and D/CC criteria. The next sections give details about this implemented approach.

6.1 Model Transformation

SysML being a profile of UML, the majority of the rewriting rules to translate SysML4MBT into UML4MBT models can be automatically performed by deleting the stereotype layer of SysML (blocks become classes, block properties become class attributes, block operations become class operation...). For specific SysML entities, the following dedicated translation rules are defined:

- Each SysML4MBT signal is translated into a dedicated UML4MBT class.
- Each receive port is translated into a link between the class representing the block hosting the port and each class representing each signal that can be received on this port.
- The OCL operator circumflex is translated into an OCL expression that manipulates the link resulting from the translation of the receive ports
- Historic states are rewriting using a class attribute that simulates a memory state and related OCL constraints are added on transitions.
- parallel structures (fork/join, parallel states and multiple Statemachines) are translated into sequential structures by applying a synchronized product.

The SysML4MBT model is thus automatically translated into an equivalent UML4MBT model that can be used as input of the existing test generation tool. The rules to translate SysML4MBT models into an equivalent UML4MBT models are detailed and formalized in [23].

6.2 ComCover Implementation

To apply the `ComCover` strategy, specific transitions have to be introduced during the translation from SysML4MBT into UML4MBT model. These artificial behaviours concern each signal send/receive couple, which defines the goal of the *All* DU_{sig} coverage criterion: each pair signal send/receive of the SysML4MBT model is thus represented by one specific transition behaviour in the resulting UML4MBT model. Since UML4MBT applies a selection strategy based on the criteria *All transitions* and D/CC, we can ensure that each pair signal send/receive is covered by the generated test cases. The implementation of this dedicated translation requires three steps. Firstly, at each generated UML4MBT class that denotes a SysML signal, an attribute is added. This attribute is an integer initialized to 0. Secondly, OCL expression are added to all behaviours sending a signal to update this attribute with a specific number: in this way, from each receive behaviour, it is possible to know from which send behaviour the pending signal has been sent. Finally, a nested OCL conditional expression (each condition tests a particular value of the attribute) is added to each transition triggered by a signal receive. It artificially creates behaviours that are covered by the D/CC strategy. Covering these behaviours thus allows to ensure the *All* DU_{sig} coverage of the SysML4MBT model.

6.3 Case Study Results

The `ComCover` selection strategy has been evaluated with four case studies:

- *Lightings* deals with the front lightings system of a car. This system allows to independently light on and light off headlights and highlights of the car using a control lever. The SysML4MBT model only contains simple one-to-one communications: the test cases generated using `ComCover` implementation are thus equivalent to those obtained by applying *All transitions* strategy.
- *Lightings Extended* also concerns the study of a front lighting system of a car, but it considers the ignition subsystem and the control stick is replaced by a tactile panel. Thus, communications are more complex than *Lightings* case study. The SysML4MBT model looks like smaller but contains, in fact, more functionalities. For instance, in this case study, an historic state adds complexity in the model, and the model manages much more signals. That is why the use of `ComCover` becomes relevant and more tests are generated and improves the coverage of the SysML4MBT test model.
- *Wiper* specifies a wiper system of a car. Modeled functionalities are speed of drying up (low, high and intermittently) and windows cleaning with drying up. A lot of mechatronic parts being considered and communications being complex, a lot of relevant test cases are generated by the `ComCover` algorithm.
- *Steering* aims to examine behaviours of the steering column of a car, by observing reactions of the system contingent on road plots. In the SysML4MBT model, road characteristics are represented using blocks. Those blocks are linked to the steering column that defines the SUT. Test cases generated by `ComCover` are more relevant regarding mechatronic validation purpose, i.e. interactions between components and its environment.

Table 3 summarizes the global metrics of the experimentation results. From the second to the sixth line, the amount of the different entities specified in the SysML4MBT test model is given: Blocks, Connectors/Sending signal/Receive signal (c/s/r), parallel Statemachines (SM), States, Transitions. *States* (resp.

Table 3. Case study results

		Emergency Stop	Lightings	Lightings Extended	Wiper	Steering
SysML4MBTmodel	Blocks	4	6	4	15	9
	c/s/r	1/2/2	4/8/4	8/14/10	26/58/65	10/25/20
	SM	3	5	3	12	6
	States	(2,2,2)	(2,2,2,2,4)	(2,5,5)	(1,1,1,1,1,2, 17,10,2,2,2,2)	(2,4,3,6,3,4)
	Transitions	(3,3,3)	(3,3,3,3,9)	(2,8,8)	(2,3,2,4,1,3, 52,16,2,2,2,2)	(3,8,4,5,9,8)
All transitions		4	8	17	85	35
ComCover (new tests)		1	0	8	19	26
Total		5	8	25	104	61

Transitions) line details the number of states (resp. transitions) of each Statemachine of the model. The lines *All transitions* contain the number of tests generated using this strategy, while `ComCover` line introduces the number of new test cases produced by this coverage criterion. Finally, the line *Total* indicates how many test cases are generated by applying both strategies.

These case studies, presenting a growing complexity in terms of model expressiveness and behavioral aspects, show that `ComCover` can be successfully applied to embedded system domain. In the one hand, it ensures a better coverage of communication issues specified in the test model, as shown previously with the Emergency Stop case study. In the other hand, the number of new generated tests is manageable. In [24], we prove that, given c, s and r being respectively the number of Connectors, Sending signal and Receive signal of a SysML4MBT specification, the maximum number of new tests generated by `ComCover` is theoretically equal to $(s - c + 1) * (r - c + 1) + (2c - r - s + 1)$. But in practice, this maximum, which defines the worst case (all the signals can be sent to and received from all the connectors) is never reached.

While *Lightings*, *LightingsExtended* and *Wiper* case studies have been realized in an experimental way (the generated test cases have been executed and studied using only a simulated version of the concrete mechatronic system), *Steering* case study gave rise to a complete use of the toolchain (from the SysML4MBT test model to the execution of the generated test cases on a physical test bench. More concretely, the generated test cases have been executed to validate a concrete physical Steering Column against a Matlab/Simulink simulation model. The Matlab/Simulink simulation model has been used as reference to calculate the expected value on the concrete system. In this way, the generated test cases have been executed on the physical test bench and values calculated by the simulator were compared with the values observed on the Test Bench.

This realistic experimentation enables the detection of some errors both in the simulation model and in the concrete system configuration. More details about this end-to-end toolchain and the experimentation results on this case study have been respectively published in [25] and [26]. A short videotape, exemplifying it, is also available[2]. This experimentation, as well as the other case studies in a less realistic manner, enabled to validate our approach. They show its relevance for embedded mechatronic systems, which strongly rely on subsystem communication, by focusing the test objectives on signal exchanges.

7 Conclusion and Future Work

This paper proposes original coverage criteria (*All* DU_{sig} and *All* $DU_{sig} - paths$) to increase the model coverage, within MBT approach from *Systems Modeling Language*, to validate mechatronic systems. These criteria are based on a Def-Use approach focused on the communication features of the SysML test model. A dedicated test selection strategy, called `ComCover`, has been defined and implemented to automatically generate test cases covering the *All* DU_{sig} criterion.

[2] VETESS project web site - `http://lifc.univ-fcomte.fr/vetess/`

This strategy aims to improve an existing MBT process by considering communicating embedded systems modeled using SysML. However, this result is not restricted to this process and can be applied in all approaches that consider systems defined by material and logical subparts that communicates to each other. Finally, this automated toolchain has been experimented with industrial case studies, which allow to highlight the relevance of the `ComCover` strategy to generate test cases for communicating system. We are now investigating the use of real-time constraints to complete the SysML4MBT test model and improve the relevance of test cases for real-time systems. This model feature, major aspect of embedded system domain, will be addressed using UML MARTE profile.

References

1. Utting, M., Legeard, B.: Practical Model-Based Testing - A tools approach. Elsevier Science (2006) ISBN 0 12 372501 1
2. Bouquet, F., Grandpierre, C., Legeard, B., Peureux, F.: A test generation solution to automate software testing. In: Proceedings of the 3rd Int. Workshop on Automation of Software Test (AST 2008), Leipzig, Germany, pp. 45–48. ACM (May 2008)
3. Rumbaugh, J., Jacobson, I., Booch, G.: The Unified Modeling Language Reference Manual, 2nd edn. Addison-Wesley (2004) ISBN 0321245628
4. Warmer, J., Kleppe, A.: The Object Constraint Language: Precise Modeling with UML. Addison-Wesley (1996) ISBN 0 201 37940 6
5. Bouquet, F., Grandpierre, C., Legeard, B., Peureux, F., Vacelet, N., Utting, M.: A subset of precise UML for model-based testing. In: Proc. of the Int. Work. on Advances in Model Based Testing, London, UK, pp. 95–104. ACM Press (July 2007)
6. Offutt, A., Xiong, Y., Liu, S.: Criteria for generating specification-based tests. In: Proc. of the IEEE Int. Conf. on Engineering of Complex Computer Systems, Las Vegas, USA, pp. 119–131. IEEE Computer Society Press (October 1999)
7. Rapps, S., Weyuker, E.: Selecting Software Test Data Using Data Flow Information. Journal of IEEE Transaction on Software Engineering 11(4), 367–375 (1985)
8. Vilkomir, S., Bowen, J.: Formalization of software testing criteria using the Z notation. In: Proceedings of the 25th Int. Conf. on Computer Software and Applications (COMPSAC 2001), Chicago, USA. IEEE Computer Society Press (October 2001)
9. Friedenthal, S., Moore, A., Steiner, R.: A Practical Guide to SysML: The Systems Modeling Language. Morgan Kaufmann (2009) ISBN 9780123743794
10. Ehmanns, D., Hochstadter, A.: Driver-model of lane change maneuvers. In: 7th World Congress on Intelligent Transportation Systems (November 2000)
11. Glaser, S., Mammar, S., Sainte-Marie, J.: Lateral driving assistance using embedded driver-vehicle-road model. In: Conference on Engineering Systems Design and Analysis, Istanbul, Turkey, July 8-11 (2002)
12. Petin, J.F., Evrot, D., Morel, G., Lamy, P.: Combining SysML and formal models for safety requirements verification. Systems Engineering, 1–10 (2010)
13. Bonfé, M., Fantuzzi, C.: Object-oriented modeling of logic control systems for industrial applications. Journal on Automation Technology in Practice 2 (2005)
14. Iqbal, M., Arcuri, A., Briand, L.: Automated system testing of real-time embedded systems based on environment models. Technical Report 2011-19, Simula (2011)

15. Thacker, R., Myers, C., Jones, K., Little, S.: A new verification method for embedded systems. In: Proceedings of the 2009 IEEE Int. Conference on Computer design, ICCD 2009, Piscataway, NJ, USA, pp. 193–200. IEEE Press (2009)
16. Iqbal, M.Z., Arcuri, A., Briand, L.: Environment modeling with UML/MARTE to support black-box system testing for real-time embedded systems: Methodology and industrial case studies. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part I. LNCS, vol. 6394, pp. 286–300. Springer, Heidelberg (2010)
17. Mueller, W., Bol, A., Krupp, A., Lundkvist, O.: Generation of executable testbenches from natural language requirement specifications for embedded real-time systems. In: Hinchey, M., Kleinjohann, B., Kleinjohann, L., Lindsay, P.A., Rammig, F.J., Timmis, J., Wolf, M. (eds.) DIPES 2010. IFIP AICT, vol. 329, pp. 78–89. Springer, Heidelberg (2010)
18. Evrot, D., Pétin, J.F., Morel, G., Lamy, P.: Using SysML for identification and refinement of machinery safety properties. In: Proceedings of IFAC Workshop on Dependable Control of Discretes Systems, Cachan, France (June 2007)
19. Faria, J., Mahomad, S., Silva, N.: Tactical results from the application of model checking and test generation from UML/SysML model of on-board space applications. In: Proceedings of the Int. Conference on DAta Systems In Aerospace (DASIA 2009), Istanbul, Turkey, ESA Press (May 2009) ESA SP-669
20. OMG: UML Profile for MARTE. Revised draft 07-03-03L4.1, OMG (April 2007)
21. Frankl, P., Weyuker, E.: An Applicable Family of Data Flow Testing Criteria. The Journal of IEEE Transaction on Software Engineering 14(10), 1483–1498 (1988)
22. Weißleder, S.: Simulated satisfaction of coverage criteria on UML state machines. In: Proceedings of the 3rd Int. Conference on Software Testing, Verification and Validation (ICST 2010), Paris, France, pp. 117–126. IEEE Computer Society (April 2010)
23. Lasalle, J., Bouquet, F., Legeard, B., Peureux, F.: SysML to UML model transformation for test generation purpose. In: 3rd Int. Workshop on UML and Formal Methods (UML&FM 2010), Shanghai, China, pp. 1–8. ACM SIGSOFT (November 2010)
24. Lasalle, J.: Automatic Test Generation from SysML Models to Validate Embedded Systems. PhD thesis, DISC/FEMTO-ST - University of Franche-Comté (2012)
25. Lasalle, J., Peureux, F., Guillet, J.: Automatic test concretization to supply end-to-end MBT for automotive mechatronic systems. In: Proc. of the 1st Int. Work. on End-to-End Test Script Engineering, Toronto, Canada, pp. 16–23. ACM (2011)
26. Ambert, F., Bouquet, F., Lasalle, J., Legeard, B., Peureux, F.: Applying an MBT Toolchain to Automotive Embedded Systems: Case Study Reports. In: Proceedings of the 4th Int. Conference on Advances in System Testing and Validation Lifecycle (VALID 2012), Lisbon, Portugal, pp. 139–144 (November 2012)

A Network-Centric BPMN Model for Business Network Management

Daniel Ritter

SAP AG, Technology Development – Process and Network Integration,
Dietmar-Hopp-Allee 16, 69190 Walldorf, Germany
daniel.ritter@sap.com

Abstract. *Business Network Management* (BNM) helps enterprises managing their trading partner networks by making technical integration, business and social aspects visible within a common *Business Network* (BN) model that sets them into context to each other. This allows various roles, from the business specialist to the integration expert, to monitor, enrich and setup business processes by collaborating across its contexts.

In this paper we propose a common network model for BNM, which features inter-connected business and technical perspectives capturing the complete BN. Since the *Business Process Modeling Notation* (BPMN) is a well-established standard for describing business process and integration semantics, we define a network-centric BPMN model as graphical notation on UI and as basis for our BN by extending a subset of BPMN to cover both business and integration aspects. We present a novel approach on applying BPMN to BNM and discuss its application to real-world BNs.

Keywords: Business Network, Business Network Management, Network-centric BPMN.

1 Introduction

Enterprises are part of value chains consisting of business processes connecting intra- and inter-enterprise participants. We call the network that connects these participants with their technical, social and business relations Business Network (BN). Even though the BN is very important for enterprises, there are few - if any - people in the organisation who understand this network as the relevant data is hidden in heterogeneous enterprise system landscapes. To change that, Business Network Management (BNM) [9] allows enterprises to get insight into their technical, social and business relations. It identifies relevant data hidden within heterogeneous and distributed systems in complex enterprise landscapes to computationally link it into business process and (technical) integration networks [10]. In addition, BNM computes semantic correlation between entities of both perspectives.

For instance, Figure 1 shows participants in a sample business process network, conceptually showing linked data within a business perspective of a (cross-)

P. Van Gorp, T. Ritter, and L.M. Rose (Eds.): ECMFA 2013, LNCS 7949, pp. 152–164, 2013.
© Springer-Verlag Berlin Heidelberg 2013

enterprise partner network. The participants represent business artefacts within an enterprise, that are related to participants within the partner or customer network. The participants as well as their relationships are considered complex and contain the underlying business processes which specify, e.g., a business document or goods exchange between related participants.

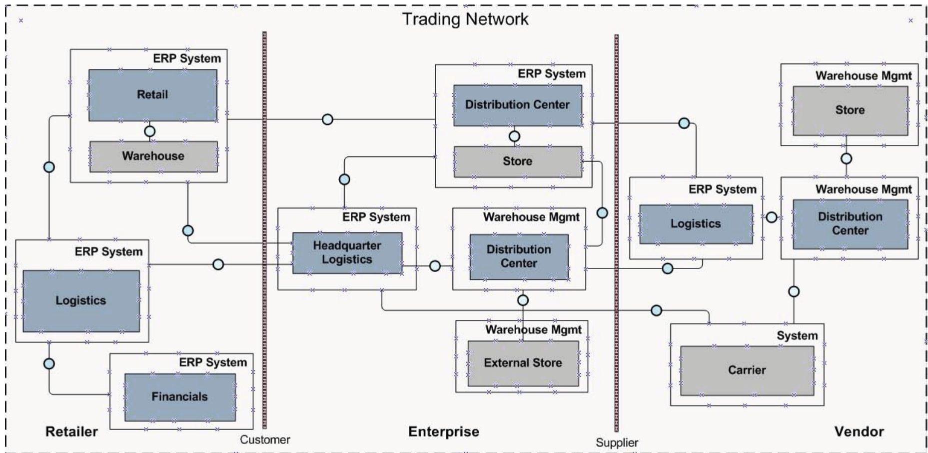


Fig. 1. Sample (cross-) enterprise BN showing participants and business document exchange as edges. Enterprises are characterised by their roles they play within a process.

For that, we present a novel and comprehensive approach on how to develop a model for network-centric business data. Based on a sound definition of BNs, we define the network-centric model suitable for BNM from basic to more complex structural, integration and business specific entities. The model is able to bridge from BNM to related areas like BPM. In this paper we describe the use of BPMN version 2.0[1] for our BN model and discuss specialisations needed for the specific domain. The approach leads to a network model which challenges BPMN in areas of nodes and edges, integration and business artefacts, semantic links, and mass network data management. The BN covers these areas and proposes new entities relevant for networks. For the evaluation of our approach we implemented a BNM prototype [11] and applied it to real-world enterprise landscapes.

We introduce BNs in Section 2 and discuss their design principles in Section 3. The BN is defined in Section 4. In Section 5 we show its application in a BNM system and share our experiences. We conclude with related work in Section 6, summarize and outline future work in Section 7.

2 Business Networks

The BN shown in Figure 1 is a conceptual view on how business-related participants exchange business documents and thus interrelate within and across

[1] http://www.omg.org/spec/BPMN/2.0/

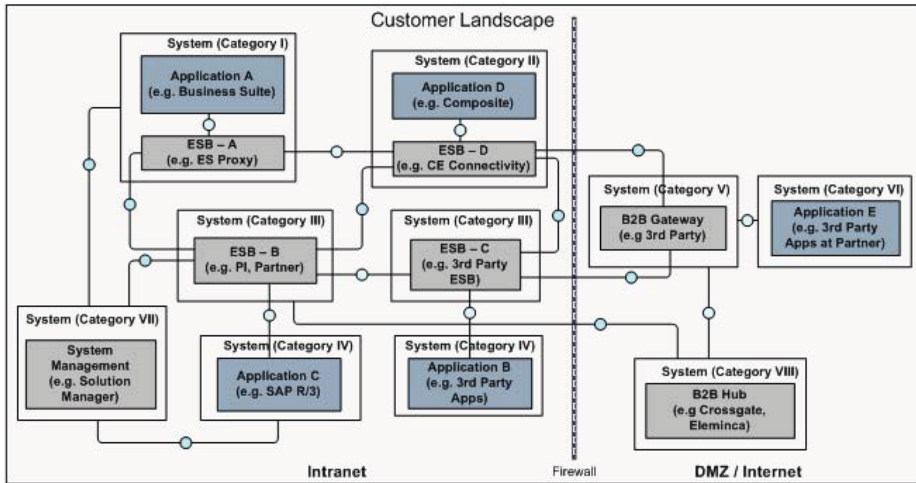


Fig. 2. Sample (technical) Integration Network showing logical systems as participants with embedded integration capabilities and standalone middleware

enterprises. The underlying business processes are actually implemented within the applications and integration capabilities of the enterprises denoting a more technical perspective, called integration network. Hence, the definition of the network for enterprises contains applications and middleware systems for internal business processes related to external processes interacting with business partners like suppliers, transport carriers, dealers. Figure 2 sketches a generalized view of such a network. When looking at an enterprise landscape, the systems within the integration network can be classified into different categories based on the integration content and the role they play. The classification provides insight into the capabilities and complexity of the network and allows to manage business processes, contextualized visualization and operations on the network. These categories span from applications with embedded connectivity or even integration capabilities, like proxies, enterprise services, composite applications or applications with service adaptation (Categories I+II), over standalone middleware instances with flexible pipeline processing (e.g., mapping, routing) and connectivity to legacy systems (Categories III+IV), to Business to Business (B2B) gateways for cross-enterprise document exchange (Categories V+VI) and system management solutions, which allow to operate these systems, their software and lifecycle (Category VII). The linked business processes define the roles applications play and their business document exchange. The knowledge about the business process as well as the integration domain, leads to a definition of a network where participants represent nodes and relationships between participants denote edges. Examples for participants are applications based on integration or business information. Relationships stand for integration or business documents as well as semantic relations between participants. Participants play roles within the network, which are defined by their relationships. Roles can be

retailer, mediator or contact person. Content of different kind, like social media models, process logs, is defined by participants. When the content is shared via relationships, it is protected by access control mechanisms, as references are by privacy control. The sample networks hint on a conceptual model that covers the definition of a BN and sketches its foundation.

3 Network Model Design Principles and Decisions

The BNs consist of different complementing perspectives like business process, integration, social/ orgranizational, whose real-world entities like hosts, business systems, applications and (semantic) relations shall be covered by the model (*REQ-1*: "one model" approach, different perspectives with domain-specific entities; *REQ-2*: semantic relations, cross-perspective contexts, e.g., from process to host). In our approach, the model serves as visual represenation (*REQ-3*: visualization, for (non-)technical persona/ roles) and standardized exchange format (*REQ-4*: computer readable, exchange format is well-standardized), e.g., for data exchange with related fields like BPM (see Section 6). The notation shall cover the requirements for defining entities, their relationships and properties representing the business network (*REQ-5*: network model). More precisely, the integration perspective shall cover the common integration patterns [7], called Enterprise Integration Pattern (EIP), as well as the integration artefacts discussed in Section 2 (*REQ-6*: cover integration domain), and the business process perspective shall model business aspects like business entities and the business document exchange (*REQ-7*: cover business domain).

Alternatives considered were notations like the *Service Component Architecture* (SCA) [16] and SoaML [17], which focus on the technical communication (e.g., within Service oriented Architectures), business related approaches like ARIS [1] or Supply-chain operations reference-model (SCOR) [18], and general modeling languages like UML [20]. In a nutshell, these approaches miss either real-world integration or business and social artefacts like services, contact person or business partner, thus contradicting the defined requirements. The SCA and SoaML notations support the technical side very well (*REQ-6*) and have standardized, computer-readable formats for data exchange, however not for BNM related areas (partially contradicts *REQ-4* and *REQ-1*). The modeling of semantic relationships for contextualization is only possible when extending the notations out of their domains (partially contradicts *REQ-2*), and support for the business domain does not exist (contradicts *REQ-7*). The latter requirement is fulfilled by more business related languages like ARIS or supply chains, which do not cover the integration aspects (contradicts *REQ-6*). With the general modeling approaches like UML, the domain specificities can be modeled (*REQ-6*, *REQ-7*). However for the same reasons they do not offer an business process near exchange format (partially contradicts *REQ-4*).

The Business Process Modeling Notation (BPMN) is a standard for defining, visualizing (*REQ-3*) and exchanging business procedures within (A2A) and

across (B2B) enterprises and is widely used within disciplines related to BNM like BPM [9] (*REQ-4*). The business aspects are well covered through the business process near notation (*REQ-7*) and the integration aspects, including the EIP and different categories of integration, have been proven to be expressable in BPMN [19] (*REQ-6*). With the BPMN conversation diagram, which was published in the standard in version 2.0, the different perspectives can be represented (*REQ-1*) and semantic relationships exisit between some of the model entities (*REQ-2*). A part of the contribution of this work is to define a network model suitable for BNM (*REQ-5*) and map the business and integration domain to the network. Hence this matches the defined requirements, we decided to base our BN on BPMN.

4 The Business Network Model

4.1 Basic Business Network Entities

The BN Model defines a subset of BPMN. Figure 3 shows the mapping to the BN's basic entities.

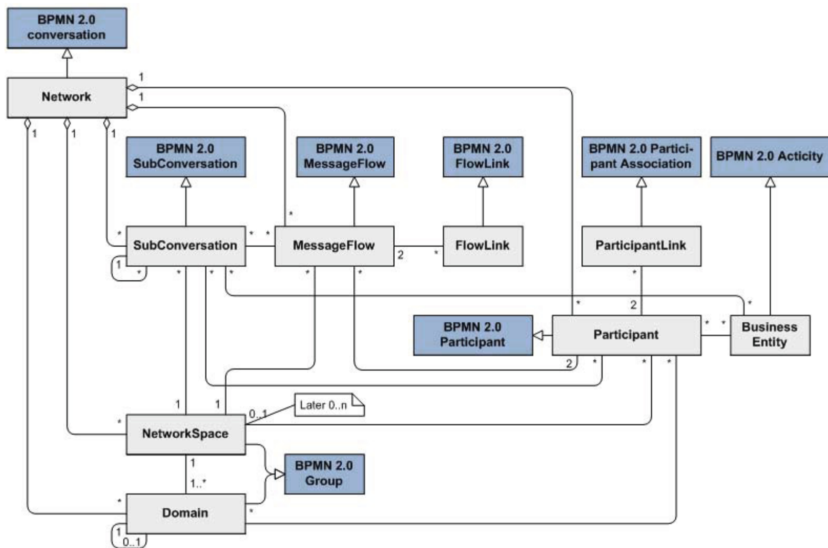
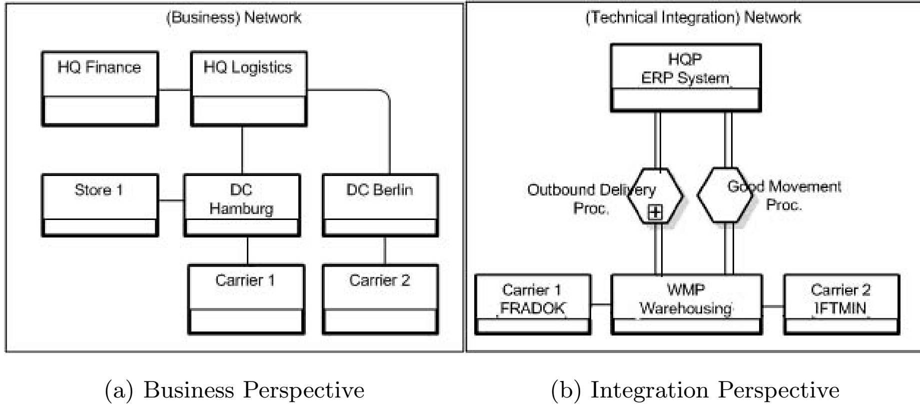


Fig. 3. Basic Business Network Model entities derived from BPMN

The *Network* itself is represented by the BPMN *Conversation Diagram*, as a special type of *Collaboration Diagram*, and defines a superset of the computed network and all manual extensions. BPMN *Pools*, referred to as *Participant*, and BPMN *Conversation* within the conversation diagram represent the process, document and control flow between business partners, applications and systems. For



(a) Business Perspective (b) Integration Perspective

Fig. 4. Sample BN representation in network-centric BPMN

instance, Figures 1 and 2 show a conceptualized BN representation of an outbound delivery process of an enterprise from different perspectives according to [13]. The BN model differentiates between a business perspective (Figure 4(a)) and an integration perspective (Figure 4(b)) to show the same network with different focus and for different roles. The central logistics department (*HQ Logistics*) interacts via application system *HQP* with the distribution centers (*DC Hamburg*/ *DC Berlin*) that use application system *WMP*. They both work with external transport agencies (*Carrier 1*/ *Carrier 2*) that communicate via interface standards *FRADOK*/ *IFTMIN*. Since participants of external carriers are typically not known, they are annotated with the employed interface standards. At the end the finance department (*HQ Finance*) generates an invoice via application system *HQP*. The interaction between business partners, applications and systems is depicted as top-level connections, e.g., between *HQ Logistics* and *DC Hamburg* in Figure 4(a), and it can expand to BPMN (sub-) conversations (e.g. *OutboundDeliveryProc.* and *GoodMovementProc.* in the corresponding technical perspective Figure 4(b)).

Nodes and edges are the basic entities of a network. A participant represents a node denoting a real-world entity, which communicates with other participants. The participant has two specializations: a *BusinessParticipant* (e.g., *HQ Logistics*) represents organizational units within the enterprise and external business partners while a *CommunicationParticipant* (e.g., *HQP*) has an IT perspective like system landscape, middleware configuration. To contextualise the two perspectives, a *ParticipantLink* is derived from the BPMN *ParticipantAssociation* (not shown). For example, *HQ Logistics* is related to *HQP* by a participant link of type "is-implemented-by".

A *Top-Level Connection* (single, straight line) is an extension to BPMN to visually represent the interaction/ edges between participants and group their (sub-) conversations and message flows. The Conversation links two or more participants and aggregates the *MessageFlows*. The MessageFlow represents the flow of messages between separate participants and is specialized as *Business-Flow* for business documents and *CommunicationFlow* for technical messages. Conversation and message flow can be grouped by *SubConversation*. This notion is based on the specification of BPMN, where B2B is supported by pools (i.e., participants as black box) and message flows. The BN is defined as inter-related participants by (sub-) conversations or message flows. Hence a network is defined as collection of all participants and conversations inferred from NM raw data. The network entity is used as entry point for visualizing and operating on the network. BPMN (sub-) conversations are the aggregation entities for inter-participant communication while message flows represent a single message exchange. A conversation can be visually expanded to several message flows.

4.2 Structural Elements of the Network

For the structural and data privacy network support the BPMN *Group* is used to define *Domain* and *NetworkSpace*. A domain is a subset of the network and is built to assign access rights to network entities with an access control list notion. Following the idea of directory services (e.g., ActiveDirectory or LDAP) domains can be hierarchically structured by associating multiple *CategoryValues* of the BPMN *Group* and particularly useful for multi-tenant contexts. The network space represents a subset of the network entities a user works with. The user can assign arbitrary BN entities to its network space limited by the domains the user is allowed to access. The user can propagate access rights to other users (according to the network space) in order to share it, while the domain-based access rights have higher priority.

For instance, Figure 5 shows a network, which is structured into three domains namely *Domain A* with a sub-*Domain B* and a disjunctive *Domain C*. A user with access rights to domains A and C has defined a network space from both to work with. A second user with access rights for domain B only, could not create this space nor could collaborate on it with the first user, even if the first user shares the space. Network spaces are comparable to work spaces in common development environments. They represent a subset of the network. Changes in the network space are also visible in the overall network view once they are "released" from a local copy.

4.3 Additional Integration Aspects

BPMN *Messages* are used to transfer data. They can be mapped to synchronous service calls (e.g., in a SOA domain) an event or any kind of asynchronous messages used for A2A/ B2B processes. For that, BN leverages the BPMN service extension point package to describe service interface (structure), operation (method) and endpoint (binding) configuration (see Figure 6). This allows an

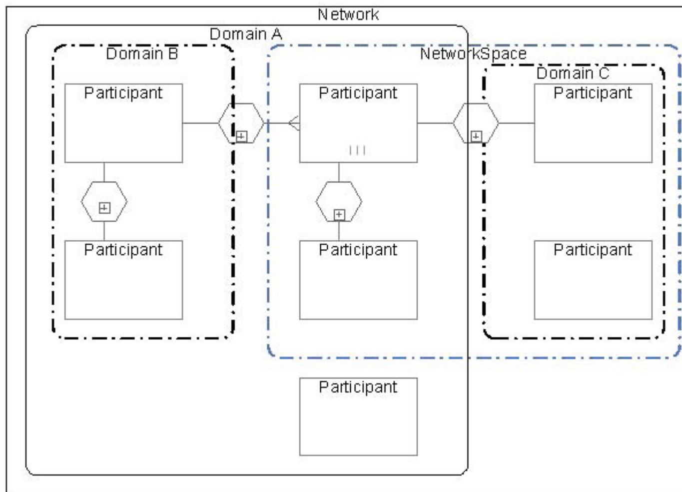


Fig. 5. Domain and NetworkSpace

integration of SCA-like artefacts [16] into BN (e.g., by mapping WSDL or SCDL to that triple). A participant can be associated to a *ServiceInterface* directly or via *ServiceOperation*, which is linked to the message and describes the action executed on the data. The *ServiceBinding* defines a technical communication channel configuration used for the message exchange.

4.4 Further Business Aspects

To contextualize the basic BN with business information, a business application related artefact called *BusinessTerm* is introduced (see Figure 7). A term is a scalar value or a tuple of values, which are relevant to a business user. During the discovery of the network, the terms are extracted from a concrete data entity (e.g., a business object or a message). In a nutshell a business term is an annotation to any part of a message like header, body, or meta-data. In addition, terms can have different values for different data representations, even if they refer to the same real-world object. That means, that business terms do not have canonical values, but map to specific values in different systems. For that, a term comprises locators to identify the value in different design time artefacts (e.g., data types, schemas, message types) and runtime artefacts (e.g., application objects, messages). Since a term can also refer to a tuple of values, thus referencing several locators. Moreover, when different systems use the same message type in different ways, the concrete locator can also be interface-dependent. However, it is assumed that this is an exceptional case. Each Locator refers to a value domain, which is the domain of all legal values that can be identified by the locator at runtime. The value domain for each locator is the domains for the values that a) are the basis for a value mapping between the between different

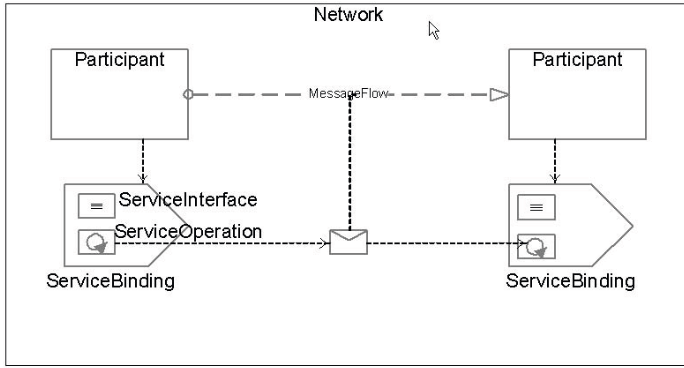


Fig. 6. Message, ServiceInterface, -Operation, and -Binding

representations of a business term and that b) can be returned by a BPMN *DataService*. In other words, the locators are the link between the semantically rich business terms and the data that is exchanged between systems. This link can be used in many ways, among them the translation of business related SLAs to operational SLAs that can then be monitored.

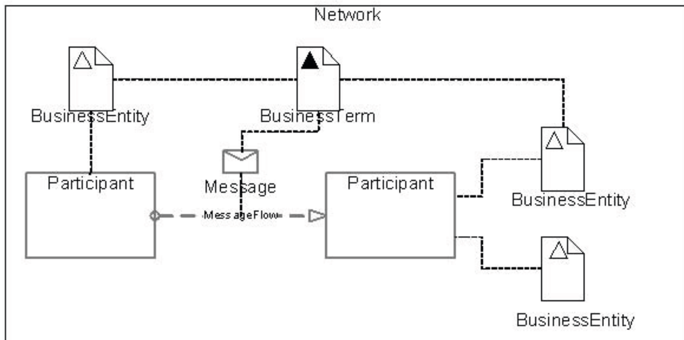


Fig. 7. BusinessTerm and BusinessEntity in the context of Participant, MessageFlow and Message

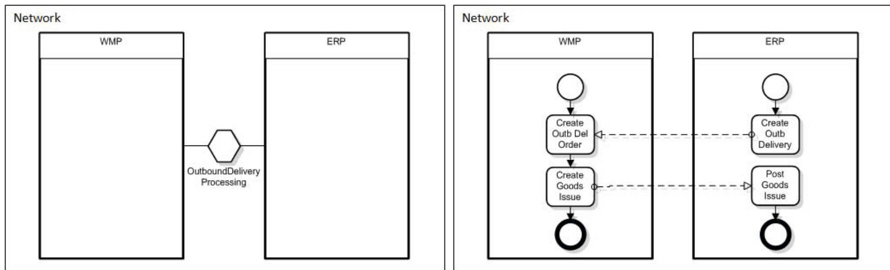
A *BusinessEntity* is a set of views on a collection of real-world objects (see Figure 7). Each view is based on the representation of the same object in a different system and should provide comparable (ideally the same) information about it. There is a 1-1 relationship between business entities and collections of these objects in a specific role (e.g., a customer entity and a partner entity can relate to collections that contain the same company), while the company could act in different roles. Technically, the views are a collection of business terms. The business entity data is accessed via locators associated to the corresponding

business terms. Business entities can also be related to each other. As terms can be shared between entities, a shared term is one way to indicate and potentially represent a relationship between business entities. Business SLAs that relate to a subset of a business term as part of a business entity can also be associated to that entity. Business entities can be associated to participants in a Network.

To describe the access to data managed by a participant, data services are introduced as enrichment entities, which can be associated to participants. They act as consumers of business terms and integration activities. For instance, a database table with application specific identifiers, needed for value mapping in a complex message flow, could be exposed and used by a middleware system.

4.5 Relationship to Business Process Models

The BN model approach defines an abstraction of the business and integration processes to show a BN instead. For that, a network model only contains BPMN participants and their connections (i.e., as conversations and message flows), where participants are used as black-box hiding internal details as depicted in Figure 8(a).



(a) System-centric process showing the outbound delivery conversation

(b) Extended outbound delivery processing conversation

Fig. 8. Process entities in BN shown for the outbound delivery processing conversation

By choosing BPMN as foundation for our model, the network model can be extend towards e.g. process steps, which denote the internal processing within a participant. Figure 8(b) shows the extended conversation for *OutboundDeliveryProcessing*, which now gives insight into the business process model for this conversation. The process model helps to better understand the business context of message flows and simplifies the communication between business and technical personas. Since the BPMN standard has no support for one participant participating in several collaborations, process steps are currently limited to single conversations and cannot be done simultaneously for multiple collaborations in our extended conversation.

5 Experiences with Real-World Business Networks

We developed a prototype BNM system that auto-discovers and graphically represents a integration network represented as BN model. The prototype works on top of an existing integration middleware (i.e., SAP Process Integration (SAP PI) [14]), together with system landscape data from SAP System Landscape Directory (SAP SLD) [15]. From that, it reads out standalone middleware instances with flexible pipline processing (e.g., mapping, routing and connectivity) for legacy systems, and operations data, which includes business systems, business components from SAP PI and enriches it with system information from SAP SLD. For the computation of message flows, it reads all communication channel related information like interfaces and combines it with logs of all executed message exchanges.

The novelty of this prototype is that the complete integration network is automatically discovered and does not require manual "modeling" of the network. Besides studies on internal enterprise network testbeds, we used the prototype to represent real-world enterprise landscapes like various internal and selected SAP customer landscapes, from which the integration networks were auto-discovered and we collected feedback both on discovery quality as well as on usefulness of such a solution. This real-world validation was very successful on both counts. Firstly, it proved that the auto-discovery is indeed feasible and resulted in highly reliable results. Secondly, such an integration network tool would be quite helpful in the everyday work of an integration architect, consultant or integration developer, since it gives an overview of the complete integration network which is currently not possible within the middleware integration tools. The prototype reduces the effort to document integration scenarios substantially, in particularly by a foreseen export of network details into PDF or office format. Furthermore helps to answer specific questions about the network which are currently still not (or only difficult) to achieve. For example, when combing configuration and runtime data it is possible to find connections that are not used any longer or were seldomly used in a given period of time. Hence, one of the customers was planning an upgrade project and with such a system a substantial migration time and effort will be saved.

The BN is expressed as network-centric BPMN (*REQ-1* entities that are made accessible to applications for visualisation (*REQ-3*) and further processing like network analytics, artefact and data migration, and network enrichment (*REQ-4*). The decision for BPMN is based on the expectation to benefit from using a widely adopted standard (*REQ-4*) (i.e., faster model design, lower learning curve, a standardized exchange format, etc). The prototypical implementation shows, that the network-centric BPMN allows to model (*REQ-6*, *REQ-7*) and contextualize integration and business process perspectives (*REQ-2*). Participants can be expanded to show activities and assign them to flows. This combines the domain with BPM, to e.g. start from the BN and drill-in to the activity level.

6 Related Work

Related work is conducted in the area of Process Mining (PM) [2], which is a relatively young research discipline that sits between computational intelligence and data mining. It has similar requirements to data discovery, conformance and enhancement. However, its approaches and goals are different. PM strives to derive BPM models from process logs. From that, models are automatically generated and checked. PM as well as BNM complement BPM by making it visible through automated discovery and in case of BNM to set the business processes in a broader context to each other.

Gaining insight into the network of physical and logical nodes within companies could be a future extension of BNM, but is not primarily relevant for visualizing and operating business networks. This domain is mainly addressed by the IT service management [8] and virtualization community [5].

The linked (web) data research [3,4] shares similar approaches and methodologies, which have so far neglected linked data within enterprises. However, our approach for a BN model could be enhanced to cover web data artefacts like social media or governmental entities.

7 Discussion and Future Work

In this paper we present a novel and comprehensive approach to use BPMN in a business network domain, namely the Business Network Management. We showed how a network model can be derived from BPMN (i.e. network-centric BPMN) and stated on our experience when using it in a BNM system prototype, which we applied to real-world customer landscapes.

Future work will be conducted for the BPMN especially in the areas of (1) re-fining the termininology (e.g., naming of conversation vs. sub-conversation), and adding views, (2) support for large networks containing hundreds or even thousands of participants (e.g., entry-points to the network, grouping, since currently BPMN groups are only specified for flow elements but not participants), and (3) nesting concepts (i.e., a concept of nesting for participants as for BPMN *Lanes* is missing). The application to new domains like social media and other linked (web) and public domain data will lead to the integration of new information models relevant for enterprises. Based on the static business process and integration networks computed by the BNM system, runtime data correlation for real-time monitoring of messages and business and technical exceptions have to be studied from a network model point of view.

References

1. ARIS Platform, http://www.softwareag.com/corporate/products/aris_platform/default.asp
2. van der Aalst, W.: Process Mining: Discovery, Conformance and Enhancement of Business Processes (2011)

3. Bizer, C., Heath, T., Berners-Lee, T.: Linked Data – The Story so Far. International Journal on Semantic Web and Information Systems 5(3), 1–22 (2009)
4. Bizer, C.: The Emerging Web of Linked Data. IEEE Intelligent Systems 24(5), 87–92 (2009)
5. Chowdhury, N.M.M.K., Boutaba, R.: Network virtualization: state of the art and research challenges. IEEE Communications Magazine (2009)
6. Heath, T., Bizer, C.: Linked Data – Evolving the Web into a Global Data Space. Synthesis Lectures on the Semantic Web: Theory and Technology. Morgan and Claypool Publishers (2011)
7. Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions. Addison-Wesley Longman, Amsterdam (2003)
8. O'Neill, P., et al.: Topic Overview – IT Service Management. Technical Report, Forrester Research (2006)
9. Ritter, D.: Towards Business Network Management. In: Confenis: 6th International Conference on Research and Practical Issues of Enterprise Information Systems, Ghent (2012)
10. Ritter, D.: A Logic Programming Approach to Integration Network Inference. In: The 26th Workshop on Logic Programming (WLP), Bonn (2012)
11. Ritter, D.: Towards Connected Enterprises: The Business Network System. In: 15th GI-Fachtagung Datenbanksysteme für Business, Technologie und Web (BTW) Workshops, Magdeburg (2013)
12. Rozinat, A., van der Aalst, W.: Conformance Checking of Processes Based on Monitoring Real Behavior. Information Systems 33(1), 64–95 (2008)
13. Outbound Delivery Processing, `http://help.sap.com/saphelp_crm40/helpdata/en/63/e48939728dd04abac5b86aa66002c2/content.htm`
14. SAP Process Integration, `http://www.sap.com/germany/plattform/netweaver/components/pi/index.epx` (2012)
15. Hengevoss, W., Linke, A.: SAP NetWeaver System Landscape Directory – Grundlagen und Praxis. SAP Press (2009)
16. Service Component Architecture (SCA), `http://www.osoa.org/display/Main/Service+Component+Architecture+Home`
17. SoaML, `http://www.omg.org/spec/SoaML/`
18. Supply-chain operations reference-model, `http://supply-chain.org/`
19. Stiehl, V.: Composite Application Systems – Systematisches Konstruieren von Verbundanwendungen unter Verwendung von BPMN. Dissertation, TU Darmstadt (2011)
20. Unified Modeling Language, `http://www.uml.org/`

Design Management: A Collaborative Design Solution

Maged Elaasar[1] and James Conallen[2]

[1] IBM, Rational Software
770 Palladium Dr., Kanata, ON. K2V 1C8, Canada
melaasar@ca.ibm.com
[2] IBM, Rational Software
30 S 17th St # 14 Philadelphia, PA 19103, USA
jconallen@us.ibm.com

Abstract. Design is more important than ever as software systems continue to increase in complexity, become more distributed, expose multiple interfaces and have more integration points. Design process has also become more complex, involving dispersed teams, third-party components, outsourcing providers and business partners. Nevertheless, design tools have not sufficiently been coping with these growing challenges. In this paper, we discuss design challenges and highlight features of design tools that should help address them. We also describe a new application; Rational Design Management (DM) developed to boost the quality of design and streamline the design process. DM enables a collaborative approach that broadens the understanding of design, improves design quality and shrinks design time. DM leverages semantic web technologies and implements the Open Services for Lifecycle Collaboration (OSLC) specification to deliver a linked data approach for managing design. Such an approach facilitates design extensibility, reuse and integration across the development lifecycle.

Keywords: Design, Architecture, Linked Data, Design Management, UML.

1 Introduction

Software design is a process of problem solving and planning for a software solution. Design has evolved from an ad-hoc, and sometimes overlooked phase, to an essential phase of any serious software development process [1]. Furthermore, the increasing complexity of today's systems has created a set of particular challenges that makes it hard for software engineers to meet the continuous customer demand for higher quality software [2]. These challenges have prompted software engineers to pay closer attention to the design process to better understand, apply, and promulgate well known design principles, processes, and professional practices to overcome these challenges. Some of the challenges include design complexity, requirements volatility, quality aspects (e.g., performance, usability, security), distributed teams, efficient allocation of resources, limited budgets, unreasonable schedules, fast-changing technology, use of third-party or open source components, and accurate transformation from software requirement to a software product.

P. Van Gorp, T. Ritter, and L.M. Rose (Eds.): ECMFA 2013, LNCS 7949, pp. 165–178, 2013.
© Springer-Verlag Berlin Heidelberg 2013

The outcome of a design process is a software design. The impact of this design on other related software development activities cannot be underestimated. For example, good design provides an abstraction that helps determine the best possible solution. It also reduces development time through better reuse of services and less rework. In addition, it reduces integration problems through communication of and agreement on interfaces and deployment topologies. It also reduces the cost of maintenance by providing developers, who might not have been involved in the initial construction, with a blueprint of the application [3].

Furthermore, designers today vary with respect to the formality and rigor of their designs [4]. Some use informal approaches that capture design with sketches, others use more standard modeling notations (e.g., UML [5]), yet others employ full model-driven architecture (MDA) [6], which uses transformations to automate downstream activities (e.g., code generation). Also, designers differ in the nature of their development process (agile, waterfall or hybrid). Regardless of the formality or nature of the design process, designers face more or less a similar set of challenges. A lot of these challenges are a result of design tools focusing on the designer and lacking support for team aspects of software design.

Unfortunately, this lack of support for team aspects leads software designers to work in silos and be disconnected from the rest of the team [4]. Examples of this disconnection include: a) stakeholders unable to find designs related to their work and unsure if they have the latest version of the design; b) designers spending time creating static design documents to send to stakeholders; c) too much time being spent on manual design reviews late in the project or iteration only to discover changes that result in rework; d) failure to use the best people as efficiently or as broadly, as they could be; e) manually building and maintaining of spreadsheets or documents to track the impacts of requirements and design changes. We believe that a collaborative approach to design can go a long way in eliminating such scenarios. Such a collaborative approach has in fact become a necessity rather than a luxury.

In this paper, we highlight what we believe to be the most pressing software design challenges today. We also outline some of the features that software design tools must have in order to cope with those challenges. Such features are identified based on our experience developing such tools over the years and on feedback from industry practitioners. We also describe our new application in this space, called Rational Design Management (DM) [7]. DM is a server-based application that is built on the IBM Jazz platform [8]. It provides a central repository for designs and capabilities that can be accessed from either a web client or rich clients. These capabilities allow project stakeholders to easily find, access and collaborate on designs. DM also implements the emerging Open Services for Lifecycle Collaboration (OSLC) [9] specification, which is based on the principles of linked data and semantic web. This provides DM with a solid architectural foundation that facilitates design reuse, extensibility and integration with other resources across the software development lifecycle.

The rest of this paper is organized as follows: section 2 discusses the main software design challenges today and outlines a list of features that design tools should provide to address them; an overview of DM's architecture and its main features is given in section 3; section 4 elaborates on some of DM's technical design decisions that allow

it to provide these features; a brief review of related tools is given in section 5; and finally section 6 provides the conclusions and highlights future works.

2 Software Design Challenges and Required Tool Features

Software designers are facing increasing business pressures to deliver faster, reduce costs and meet regulatory requirements. In trying to do so, they are confronted with challenges ([2] and [3]). We discuss here some challenges that we believe matter most to designers based on our experience helping them over the years. For each challenge, we highlight features that a modern software design tool should have to deal with it.

2.1 Expression Challenges

These challenges stem from the need to express software design using a technology that best meets the nature and goals of the design and broadens its understanding by team members. There is no doubt that the technological landscape for software design is continuously evolving and includes a myriad of formalisms; some are structured like UML and BPMN [10]; while others are less structured, like free-form sketches and rich text documents. Some of these formalisms may also need to be customized to fit the needs of a particular domains or projects. A modern software design tool should be able to not only support these formalisms, but also ensure they integrate well together to deliver synergetic value and reduce the learning curve for designers.

2.2 Access Challenges

These challenges come from the need for team members to easily find, access and collaborate on designs. When designs are hard to find or access, they tend to be built in silos, which increases the chance of discovering errors that result in rework and project cost and/or schedule overruns. They also tend to be more static and go out of sync with other related or derived resources. Moreover, they either do not get reused at all or wrong versions get used instead, resulting in a waste of design resources. They also become less comprehensible to other stakeholders like project managers, developers, testers and technical writers who have not been following them. A modern software design tool should make it easier to search for a design, access specific versions of a design, facilitate collaboration on a design and reuse of previous design components.

2.3 Lifecycle Integration Challenges

These challenges come from the need of design teams to have visibility into and coordinate activities on other interrelated lifecycle resources (e.g., requirements, work items, code, builds, test cases and test results). Faced with difficulties tracing to and assessing impact of change on those resources, designers tend to work in silos and defer coordination to the end, when rework is least possible and most expensive. This challenge occurs due to missing or poor integration between lifecycle resources that are likely to be managed by different applications. A modern software design tool

should allow designs to be linked to, and have rich data-integrations with, other lifecycle resources. It should also bring transparency to the development process by enabling designers to report on, trace to and analyze how their designs relate to those resources. This would allow them to answer questions like: are all requirements covered by the design? Has the design been completely implemented? What is the status of design testing? Are quality goals being met? What needs to be changed based on changes for the design?

2.4 Awareness Challenges

These challenges relate to the need of designers to stay aware of what is going on in the project, while they are busy getting their work done, because it might impact their work or they might be able to provide valuable input. Manually searching for this information (e.g., by sending emails or attending meetings) is time consuming and is usually abandoned after project schedules start to get tight, resulting in designers becoming unaware and slipping back into their silos. A modern software design tool should allow team members to stay on top of project activities, follow project progress, see project statistics and promptly get notified of other members' requests.

2.5 Configuration Management Challenges

These challenges result from having to cope with software design variability and change. Software design may vary to cater for the needs of different users or computing environments. It also continuously changes over time to provide different or new functionality or to address problems. This motivates designers to anticipate variability and plan for it. This is also why a modern software design tool should give designers the ability to setup different but related configurations for design to reflect the varied needs of the project (e.g., different design details for different product lines, markets or computing environments). It should also allow each configuration to evolve independently over time to produce newer versions and propagate its changes to dependent configurations.

2.6 Parallel Development Challenges

These challenges appear when members of a design team work in parallel on the same design. Design teams vary in the development process they follow. One process may involve each designer working independently on a branched stream of the design and periodically merging it into an integrated stream. Such process provides the least interruption to an individual designer but makes integration between members of the design team harder. Alternatively, a process can be more agile where several designers work directly on the same copy of design. Such process leads to occasional interruption (when design components need to concurrently be changed by several designers) but provides spontaneous integration. A modern software design tool should cater to both kinds of processes. Specifically, it should ease the integration of design branches with compare and merge features. It should also improve concurrent editing experience by allowing finer componentization of design. Furthermore, it should offer design review and approval features to ease collaboration.

2.7 Summary

Software design tools must evolve to address a number of important challenges. Among them, we believe the expression challenge is the most pressing as it makes it hard to use different but related formalisms together. The next pressing challenge is lifecycle integration as it prevents linking related lifecycle artifacts making it harder to understand the complete development solution. Parallel development challenges come next, since they make a design team less productive. They are followed by configuration challenges that complicate the management of design variability. Finally, access and awareness challenges make it harder to find and reuse designs and be informed of activities on the design project.

3 Design Management: Architecture and Features

In section 2, we outlined important challenges that face software designers today. We also highlighted features that should be provided in a modern software design tool to address them. In this section, we describe the architecture of a new software design application, called Rational Design Management (DM), which we developed to boost the productivity of the design process with collaborative features. We structure the description of DM's features based on the challenges we highlighted earlier.

3.1 Architectural Overview

DM is a server-based application that adds new capabilities (e.g., a central design repository, role-based access permissions, contextual collaboration, configuration management, parallel development and lifecycle integration) to traditional software design tools, specifically Rational Software Architect (RSA) [11] and Rational Rhapsody (RHP) [12]. DM exposes these capabilities as a set of RESTful web services that are used by these tools' rich clients, as well as a web client (Figure 1).

DM allows designs to be managed in one of two modes: externally managed or actively managed. In externally managed mode, a design is managed externally to DM by another file-based software configuration management (SCM) system and periodically (e.g., daily) gets published to DM to enable other collaborative features on it (e.g., reviews, comments and links). On the other hand, in actively managed mode, a design is managed directly and exclusively by DM. One advantage of externally managed mode is the ability to access a design offline, while in actively managed mode a live connection to DM is required. Another advantage is the ability to manage both design and non-design resources, while in actively managed mode only designs are managed by DM. On the other hand, an advantage of actively managed mode is dealing with one SCM system only (i.e., DM), versus dealing with two in externally managed mode. Another advantage is an enhanced design editing experience as DM automatically manages design componentization at a more granular level than is typically done in external SCM systems, reducing the possibility of collisions or lock-outs when multiple designers work collaborative on the design.

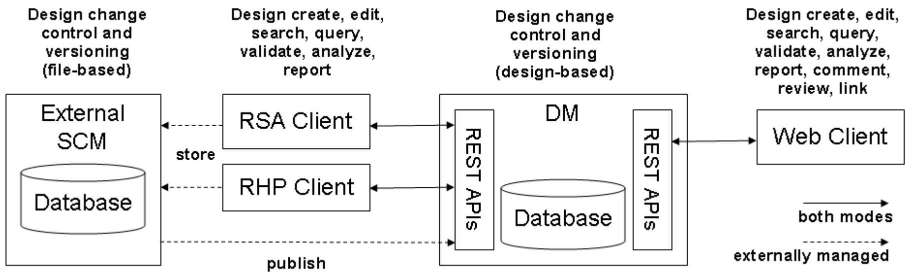


Fig. 1. Design Management Architecture

3.2 Expression Features

DM supports the expression of design in various domains (formalisms). DM supports a number of structured domains (e.g., UML, BPMN and SysML [13]). It also supports less structured domains (e.g., free-form sketches and rich text documents). A design domain is defined declaratively in DM with a set of OWL [14] ontologies specifying the abstract syntax of the formalism. Designs in a given domain are represented in RDF [15] using the ontologies of the domain. In addition to the predefined domains, DM supports user-defined domains that extend predefined domains, integrate them, or define completely new ones. DM also allows the customization of domain tooling by annotating the domain's OWL ontologies with DM-specific tooling annotations.

3.3 Access Features

DM stores designs in a central repository (a RDF store) and allows role-based access (with read/write permissions) to them using a web client or rich clients. This makes it easier for designers, and other stakeholders, to find, access and browse designs. DM allows designs to be searched using keywords or queried with SPARQL [16] queries, allowing stakeholders to easily find the information they look for. It also allows them to collaborate on designs by contributing to them directly or by reviewing them with threaded comments and mark-ups (Figure 2). DM also supports design configuration management (section 3.6), which allows stakeholders to choose to access a particular version (e.g., the latest) of the design.

Fig. 2. DM screenshot showing design comments and mark-ups on a design

3.4 Lifecycle Integration Features

DM is one of several applications in an application suite called Collaborative Lifecycle Management (CLM) [17]. Other applications in the suite manage different lifecycle resources (e.g., requirements, tests, work items and builds). DM supports integration with those applications using a specification called Open Services for Lifecycle Collaboration (OSLC) [9]. By implementing OSLC, DM allows designs to have links to other lifecycle resources. A link allows navigation to the linked resource. It also allows retrieving important information (e.g., last time modified) about the linked resource that the defining application chooses to expose. DM leverages those OSLC links to generate cross-lifecycle reports. DM and other CLM applications also support the periodic publishing of parts of their resources, including OSLC links, to a common index. This allows multi-level cross-lifecycle traceability analysis with queries to this index (Figure 3). This analysis can answer specific questions on the relationship between designs and other linked resources.

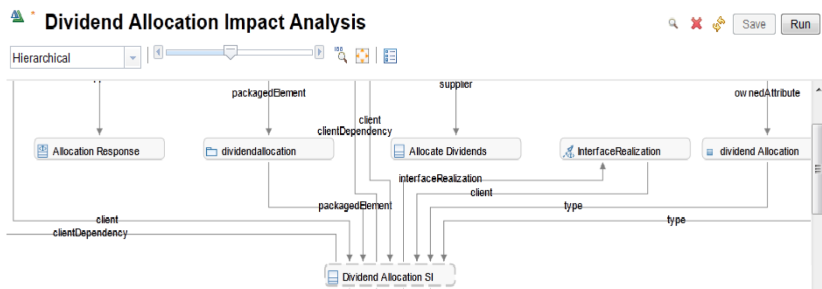


Fig. 3. DM screenshot showing impact of a change in Dividend Allocation

3.5 Awareness Features

DM provides a web-based dashboard (e.g., Figure 4) that integrates relevant project info into a single location. The dashboard includes widgets for showing collaboration details, lifecycle traceability links and design queries. For example, dashboard widgets can show users which design resources have the most comments over the past week. A dashboard is not just limited to designs; it can be a mashup that combines information from the entire lifecycle using OSLC links. It is also customizable for teams and individuals with widgets that provide news feeds, bookmarks, etc.

Reviews (2) My Reviews
361: testReview Apr 18, 2013
test Yesterday
349: rev Apr 17, 2013
test 2 days ago
Recent Changes (Design) (0)
No results available.
Recent Links (10) Past Week
27: JKE Design Model Yesterday
5: Process email requests
My Design Projects (6)
Design Management (/dm) (6) hide
Domains
JKE Banking (Design Management)
Page 1 of 3
Recent Comments (4) Past Week
27: JKE Design Model RE: alex
test Yesterday
27: JKE Design Model alex
alexh Yesterday
27: JKE Design Model RE: test
alexh Yesterday

Fig. 4. Web-based dashboard screenshot showing relevant project info

3.6 Configuration Management Features

DM provides its own configuration management features. Designs in DM are organized into project areas based on the product components they belong to (e.g., for a car product, there could be project areas for engine, radiator and steering). Designs can have multiple versions, all of which are stored in the same project area. However, a project area has a specific version active at a time. This version belongs to the active configuration, which is one of many related configurations in an n-dimensional configuration space. These dimensions represent product line variability parameters (e.g., for a car product, they can be model, trim and year), but the last dimension always represents time (e.g., for a car product, it can be the milestones within a year). Related project areas (e.g., those belonging to the same product) can be associated with the same configuration space allowing them to have synchronized versions.

A configuration in DM (e.g., Figure 5 shows a web-based configuration browser) can be one of two kinds: a workspace (e.g., ChildWS) or a snapshot (e.g., SomeSS). A workspace is a mutable configuration that allows designs to be changed. This is used for active work on designs. A snapshot, on the other hand, is an immutable configuration that has frozen at a point in time. This is used for releasing milestones. A new configuration can be branched off an ancestor configuration, from which it inherits its initial content. A new workspace can be branched off an ancestor snapshot, while a new snapshot can be branched off an ancestor workspace (it is not useful to branch a snapshot from another since it cannot change).

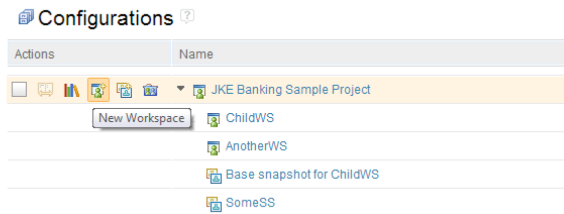


Fig. 5. Configuration browser screenshot showing configuration hierarichy

3.7 Parallel Development Features

DM allows two styles of parallel development on designs. The first style is typically used in a traditional development process where each designer has a private workspace that is branched off an integration workspace (e.g., Figure 6 shows a branch workspace of project JKE Banking being edited in RSA). The designer makes changes in this branch workspace, periodically rebases by accepting latest changes from the integration workspace, and when ready delivers the branch changes to the integration workspace. This rebase and delivery operations might involve resolving conflicts when the same design components have changed. DM eases conflict resolution with a design compare and merge feature. The other style of parallel development, supported by DM, suits a more agile development process, where more than one designer works on the same active workspace in the same time. This may

potentially lead to lock-outs if several designers happen to edit the same design components concurrently. To minimize this chance, DM componentizes (fragments) its designs at a more granular level. This level is configurable for each domain and is applied automatically by DM.

Furthermore, changes made by designers can be grouped into change sets (shown in Design Changes view in Figure 6). These change sets can be reverted or delivered. They can also be shared between designers for the purpose of collaboration, including commenting, reviewing and approving.

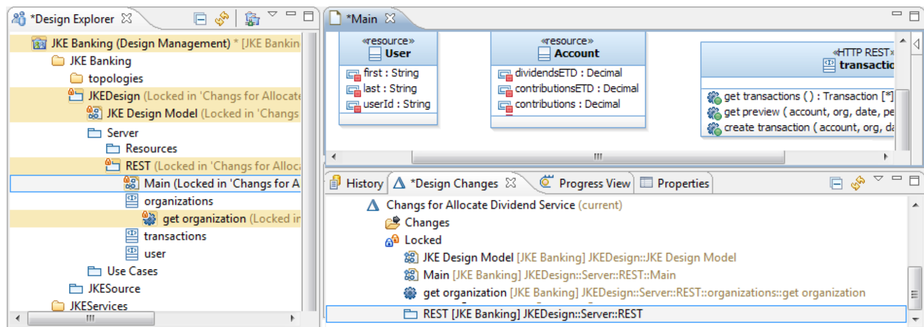


Fig. 6. Parallel development screenshot in RSA with DM capabilities

4 Design Management: Design Decisions

In section 3, we briefly overviewed the architecture of DM and described its main features that allow it to address the design challenges outlined in section 2. In this section, we highlight some of the technical design decisions of DM that allow it to provide these features. In particular, we choose to discuss three main design decisions: 1) representing designs using RDF, 2) defining design domains using OWL, and 3) linking designs with other lifecycle resources using OSLC.

4.1 Representing Designs Using RDF

Traditional software design tools, like RSA or RHP, use XSD [18] or MOF [19] based architecture to represent their designs. These architectures adopt the closed world assumption that states that what is not currently known to be true is false. This makes extending these designs, for example to add domain or project-specific extensions, quite hard. It also makes linking different designs, say a UML detailed design with a BPMN process design, a daunting task. On the other hand, the open world assumption states that the truth value of a statement is independent of whether it is known to be true by a single entity. This assumption is adopted by the semantic web [20] architecture, which represents resources in RDF as a set of RDF triples that are grouped into graphs. These graphs describe knowledge about resources, identified by their URIs, which can be extended by adding extra triples to those same graphs or to other ones.

DM chooses to represent its designs as RDF graphs. This allows designs to be easily extensible by adding extra triples. For example, one can add triples about UML notation to a UML design even though the UML ontology do not describe notation. Furthermore, this approach makes designs easily linkable to each other by adding triples representing links (cross-references). For example, one can make an activity in a UML design reference a corresponding activity in a BPMN design. Finally, a big advantage of using RDF is allowing a modular approach to design, where different aspects of a design can be specified in different RDF graphs. This also allows parallel development on designs with a reduced risk of conflicting edits. In addition, processing (e.g., querying and transforming) designs can be parallelized.

4.2 Defining Design Domains Using OWL

In the semantic web architecture, knowledge about a given domain is defined as a set of ontologies. An ontology specifies, for a given domain, concepts (classes), relationships between pairs of concepts (properties), and/or individuals typed by those concepts (instances). However, an ontology is not a schema language for RDF in the traditional closed world sense. Rather, it defines the semantics of RDF data in a way that allows a reasoner to deduce new derived information. For example, knowing that John is a friend-of Bop and Bop is a friend-of Jim, and knowing that friend-of is a transitive property; a reasoner can deduce that John is also a friend-of Jim. Moreover, there is no notion of RDF graph validation in the semantic web philosophy. Rather, a reasoner can evaluate whether a set of RDF graphs are consistent, meaning they contain no contradicting statements (either directly or through inference).

The semantic web architecture defines two ontology languages: RDFS and OWL (OWL has richer semantics). DM chooses to represent the abstract syntax of its standard and proprietary design domains in OWL. However, since those standard domains are originally defined in other languages like MOF (e.g., UML), UML profile (e.g., SysML) or XSD (e.g., BPMN), DM provides mappings from those languages to OWL. For example, a MOF class, a UML stereotype or a XSD composed type maps to an OWL class. Similarly, a MOF property, a UML property or an XSD attribute/element maps to an OWL property. DM also provides reverse mappings from OWL to those languages. These mappings are used when DM designs are manipulated by rich clients of supported design tools (i.e., RSA and RHP).

Although the full details of these forward and reverse mappings are beyond the scope of this paper, we would like to elaborate on the way UML stereotypes are mapped to OWL in DM. Both a UML stereotype and its base classes in UML map to OWL classes. However, when mapping a UML element with a stereotype applied to OWL, DM represents it differently from UML. Specifically, instead of a stereotype application referencing a UML element, DM leverages RDF's multi-classification feature (the ability of a resource to have several types) to make a UML element additionally classified by the stereotype class thus has access to its properties directly.

Furthermore, DM's web-based tooling is mostly driven declaratively by domain definitions. For example, a property sheet for a design resource would list all possible properties for the resource by querying the domain's ontologies for all properties

whose domain match the resource's types or one of their super types. However, DM allows customization of its web tooling through annotations to the domain's ontologies that get stored in separate graphs. For example, an annotation can override which widget to use when displaying the value of a given property in a property sheet. Such annotations are themselves defined by DM-specific tooling ontologies.

One other configuration that DM allows in a domain is design componentization. DM allows a domain to configure how to componentize a design by splitting it into different graphs. DM allows a design resource to be defined in one main graph but can be extended from other graphs. Also, DM distinguishes between a type of resource that is defined in its own graph and a type that is defined in another resource's graph. A domain can flag a class in an ontology as a graph class, which forces DM to define resources typed by this class in their own graphs. A domain can also specify for each graph class, which non-graph classes can also be defined within the same graph. For example, the UML domain in DM only flags packages, classifiers, attributes and operations as graph classes. This allows only resources of these types to have their own graphs. This automatic componentization configuration can be setup to optimize collaboration (by minimizing conflicts and concurrent edits) and linking (since only graph resources can be linked to using OSLC).

4.3 Linking Designs with Other Lifecycle Resources Using OSLC

As previously mentioned in section 3.4, DM is part of the CLM suite of collaborative lifecycle applications. Lifecycle resources (e.g., requirements, designs, tests, work items) that are managed by these applications are often interrelated. These interrelationships can be specified as links between these resources. Such links allow easy navigation between the linked resources, but more importantly allow running queries, inspecting dependencies and generating reports across the lifecycle.

In order for DM (and other CLM applications) to enable this kind of linking, it implements the OSLC specification, in particular the Architectural Management (AM) sub-specification. This specification enables a linked data approach to integrating lifecycle resources. Linked data describes a method of representing resources such that they can be interlinked and be more useful. This data is often represented in RDF. In addition to the requirement that an RDF resource is identified with a web URI, a linked data approach requires such URI to be deferenceable to useful machine readable data that includes links to other related resources.

In order to implement the OSLC AM specification, DM exposes its design resources as OSLC AM resources. Specifically, it adds the oslc_am:Resource type as another type of design resources (using multi-classification). It also adds a number of expected OSLC AM properties to its resources, including a title property (`dcterms:title`) and a description property (`dcterms:description`). Since equivalent properties may already exist in a domain, DM allows a domain definition to specify those equivalent properties (e.g., a domain may specify that its `domain:name` property is equivalent to `dcterms:title`). Also, by default, DM exposes all triples of a resource to OSLC except those triples whose predicates (used properties) are flagged as private in the domain. DM uses the set of exposed

(non-private) properties of a given class to construct that class's OSLC resource shape. This shape can be retrieved for every resource and informs a linking OSLC application of what properties to expect when dealing with that resource.

5 Related Tools

While there is no disagreement on the nature of design being a collaborative effort, not all design tools today support collaborative features. This section highlights three notable design tools that boast collaborative features and compares them to DM.

Collaborative Protégé [21] is an ontology and instance editor that allows concurrent user access. It enables comments on ontologies and their changes. Users can create discussion threads, create proposals for changes, and vote on them. There is also live chat support within the editor. However, unlike DM, the tool does not have lifecycle integration capabilities, has limited SCM support, and no web access.

TopBriad Enterprise Vocabulary Net [22] is a web-based tool for the collaborative development and management of semantic web vocabulary. It supports role-based access control, vocabulary publishing, review and approval, audit-trails and parallel development. However, unlike DM, the tool has no traceability across the lifecycle and has limited SCM support.

Objecteering Teamwork [23] is a collaborative modeling tool. It supports a central shared repository, parallel development, compare/merge and integration with SCM tools. However, Unlike DM, the tool has no web-based access and no role-based user access. It also does not support user-defined domains. Finally, it does not leverage the semantic web architecture.

6 Conclusion and Future Work

The software design process has become more challenging than ever. Some of the main challenges have been outlined and discussed in this paper, including: the need to express designs in a proper formalism, the need to find, access and collaborate on designs by different stakeholders, the need to link designs to other resources in the life cycle in order to ease navigation and perform traceability analysis, the need to increase awareness of project activities that relate to design, the need to plan for design variability and change through configuration management, and the need to boost the development process by supporting parallel development.

Design Management (DM) is a new collaborative design application that we developed to address the outlined design challenges. Some of the features provided by DM include: the support of several structured and non-structured design domains with the ability to define new domains, the storage of design in a central repository with web access to ease collaboration, the ability to link designs to other lifecycle resources with the benefits of allowing traceability, analysis and report generation cross the lifecycle, the ability to have a customizable dashboard that makes designers aware of project activities, the ability for designers to create different configurations for their designs to capture their variability and perform changes in a orderly manner,

the ability to comment on, review and approve designs, and finally the enablement of parallel development on designs using a traditional or an agile development process. DM is able to provide many of these features by adopting the semantic web architecture. DM can also integrate with other lifecycle applications, using a linked data approach, by implementing the OSLC specification.

Furthermore, as a new application, DM has some current limitations that we plan to overcome in future revisions. For example, it currently lacks a declarative way to define the concrete (graphical or textual) syntax of a design domain. This capability is important especially for user-defined domains. DM also lacks a way to declaratively specify a migration plan for designs when their domains evolve in a non-compatible way. Moreover, DM currently lacks a way to define inference rules in a domain to automatically compute derived information in designs to facilitate reasoning about them and checking their consistency. It also lacks a way to define and run design transformations on the server directly. We also plan to conduct case studies to evaluate DM's use in industrial settings and report on its impact on productivity.

References

1. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice, 2nd edn. Addison-Wesley (2003)
2. Otero, C.: Software Engineering Design: Theory and Practice. Auerbach Publications (June 2012)
3. Rozanski, N., Woods, E.: Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, 2nd edn. (2011)
4. Leroux, D., Ramaswany, V., Yantzi, D.: Design matters: Collaborate, automate, innovate and be agile, IBM Software Technical White paper (2012)
5. Unified Modeling Language, Superstructure v2.2, `http://www.omg.org/spec/UML/2.2/`
6. Model Driven Architecture, `http://en.wikipedia.org/wiki/Model-driven_architecture`
7. Rational Software Architect Design Manager, `http://www-01.ibm.com/software/rational/products/swarchitect/designmanager/`
8. IBM Rational Jazz Platform, `http://www-01.ibm.com/software/rational/jazz/`
9. Open Services for Lifecycle Collaboration (OSLC) specification, `http://open-services.net/`
10. Business Process Model and Notation v2.0, `http://www.omg.org/spec/BPMN/2.0/`
11. Rational Software Architect, `http://www.ibm.com/software/rational/products/swarchitect/`
12. Rational Rhapsody, `http://www-142.ibm.com/software/products/us/en/ratirhaparchforsoft`
13. Systems Modeling Language v1.3, `http://www.omg.org/spec/SysML/1.3/`
14. OWL Web Ontology Language, `http://www.w3.org/TR/owl-features/`
15. RDF Primer, `http://www.w3.org/TR/2004/REC-rdf-primer-20040210/`

16. SPARQL 1.1 Query Language, http://www.w3.org/TR/sparql11-query/
17. Collaborative Lifecycle Management, https://jazz.net/products/clm/
18. XML Schema, http://www.w3.org/XML/Schema.html
19. Meta Object Facility v2.0, http://www.omg.org/spec/MOF/2.0/
20. Semantic Web, http://www.w3.org/standards/semanticweb/
21. The Protégé project, http://protege.stanford.edu
22. TopBriad Enterprise Vocabulary Net, http://topquadrant.com/solutions/ent_vocab_net.html
23. Objecteering Teamwork, http://www.objecteering.com/products_teamwork.php

Umbra Designer: Graphical Modelling for Telephony Services

Nicolás Buezas[1], Esther Guerra[1], Juan de Lara[1], Javier Martín[2], Miguel Monforte[2], Fiorella Mori[1], Eva Ogallar[2], Oscar Pérez[2], and Jesús Sánchez Cuadrado[1]

[1] Department of Computer Science
Universidad Autónoma de Madrid, Spain
[2] Almira Labs, S. L.
Science Park – Tres Cantos, Madrid, Spain
http://www.almiralabs.com

Abstract. Almira Labs is a software company that develops value-added services for the telecommunications industry. It is focused on innovative technologies that enable enterprise business and mobile and landline operators to offer next-generation voice-driven applications for all types of phones. Telephony services are built atop the proprietary *Umbra framework*, which is a Java API relying on the JAIN SLEE standard for event-based communication applications.

This paper describes *Umbra Designer*, a novel graphical modelling tool for the visual development of telephony services, from which Java code for the *Umbra framework* is synthesized. In this way, it is easy to develop ready-to-use services, even by users not familiar with the Java API or the JAIN SLEE standard. We also report on some experiments aimed at measuring the efficiency gain derived from using the graphical tool, compared with coding directly using the Java API.

Keywords: Model-Driven Engineering, Telephony Services, Jain SLEE, Domain Specific Visual Languages, Code Generation.

1 Introduction

We are witnessing an exponential grow in the capabilities of mobile phones and in the functionality demanded by their users. The usual approach to deliver mobile services nowadays is the development of *apps* for a particular technology (iOS, Android, BlackBerry), running in the device of the client. This business model is supported by small software firms that need to develop innovative solutions in short times, in order to cope with an increasingly competing environment. This model has the drawback of the fragmentation of the mobile platforms, which implies different developments for each platform. Moreover, the functionality offered by different phones varies, from traditional landline phones to smartphones of the latest technology. Thus, companies may lose clients if they target particular platforms or assume some functionality for running their *app* [4].

P. Van Gorp, T. Ritter, and L.M. Rose (Eds.): ECMFA 2013, LNCS 7949, pp. 179–191, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Instead, a different alternative is to build services that do not run on the phone, but on the provider infrastructure through dedicated servers or the cloud, and which are accessed through phone calls [8]. This model has the advantage that is independent of the mobile phone platform used, and to a certain extent, of the phone capacities. While these "server-side" services cannot replace all types of phone *apps*, they are useful for many scenarios. They are normally driven by voice and DTMF key strokes. Examples include voice notes and voice-to-email services; services to inject customized background sounds in phone calls; the customization of the telephone keys to inject "voice smileys" in a conversation; as well as the typical Interactive Voice Response (IVR) applications of call centers, telebanking, credit card services, and so on. These services tend to require a reduced customer learning curve compared to those offered by mobile *apps*.

JAIN SLEE [1] is a standard of the Java Community Process for developing event-based telecommunication applications, which can be used to build server-side telephony services. Applications developed with JAIN SLEE can be deployed on any server implementing it. A Service Logic Execution Environment (SLEE) is an efficient event processing application environment with high throughput and low latency. A service is built by the construction and interconnection of components, and their subsequent deployment in SLEE servers.

Almira Labs has developed the *Umbra framework*, a Java API that leverages on the JAIN SLEE standard. The framework simplifies the development of JAIN SLEE applications by providing a higher-level view of the event flows and protocols involved in a telecommunications application, and provides true portability across different SLEE implementations. Still, using this framework requires specialized knowledge in JAIN SLEE and Java. In order to make service construction possible for non-experts – namely, people from customer companies – Almira and some researchers of the Universidad Autónoma have developed *Umbra Designer*, a tool for the graphical development of telephony services. The tool abstracts services in the form of hierarchical state machines using the events and actions available in the *Umbra framework*. The tool integrates a code generator that produces a *Maven* [9] Java project, which can be deployed as-is in JAIN SLEE servers for execution. The aim of the tool is to facilitate service modelling to non-experts in the API, and speed-up the development for programmers. This is demonstrated by a series of experiments where we have measured the efficiency of manual service development using the Java API with respect to using the tool, reporting an increase of productivity of more than 40% in the average case.

Paper organization. Section 2 provides some background on JAIN SLEE and the *Umbra framework*. Section 3 introduces the graphical modelling tool. Section 4 presents the evaluation results. Section 5 compares with related work, and finally, Section 6 ends with the conclusions and plans for future work.

2 Programming Voice-Driven Telephony Services

Our work targets at voice and key-strokes driven telephony services. These services normally run on the infrastructure of the service provider. Subsection 2.1

reviews a standard (JAIN SLEE) that can be used for building this kind of applications, while subsection 2.2 introduces a Java framework built atop JAIN SLEE which provides higher-level abstractions and true portability across SLEE implementations.

2.1 JAIN SLEE

JAIN SLEE is a standard by the Java Community Process that describes a Service Logic and Execution Environment (SLEE) architecture [1]. This architecture defines a component model for structuring the logic of communications applications as a collection of object-oriented components (called Service Building Blocks, SBBs), which can be composed into services. The SLEE architecture also defines the contract between these components and the container that will host them at runtime. JAIN SLEE applications are event-driven, which means that methods of the application are invoked when suitable events arrive. In this way, each SBB to be deployed in the SLEE identifies the event types that accepts, and defines event handler methods with code to process such event types.

The framework provides an API for handling events, resources and connections, facilities like timers and alarms, and standard interfaces to be implemented by SBBs. Still, the API is low-level, and service developers would benefit from higher-level abstractions, tailored to voice-driven telephony services, as explained in the next subsection.

2.2 The Umbra Framework

The *Umbra framework* hides the low-level details of JAIN SLEE to enhance performance and simplify the development of telephony services. It masks the JAIN SLEE components behind a simpler Java API that offers enhanced, scalable, carrier-grade performance. Another benefit is portability. As different providers offer different implementations of the standard, there may be JAIN SLEE compliant applications that do not run on every SLEE container. Moreover, when migrating an application from one vendor to another, parts of its code may need to be re-written to ensure smooth porting and compatibility. With the *Umbra framework*, the code works across JAIN SLEE platforms from the main vendors without any recoding.

The framework enables building applications mixing both web and telephony services. While JAIN SLEE can deal with low-level protocol issues at the back-end, a J2EE environment can provide a front-end for Internet services. Hence, *Umbra* enables the SLEE to offer web services the J2EE world can interact with.

Fig. 1 shows the typical structure of a service built with the *Umbra framework*. The upper part shows only the most relevant interfaces provided by the framework. The lower part (package *application*) presents a schema of the classes and interfaces that programmers have to develop. As we will see, *Umbra* is based on the definition of suitable event types and listeners for such events. The listeners contain call-back methods, invoked upon the reception of the events, which need to be programmed by the service developers.

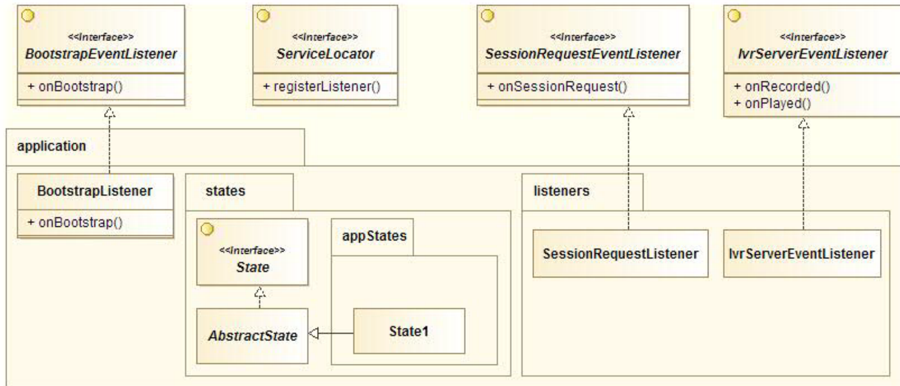


Fig. 1. Structure of an application using the *Umbra framework* (simplified)

A service is started upon the reception of the *onBootstrap* event. Hence, the developer needs to implement code to react to this event in the *BootStrapEventListener*. Normally, this code includes loading the needed resources and register event listeners through a *ServiceLocator*, especially *SessionRequest* events, which are events triggered by the SLEE container when the network requests a new session. Then, the service waits for incoming events, which may trigger specific actions like playing a message, recording a message, soliciting the user to press a key, and so on. These actions are supported by an *IVRServer* (a media server), which is a resource that needs to be identified upon bootstrap. The service will receive an event notification upon completion of the actions, as declared in the *IvrServerEventListener*. For example, the *onPlayed* method will be called upon completion of a *play* action, which plays a voice message.

Practice has shown that a suitable organization for event-driven applications is through state machines. Hence, a typical programming idiom for services built with the *Umbra* framework is the *State pattern* [2] in order to describe the different execution states of the service, the possible incoming events, the actions to be performed upon the arrival of events, and the state changes. This way, services normally define an interface *State* declaring all possible events, while the abstract class *AbstractState* defines default empty implementations for the event handlers. Therefore, developers have to create a subclass of *AbstractState* per application state (*State1* in Fig. 1), and override the methods for the events accepted by the state.

This organization is a Java implementation of a natural way of designing services as state machines. However, this style of programming is not enforced by the *Umbra framework* even though it is common practice. Hence, we decided to provide developers with a higher level representation of this pattern, closer to the abstractions of state machines. This way, the gap from design to implementation would be smaller. Moreover, this representation would facilitate the

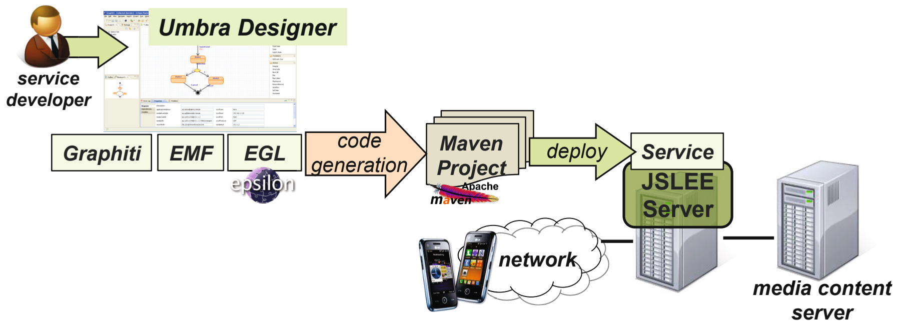


Fig. 2. Schema of *Umbra Designer*

communication with customers, most frequently non-technical people. The next section introduces a Domain-Specific Visual Language that helps in describing services at a higher level of abstraction.

3 Modelling Telephony Services with Umbra Designer

Fig. 2 shows the architecture of our solution. The developer or designer graphically defines the services using *Umbra Designer*. We have built this tool as an Eclipse plugin, using Graphiti [3] for the visual part, EMF [14] as modelling technology, and the Epsilon Generation Language (EGL) [13] for code generation. After validating the service, the tool generates a Maven project with code synthesized from the service. The code makes use of the *Umbra framework*, as explained in the previous section. Once compiled, the project can be deployed as-is on SLEE servers, like the *open cloud*'s Rhino application server [11].

Next, we introduce the main elements of the tool. Fig. 3 shows a screenshot with an example service. The tool abstracts services in the form of state machines, accommodating the State design pattern, as explained above.

The main canvas contains the description of a simple service. The service initial state is *Init*, where the service waits for incoming calls once a new session has been established. Hence, at the top level, the event from the initial state is *SessionRequest* (see arrow coming into state *Init*). The service designer does not have to take care of handling the *Bootstrap* event, as the tool itself will generate code to register the listeners for the events and actions used by the service (in the example all are *IVR* events), and identifying the needed resources. Events are depicted in blue (bold) over the arrows, while actions are shown in red below the events. The service plays a welcome message to each incoming call. This is modelled by a *Connected* event and the associated *PlayCollect* action. This action has the additional effect to demand pressing some key on the phone keypad. Thus, the service waits in state *KeyRequest* until the reception of some key stroke (event *Collected*). If the key pressed was "1" or "2", the service plays a different message in each case, as indicated by the *Play* action. Once the message

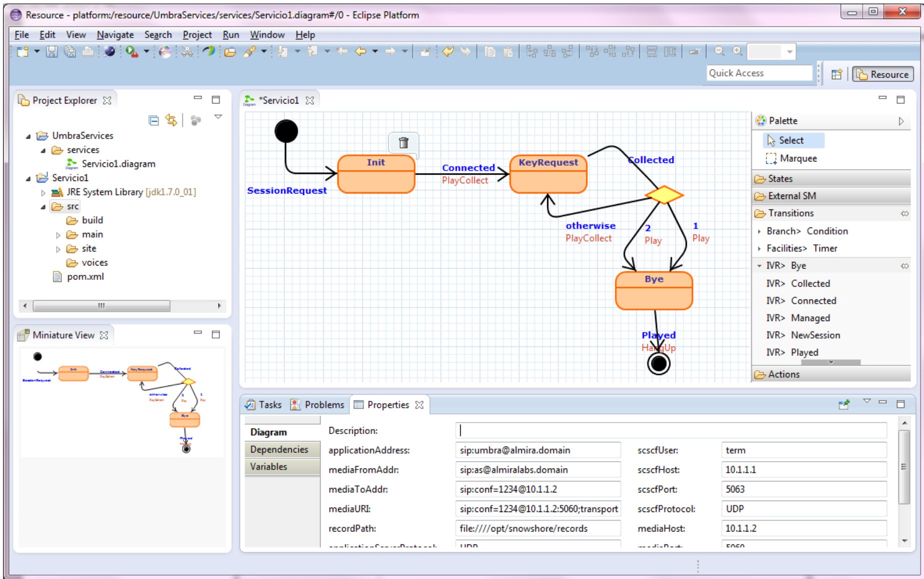


Fig. 3. Defining a simple service with *Umbra Designer*

associated to “1” or “2” is played (event *Played* in the transition going out from *Bye*), the service ends the call through the *Hangup* action.

The palette to the right contains the different types of states, transitions (i.e. events) and actions that can appear in services. The Properties view at the bottom allows configuring any item selected in the model; in the figure, it shows the configuration properties of the service and its resources. The tool has a contextual menu to validate the service (the detected warnings and errors are listed in the Problems view at the bottom), to generate Java code from the service, and to manage the generated Maven project (the project is shown in the Project Explorer tab to the left).

Altogether, the tool promotes an agile way to work, providing support for short cycles of modelling – code (re-)generation – deployment in an integrated environment. For instance, once code is generated for a service, developers can provide additional Java classes for further functionality (e.g. database persistence, which is not currently supported by our tool). We believe that this approach to agile modelling will be better accepted in development processes as it provides immediate value by code generation and service validation, and it is seamlessly integrated in the developer Java environment.

The abstract syntax of the services built with our tool is defined through a meta-model, of which Fig. 4 shows an excerpt. A state machine is configured through a number of properties (class *Properties*), like the addresses of the application and server, and the protocols used, among others. It is also possible to declare variables that will be available in Java actions. Two different types of variables are supported: shared variables (object attributes), used to pass

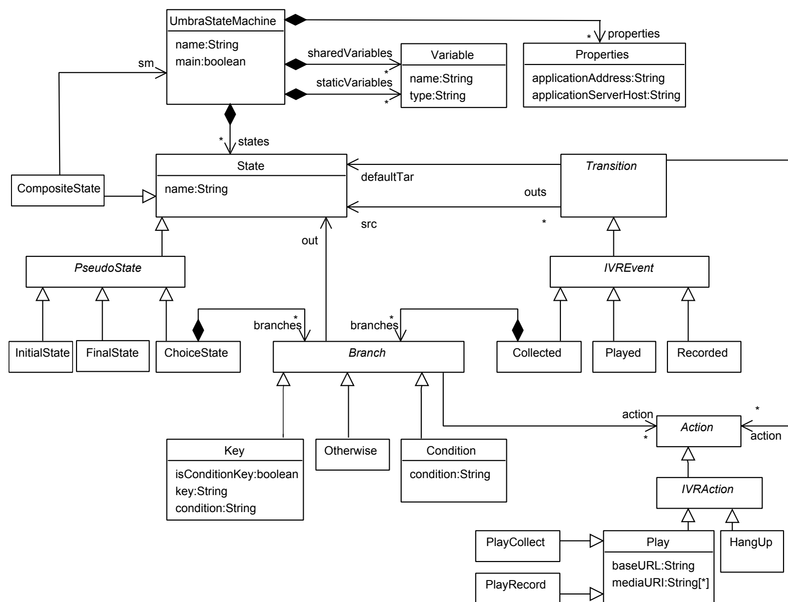


Fig. 4. Excerpt of the *Umbra Designer* meta-model

information between different service states, and static variables (class attributes), which retain their value between different service invocations.

We consider five types of states: initial, final, simple, composite and choice. Each state machine has one initial and one final states. Composite states enable hierarchical structuring of the machine, and contain a reference to a state machine which can be defined within the same model or externally in a separate file. Choice states have multiple output branches with a boolean condition each.

Transitions can be of different types that correspond to event types of the *Umbra framework*. They are organized in six categories: Interactive Voice Response (IVR) events, Call Control events, HTTP (reception of HTTP requests), SMS, Text-to-Speech (TTS) and Speech recognitions events (for clarity, Fig. 4 only shows IVR events). There are also general facilities, like timers. The initial event in a state machine is always of type *SessionRequest*. IVR events are concerned with playing and recording media streams or with key pressing in the phone keypad. Examples of supported IVR events include *Played* (fired when a media stream finishes playing), *Recorded* (when a recording finishes) and *Collected* (when the user presses some key). Call control events include those related to the connection, disconnection and transfer of call legs. SMS events are concerned with the reception of text messages. Finally, TTS events are those generated by a TTS engine, like the start, finish, pause and resume of a speech.

A transition may have associated a sequence of actions, to be executed when the transition is triggered. Actions rely on the *Umbra* API, and are organized in similar categories to those for events. A special action *JavaCode* permits adding Java code for more general actions not directly supported by the framework,

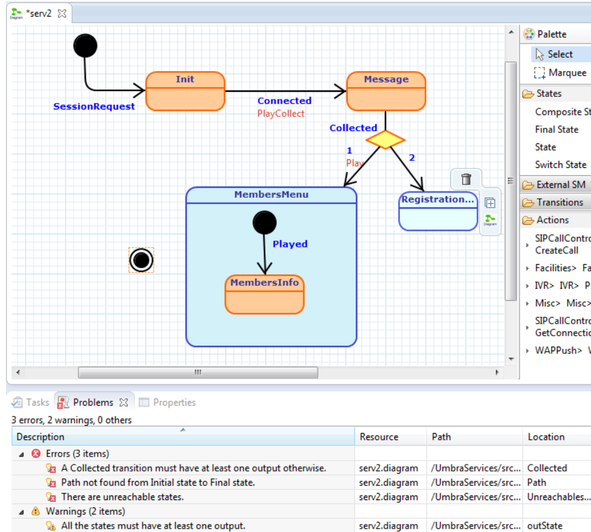


Fig. 5. Hierarchical modelling, and report of errors/warnings, in *Umbra Designer*

where it is possible to use the declared shared and static variables. The parameters of each action can be configured through a properties panel, as shown to the bottom of Fig. 6 for the case of the *PlayRecord* action.

Umbra Designer supports hierarchical modelling by means of composite states (see *MembersMenu* in Fig. 5), and through references to state machines defined in other diagrams (see state *Registration* in the same figure). In the last case, a contextual menu allows opening the referenced machine in a different tab, as well as expanding or occluding the diagram inside the state. This feature enables the construction of repositories of services, which then can be used to build other more complex services.

The tool enables simple validations of services prior to code generation, and displays the detected errors and warnings in the Eclipse Problems view (see bottom panel in Fig. 5, which contains the detected list of errors for an information and registration service for a gym). For example, the tool reports as an error any unreachable state, any state (different from the final state) without outgoing transitions, as well as non-existing paths between the initial and final states. Warnings concern the order in which some events and actions should occur, as some actions trigger the future occurrence of events. For example, the tool gives a warning if a *Played* event is declared, but there is no previous *Play* action. Notice that all these errors and warnings would be difficult to detect statically, if a direct encoding of the service in Java is used. However, we do not currently analyse *JavaCode* actions in transitions.

Once the model of the service is validated, it is possible to synthesize Java code from it. The tool creates a Maven Eclipse project, which can be deployed in an SLEE server. The generated Java code follows the State pattern [2], and

reflects the hierarchical constructs introduced in the model. Moreover, the code includes protected regions, so that if the developer modifies the code manually, this is not overwritten when code is regenerated again from the model.

4 Evaluation

In order to asses to what extent using *Umbra Designer* improves the productivity of service development, we have performed an experiment consisting in the construction of ten services of varying complexity using the graphical tool, and the comparison with the effort to develop the same services using directly the framework API (i.e. programming directly in Java).

We had two participants in our experiment, both last year undergraduates in Computer Science. The first participant had some knowledge of telecommunications services, but no deep knowledge of the *Umbra framework*'s API. The second participant had some 5 months of experience using the API. Each participant built 5 services using the tool and 5 different ones using the API. Each service was built in a different session, on a different day. The participants were given enough time to read each service description and think a solution. When they were ready, we measured the time they employed to implement the solution, one using the graphical tool and the other using directly the Java API. This way, we leave out effects related to problem understanding and solution design, and strictly measure service production efficiency by two different means.

The services used in the experiment varied in complexity, ranging from simple ones (five states and few transitions) to medium size (more than 15 states and 35 transitions). In each session, the participants were given textual definitions of the service to be developed in the session. As an example, the description of one of the services was the following: "*Build a voice service for a computer repair shop. The service will play a message, and then, it will solicit the year in which the computer was bought. The user should type the solicited year using the telephone keyboard. Then, if the computer is still in the guarantee period (2 years), the service will solicit the serial number and the address, which will get recorded*". Fig. 6 shows the service finally built for the previous service definition, using the graphical tool. The Properties view contains the configuration of the actions in the transition entering state *SolicitarDireccion* (*Solicit Address*). The transition has two actions, one for recording the address (whose configuration is shown), and the other one is just one line of Java code to transform the different key strokes into a String service variable (not shown).

Other services built include a game for guessing a number, a time service which informs of the current time, a taxi call service, a simplified airport information service and a service for pizza ordering (see Table 1).

Table 1 summarizes the experiment results. The columns show: (1) the name of the service; (2–5) the size of its model-based solution (number of states and transitions, cyclomatic complexity and lines of extra Java code in *JavaCode* actions); (6–7) the number of source lines of code (SLOC, not counting blank lines or comments) of the service that are generated by the tool or hand-coded

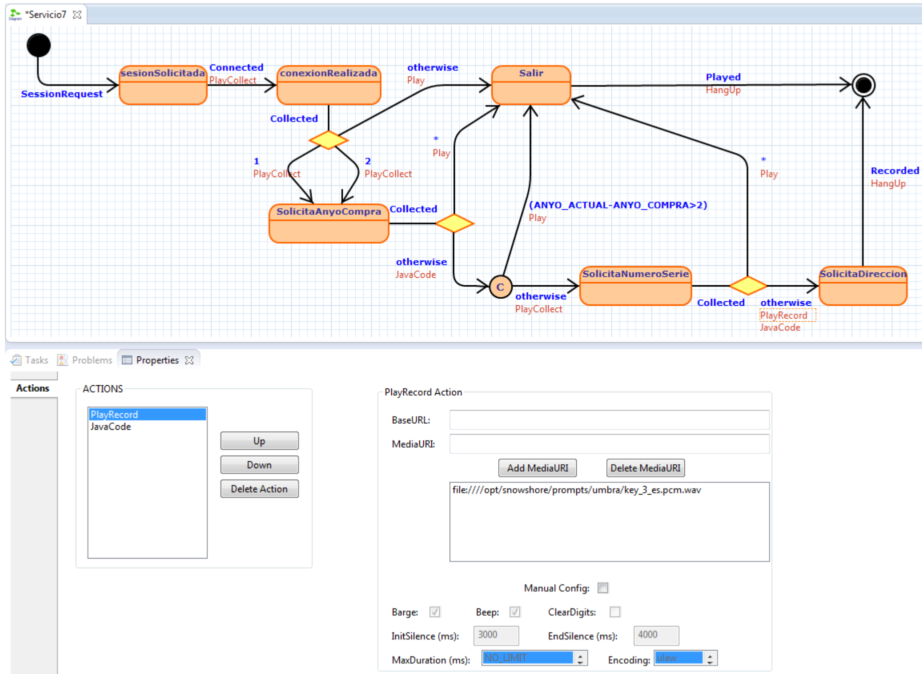


Fig. 6. Service #7: a simple service for a computer shop

using the API; (8–9) the minutes taken to build the service using the tool and the API; and (10) the efficiency gain when using the tool compared to using the API (minutes and percentage). In the case of using the API, the measurements also include the creation time of additional artefacts, like property files, needed to deploy the service (but automatically generated by the tool).

The experiments show good correlation between the number of SLOC of the hand-coded solution and of the code generated by the tool. Regarding productivity, by using the tool we observe an increase of around 45% in the average case. In all cases, the time to develop a service with the tool was less than using the API directly. Fig. 7(a) shows a graphic showing the net gain with respect to service size (SLOC of the hand-coded solution), while the right shows the percentage gain with respect to size. The graphic shows higher percentual gains for smaller services; however, the highest net gain was with the largest service. We noted higher gains in cases were the service accommodated well the abstractions of state machines: few cycles, few decision nodes, and few extra lines of code.

Altogether, the experiment shows benefits in efficiency when using the graphical tool. Moreover, we believe that it provides further benefits concerning: built-in validation checks, maintainability, understandability and reutilization of services. Experiments to assess these properties are subject to future work. As a preliminary result, we experienced in measuring the effort gain in maintainability, where a modification to service #1 consisting in the introduction of an error

Table 1. Evaluation of the construction of several services

	Model Size				SLOC		Time (min.)		
	States	Trans.	Cycles	Java Extra	Tool	API	Tool	API	**Gain (min./%)**
#1 : Message + Key	5	6	3	0	383	395	7	30	**23 (76%)**
#2 : Taxi Call	8	10	4	2	441	403	9	21	**12 (57%)**
#3 : Guessing Game	8	13	6	3	448	415	23	42	**19 (45%)**
#4 : Postal Code/ Time Warning	6	12	8	12	439	453	30	53	**23 (43%)**
#5 : Traffic	9	12	5	4	450	453	20	35	**15 (42%)**
#6 : Survey	10	14	6	0	529	477	15	33	**18 (54%)**
#7 : Computer Shop	9	13	6	4	493	501	20	40	**20 (50%)**
#8 : Airport	11	23	14	8	542	514	60	90	**30 (33%)**
#9 : Time Service	5	4	1	269	640	749	24	36	**12 (33%)**
#10 : Pizza Orders	17	52	37	60	963	925	150	200	**50 (25%)**

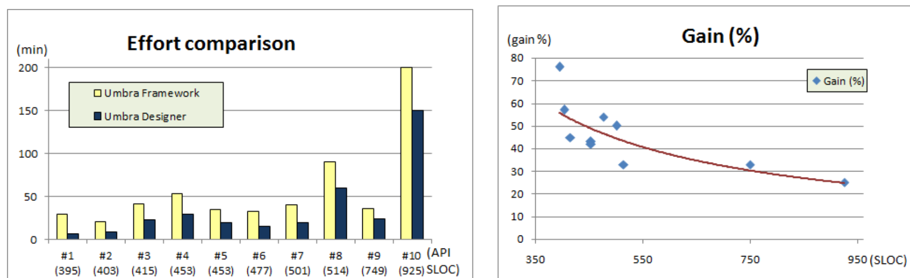


Fig. 7. Development effort comparison (left). Efficiency gain (right).

message on certain events was introduced. In this case, using the graphical tool led to shorter times (6 minutes vs 15 minutes). A finding related to this issue was that both participants found useful to draw state machine-like diagrams on paper, as a design sketch, before starting coding using the API. This means that the graphical model was deemed a good abstraction to describe services.

5 Related Work

The need for developing and making available telecommunication APIs, is discussed in [4]. Similar to our rationale, the author foresees the possibility of telecommunication application stores – similar to those of Apple and Android – based on the availability of service creation environments.

There are several implementations of the JAIN SLEE, like Mobicents [10] and OpenCloud [11]. The latter includes a visual builder for services, the Visual Service Architect (VSA) [15]. In this environment, a service is described by an application-scenario diagram (to configure properties, protocols and resources), state machine diagrams (to describe service states) and flowchart diagrams (to describe actions). VSA targets general JAIN SLEE services, not necessary for telephony, and hence lacks high-level constructs (both events and actions) for voice-driven telephony services, as we provide in *Umbra Designer*. VSA state

machines and flowcharts tend to be of lower level of abstraction due to the lack of constructs like hierarchical states, choice states and key strokes branches, among others.

In [5], the authors present an environment for service composition using MetaEdit+. SBBs are programmed in Java, which become reusable and can be composed graphically. Our approach is different as the blocks themselves are modelled using state machines, from which Java code is generated. Also in the context of MetaEdit+, in [7], the authors describe a graphical language to define simple call processing services. This language allows defining the flow for handling incoming calls like rerouting them, or sending a message upon their reception. The services can be serialized in XML. In our case, state machines are a better abstraction for the event-driven nature of voice-driven telephony services, while we need to generate more complex Java code. Another language for telephony service creation is SPL [12], a scripting textual language with formal semantics. It differs from our approach in that it is targeted to experienced programmers, and its formal semantics enables critical properties of services to be guaranteed. We plan to address exhaustive testing of service models against user actions in future work.

VoiceXML [16] is a W3C standard to describe interactive voice dialogues between a human and a computer. VoiceXML files are played by voice browsers, and contain tags that instruct the browser to provide speech synthesis, automatic speech recognition, dialog management, and audio playback. VoiceXML applications are accessed via HTTP, while we use phone protocols.

On a final comment, there are not many published results of efficiency of MDE in practice [6,7]. Our work also contributes in this direction, by describing a specific successful scenario for the applicability of MDE.

6 Conclusions and Future Work

In this paper, we have presented *Umbra Designer*, a tool for the graphical development of "server-side" telephony services. The tool facilitates the construction of services by non-experts. It includes a code generator that relies on the *Umbra framework*, a Java API that is used to build services based on the JAIN SLEE standard. Some initial experiments show promising results regarding efficiency gain for the construction of services, with respect to a direct use of the API. We believe this will be especially interesting for customer companies and users with no deep knowledge of JAIN SLEE or telecommunication applications.

In the future, we plan to improve the tool with further functionality to consider more advanced services. In some cases, we think it is possible to generate automatically a graphical user interface for a mobile app (iOS, Android) starting from the state machine description of the voice dialogue. We also plan to investigate exhaustive testing of service models against user actions and to make the tool publicly available in the immediate future.

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

References

1. Ferry, D., ODoherty, P.: JAIN SLEE (JSLEE) v1.1. Technical report, Java Specification Request 240 (2008), http://jcp.org/en/jsr/detail?id=240
2. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley (1994)
3. Graphiti Project, http://www.eclipse.org/graphiti/
4. Hall, S.: Evolving the service creation environment. In: Proc. ICIN 2010, pp. 1–6 (2010)
5. Hulshout, A.: Service creation with MetaEdit+. A telecommunications solution. In: Proc. Code Generation (2007)
6. Hutchinson, J., Whittle, J., Rouncefield, M., Kristoffersen, S.: Empirical assessment of MDE in industry. In: ICSE 2011, pp. 471–480. ACM (2011)
7. Kelly, S., Tolvanen, J.-P.: Domain-Specific Modeling: Enabling Full Code Generation. Wiley-IEEE CS (2008)
8. Martín-López, J., Monforte-Nicolás, M., Merino-Moreno, C.: Finding services and business models for the next-generation networks. In: Recent Developments in Mobile Communications - A Multidisciplinary Approach. InTech (2011)
9. Maven, http://maven.apache.org/
10. Mobicents JAIN SLEE project, http://code.google.com/p/jain-slee/
11. Open Cloud Rhino SLEE server, http://www.opencloud.com/products/rhinoapplication-server/real-time-application-server/
12. Palix, N., Réveillère, L., Consel, C., Lawall, J.: A Stepwise Approach to Developing Languages for SIP Telephony Service Creation. In: Proceedings of Principles, Systems and Applications of IP Telecommunications, IPTComm, pp. 79–88. ACM Press, New York City (2007)
13. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.A.C.: The Epsilon Generation Language. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 1–16. Springer, Heidelberg (2008)
14. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley (2008)
15. Visual Service Architect. VSAD, https://developer.opencloud.com/devportal/display/
16. W3C Voice Browser Working Group, http://www.w3.org/Voice/

Experience with Industrial Adoption of Business Process Models for User Acceptance Testing

Deepali Kholkar, Pooja Yelure, Harshit Tiwari,
Ajay Deshpande, and Aditya Shetye

Tata Consultancy Services,
India
(deepali.kholkar,ajay.deshpande,aditya.shetye,
pooja.yelure,harshit.tiwari)@tcs.com

Abstract. Model based testing or the generation of tests from machine readable models has been widely deployed in industry for testing embedded systems and devices. Attempts are being made to extend its use to business systems. However, in spite of its potential for process improvement, its large-scale adoption for testing business systems is not yet seen, mainly due to little data being available on such use. This paper presents the findings from industrial deployment of a business process model based testing approach for User Acceptance Testing of large banking and insurance systems. The approach met with easier acceptance from the user community due to use of business process models and has proved to scale to very large models. It resulted in an overall productivity benefit of 20-30% in test design and planning, in addition to digitization of domain and process knowledge and has been successfully adopted organization-wide. Benefits as well as issues faced in large-scale adoption are discussed along with solutions found and open problems.

1 Introduction

Model based testing (MBT) has been extensively researched over the last few decades [2][4][6][7][10-12]. Its industrial adoption is common for safety-critical systems, however, only recently it is being applied for testing of business applications [4]. Large-scale adoption in this area is not seen, an important reason being lack of available data on such deployments. Other factors are few off-the-shelf implementations of MBT, difficulty of switching to the modeling paradigm, cost and complexity of creating models, large size of generated test suites and scalability [7].

Traditionally MBT methods have used state machines or formal specifications as input [5][6], both of which are hard notations for non-technical teams that test business applications. It is only recently, with the advent of business process models (BPM) that are easily understood by business and test teams that this community could be considered a target user set for MBT. BPM has grown in popularity to become the de-facto method for documenting business processes. Standards such as OMG's Business Process Modeling Notation (BPMN) have brought uniformity in modeling formats across the available BPM tools [3]. Scalability becomes an issue with increase in size of state models [6].

P. Van Gorp, T. Ritter, and L.M. Rose (Eds.): ECMFA 2013, LNCS 7949, pp. 192–206, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Most available MBT tools focus on automated generation of tests without test data generation [3][4]. Conformiq generates test data as well [5]. Validation of model content is not addressed in available tools. We developed a toolset for BPM based MBT in-house since at the time there was no available work using BPM for test generation and also to apply constraint solving techniques for process model validation and test data generation.

This paper reports on the experience of applying our MBT approach [1] in an organization that conducts User Acceptance Testing (UAT) for major financial corporations. A number of pilot case studies were conducted on large real-world financial applications. The paper documents the findings from this experience including the preparatory work necessary to apply the approach on a large scale across the organization, results from the pilot case studies, problems encountered and open issues. Results indicate an overall productivity gain of 20-30% in test planning and more accurate estimation of testing effort due to digitization of process and domain knowledge. Performance of our method on other measures, viz. cost and complexity, scalability and test suite size is also discussed.

2 The UAT Context

User Acceptance Testing of business applications involves validation for important/critical user stories. The UAT process in industry is predominantly manual with consequently low productivity and has heavy domain knowledge dependency.

The organizational unit that applied our approach is a 1200+ member team responsible for the UAT for major multinational banks and insurance corporations, on financial applications in areas of Corporate Banking, Consumer Banking, Capital Markets and Insurance. UAT is done on each software release before it is rolled out into production.

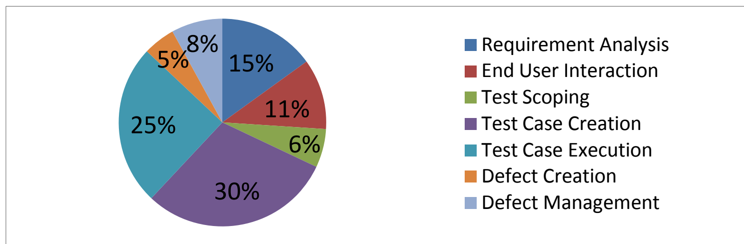


Fig. 1. Effort distribution in UAT

Critical activities performed for each release and their average effort distribution in the manual UAT process are depicted in Fig.1. This data was obtained from effort logs maintained by the team as part of their operational process. As seen from the figure, most effort is spent in Requirement Analysis, Test Case Creation and Execution. The first two of these constitute the Test Planning phase which also includes Test Scoping.

Table 1. Problems faced by UAT team

1. Productivity	• Improve productivity in Requirement Analysis, Test Creation and Execution, Test Maintenance
2. Quality	• Dependence on Test lead for coverage of test scope. Errors in scoping lead to rework and potential post production defects • Lack of standardization in test processes and formats • Unavailability of end-to-end process knowledge since teams work in silos. Lack of awareness of impact of change on upstream and downstream systems results in post-production defects
3. Planning	• Inaccuracy of testing estimates leading to schedule slippage • Need for accurate change impact estimation

The unit was faced with multiple challenges such as showing year-on-year productivity, safeguarding domain knowledge of the team when faced with attrition and accurate test cycle estimation to avoid production release delays. Specific problems to be tackled in order to address these were identified by the UAT team as in Table 1.

Although taken from the UAT context, the activity break-up as well as challenges are true of the Testing activity in general and form a set of goals and metrics for measuring effectiveness of an MBT approach.

3 The MBT Approach

Our MBT approach [1] and toolset for it named Assurance Workbench (AWB) is depicted in Fig.2. The approach comprises capturing a model of the application and automated test generation from the model in two stages – Scenario and Test case generation and is described here using one of the pilot case studies viz. Insurance New Business as a running example.

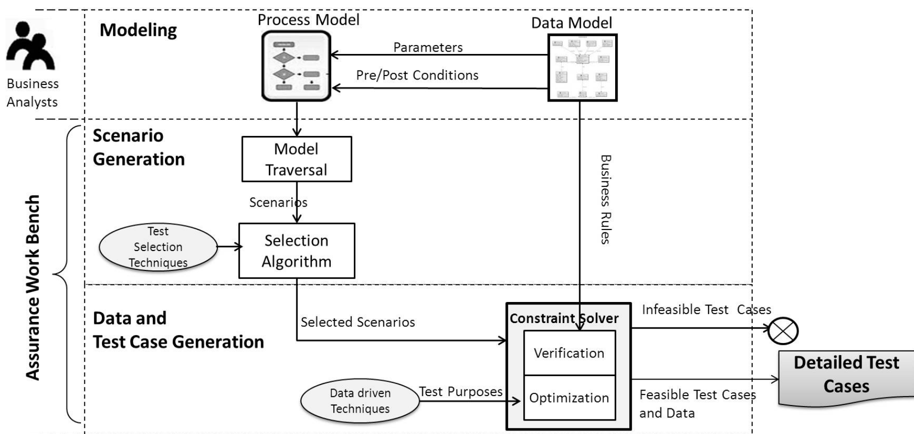


Fig. 2. Our MBT approach

3.1 Modeling

Business Analysts capture the important workflows of the system in the form of business process maps. Fig. 3 shows the top level New Business process flow drawn in BPMN, which depicts issuance of a new Insurance policy for a customer, made up of Activities (lowest level atomic tasks) such as *Select Product*, Sub-Processes (complex tasks) such as *Generate Illustration* and Gateways (decision or branch points). Each sub-process is detailed in a separate process diagram. The New Business process hierarchy consisted of this main process and 12 sub-processes, containing a total of 12 gateways, 28 branches and 75 activities.

The domain data model is captured as a Unified Modeling Language (UML) class diagram, shown on the right in Fig. 3. Business Rules are specified as Object Constraint Language (OCL) constraints on the data. The New Business pilot data model had 6 classes, 20 attributes and 4 business rules. A sample Business Rule is *SexTypeSelectionRule* which states that when StateCode is NY, Sex can be specified as Unisex only, else as Male/ Female. The OCL expression for this rule is

((*self.StateCode* = Insurance::Enum_State_Code::NY) and *self.Sex* = Insurance::EnumSex::Unisex)) or not(*self.StateCode* = Insurance::Enum_State_Code::NY)
where *self* denotes the class *InsuredDetails* to which this rule is attached, of which *Sex* and *StateCode* are attributes.

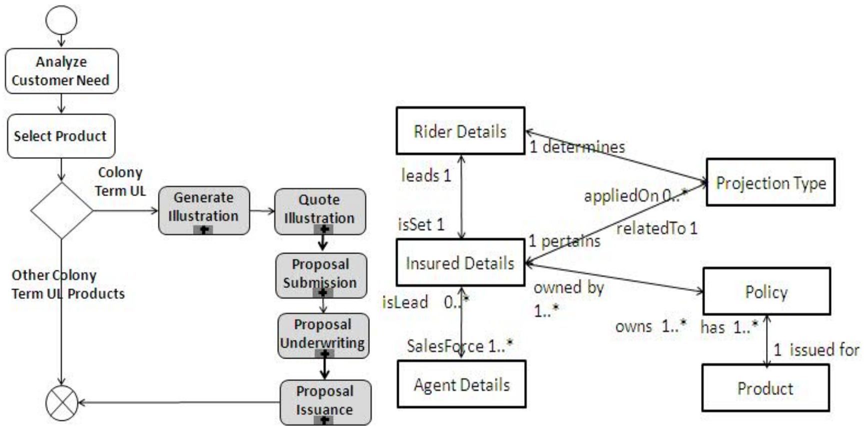


Fig. 3. Process and Data model: Insurance New Business

Input and output parameters of process steps form the link between the process and data models as shown in Fig.2. E.g. an instance *p* of class *Product* is attached as input parameter to the task *Select Product* of Fig. 3. Processes can be annotated with rules in the form of pre/ post conditions for tasks e.g. *Colony Term UL* in Fig. 3. These are expressed as OCL constraints on parameters of the process. The OCL expression for pre-condition *ColonyTermUL* is

p.ColonyTermULType = Insurance::Enum_ColonyTermULType:: ColonyTermUL
where *p:Product* is a parameter.

3.2 Scenario Generation

The process model graph is traversed using standard Breadth-first traversal to generate an exhaustive set of paths. Each generated end-to-end path consisting of sequence of branches, conditions and tasks forms a business/ test **scenario.** Branches are sufficient to uniquely identify a scenario. Table 2 shows the business scenario matrix (BSM) for the New Business process generated by our tool, in which each row is a scenario defined by the branches in the column values. Each column denotes a gateway. e.g. Row #1 is the scenario defined by branches SelectProduct=Colony Term UL, CustomOption=Y, Projection= LGS and so on. Only 10 of the 12 gateway columns are shown here due to space constraint.

The set of scenarios generated is usually very large and can be reduced using the following Test Selection strategies

- **Minimal selection:** Selects a minimal test set from the exhaustive set, covering every element of the process graph at least once, i.e. gives standard node-edge coverage. Useful to get the most efficient test-suite covering all system functionality. The minimal test set for New Business contained 7 scenarios out of 290, shown in the BSM in Table 2.
- **All Combinations/ Pairwise selection:** Selects the minimal test set covering all combinations/ pairwise [9] combinations of branches from selected gateways. Used when maximum coverage of combinations needs to be ensured for critical branches of the workflow.
- **Selective generation:** Selects a reduced test set containing paths passing through selected Gateways/ Activities/ Pre/ Post Conditions only. Used when test cases for only specific parts of the system are required. Useful for getting the impacted test cases by specifying elements of the model which have changed.

Table 2. Scenario Matrix for New Business

#	Select Product	Custom Option	Projection	Face Amt	Catchup Provision	Catchup Tier1	Projection Option	Rider	Waiver of specified premium	Form Generation
1	CT- UL	Y	LGS	N	Y	N	Y	Y	Y	Welcome Letter
2	CT- UL	N	-	-	-	-	-	N	-	PA Disclosure
3	CT- UL	Y	LGS	Y	Y	N	N	Y	Y	Policy summary
4	CT- UL	Y	LGS	Y	Y	Y	N	Y	N	Demo of CPB
5	CT- UL	Y	LGS	Y	N	-	N	N	-	-
6	CT- UL	Y	LGNS	-	-	-	-	-	-	-
7	CT- OUL	-	-	-	-	-	-	-	-	-

3.3 Test Case Generation

For each **scenario** selected, pre- and post-condition and business rule constraints are automatically translated by the toolset to a specification for the model checker Symbolic Analysis Laboratory (SAL) [8]. *Sal-atg* is used to solve these constraints to find

a solution state that represents appropriate data values for testing the scenario. Conflicting constraints result in no solution and constitute infeasible paths that are filtered out as depicted in Fig.2.

A **test case** is defined as the **scenario** plus **data.** Multiple test cases can be obtained for a scenario using any of several data-driven strategies, viz.

- **Boundary Value Analysis (BVA)**: Generates 5 values per integer attribute, on and either side of each bound of value range.
- **All Value Coverage:** Covers all valid values of an attribute.
- **Positive/ Negative tests:** User-defined combinations of data values

Each data condition is translated to a test purpose for SAL, as shown in **Fig.2**. SAL tries to satisfy multiple test purposes in each solution, generating an optimized solution set. A second level of infeasible test filtering takes place here since conflicting combinations of test purposes, workflow paths and business rules are eliminated by the solver. E.g. when attribute StateCode (values: CA, NY, VT, NJ, MT) and Sex (values: Female, Male, Unisex) selected for All Value coverage, test purposes created are

TestPurpose01= (StateCode = CA), TestPuropose02 = (StateCode = NJ)…
TestPurpose11= (Sex= Female), TestPurpose12 = (Sex= Male)...

Exhaustive set would be 5*3 = 15 combinations. SAL combines multiple independent test purposes and generates an optimized set of 6 solutions that conform to the business rules, in this case *SexTypeSelectionRule* explained above.

Automated validation for conflicts and generation of test data sets in accordance with rules enforces semantic correctness.

A sample generated test case from the New Business test suite is shown in **Table 3** and is defined by the test (pre-) conditions, sequence of SubProcesses and Tasks and their parameters with generated data values.

Table 3. Test case sample from New Business

Test Case #	Test Condition	Sub Process	Task	Parameter field	Value
1	Colony Term UL		Customer Need Analysis		
			Select Product		
		Insured Details	Select Insured Details Tab	InsuredAge	27
				Branch	16
				State_Code	NY
				Sex	Unisex
	…………..				
	Rider = Y	Rider Details	Select Policy Rider Tab	PolicyOwnersAge	64
				Child Rider Units	0
				Waive_premium	Y
				Face Amount	50,000

4 Application of MBT

The UAT organization selected key processes from various application areas for carrying out the pilot case studies. A total of 20 pilots were carried out, of which results from 9 are presented here. The processes chosen for the case studies were all large and complex to check scalability for real-life models. Three preparatory stages were needed before actual application of MBT could commence

1. Modifications to existing UAT process to accommodate BPM
2. Arriving at a strategy for modeling and
3. Training the workforce on modeling.

This was the additional investment for effecting the process change, which the organization decided to make, expecting the knowledge captured in BPM would help manage the problem of knowledge loss due to attrition and facilitate reuse.

4.1 Changes to the UAT Process

The manual UAT process relies completely on the domain knowledge of Business Analysts and Test Leads for test design. Business Analysts create Business Requirement and Functional Requirement Documents (BRD and FRD) for each feature. These are used by Test Leads as the basis for deciding testing scope and designing tests. Execution of Test cases is manual.

The new BPM based process required analysts to create process maps and specify business rules instead of BRD/ FRD. Impacted test cases, UAT scenarios and test cases were to be generated automatically from the model. **Fig. 4** depicts the existing and changed UAT process.

4.2 Adoption of BPM

A modeling scheme was required in order to ensure uniformity across models created by different users, so that results across teams could be compared.

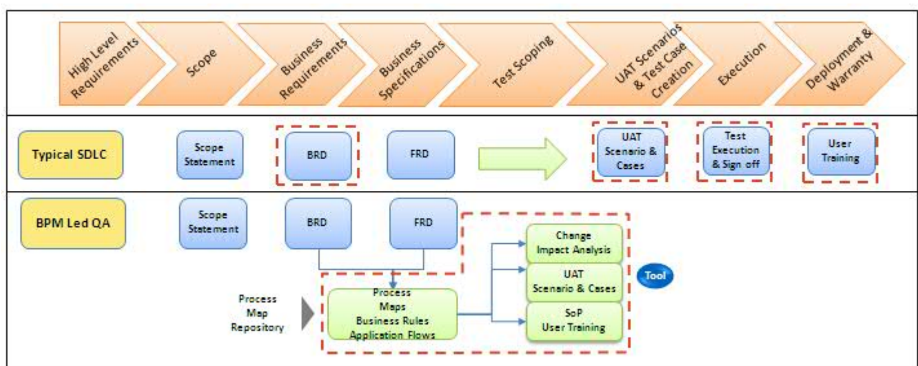


Fig. 4. Comparison of existing and MBT approaches for UAT

The UAT organization devised an architecture clearly defining the levels of abstraction at which the process model would be captured. Levels 1-4 in the process hierarchy capture generic information pertaining to the domain while Levels 5 and 6 contained system specific details. These are logical levels that may map to several physical levels.

- Level 1 gives the **product** level information
- Level 2 maps contain core **processes** that fall under that product
- Level 3 maps list down the **sub-processes** for the corresponding processes
- Level 4 maps have **activities** performed for a sub-process.
- Level 5 maps have the specific **tasks** that a user will perform to execute an activity. Data parameters and business rules are tagged here
- Level 6 maps have the **key strokes** that a user performs to execute the tasks.

This standardization enables automated processing of the information during test generation. Level 5 of the model is used to output the detailed test case sheet in Table 3. Level 6 is used to generate test cases at user interface action level, i.e. test scripts that can directly be used by testers for manual execution.

Separation of domain and system-specific aspects into separate layers was extremely interesting. The former could now be modeled by domain experts who did not need to know operational details of systems. Secondly, domain-specific or application-specific test cases could be generated by traversing down to the appropriate level in the model.

Training given to users included this architecture, the modeling notation, data modeling and specification of business rules through a simplified user interface. To ensure consistency of modeling, a maker-checker process is followed where a checker reviews content, levels, notation and standards for each process created by a maker.

4.3 Pilot Case Studies

The nine pilot case studies discussed in this paper along with the problem areas/ key features they represent are listed below.

- **Credit Cards:** Inaccurate testing estimates, non-standard test case formats and dynamic test cycles.
- **Payments process 1:** Immediate need from the end-user organization to show productivity benefits. Important module impacting 9 upstream and downstream sub-systems
- **Securities Trading:** Improper test planning, post production defects, high attrition problem. Complex functionality with lot of variations.
- **New Business:** Wide range of reusable functionalities.
- **Trade Processing:** Most mature process with good information of requirements, Planning, execution and Operations. Critical module impacting 45 upstream and downstream sub-systems
- **Wealth Management (WM), Payments processes 2 and 3, Check Processing:** Very large and complex processes with loops and multiple entry points. WM impacted 30 upstream and downstream sub-systems.

Table 4. gives the size of each process in terms of the number of sub-processes, gateways, gateway branches and leaf-level activities. The number of branches impact number of paths, adding complexity to scenario generation. Number of pre-conditions, classes and attributes signify complexity for the constraint solver. Number of Activities adds complexity to test execution. *NA* indicates data modeling was not done for the last four; being very large models, it was decided to focus only on their scenario generation, described in the next section.

Table 4. Complexity Of Case Study Models

Model	Sub Processes	Gate ways	Branches	Pre-Condition constraints	Activities/ Test Steps	No of classes	No of attributes
Credit Cards	5	4	14	26	31	3	6
Payments process 1	15	7	53	55	73	5	39
Securities Trade	67	17	100	56	147	8	71
New Business	12	12	28	26	75	6	20
Trade Processing	33	14	55	50	22	9	49
Wealth Management	103	142	401	622	573	NA	NA
Payments process 2	52	36	193	269	322	NA	NA
Check Processing	53	26	99	83	225	NA	NA
Payments process 3	112	71	637	775	849	NA	NA

For each pilot case study, models were captured, test generation and impact analysis done for a set of identified changes to the initial model. Generated tests were reviewed by the UAT team. Results are discussed in the section below.

5 Results

Results of test generation from the pilot case studies are listed in **Table 5** in chronological order. Column 3 shows the size of the generated test set, being quite large, test selection was employed to get a reduced test set. Column 4 shows the selection strategies chosen in each case by the UAT team. Column 5 gives the size of the reduced test set and Column 6 the percentage of tests selected in it.

5.1 Coverage

Exhaustive coverage of the specification was achieved using automated generation, from which desired coverage could be achieved using selection. Minimal selection gave a very efficient test set covering all tasks and branch conditions in the process at least once. Payments and SFS needed combination coverage of critical branch

conditions, which was easily possible using All combinations selection. Column 6 shows that in every case, a very small percentage of tests was sufficient to satisfy the selection criteria. Any other paths in the workflow that are not part of this selection but are of interest could be included by additionally using Selective generation described in Section 3.2. In cases where particular data values needed to be tested for, tests were obtained using Positive/ Negative selection.

Table 5. Test case generation results

Application process	#of Sub-Processes	Exhaustive Test Cases (paths)	Selection Type	Reduced Test Cases Selection	% Test cases selected	Manual Test Planning Effort (ph)	Automated Test Planning Effort (ph)	Effort Reduction %
Credit Cards	5	150	Minimal	5	3.33%	108	84	22
Payments process 1	15	8076	AllCombinations	12	0.15%	672	491	27
Securities Trade	67	17 million	AllCombinations	33	0.01%	1632	1224	25
New Business	12	290	Minimal	7	2.42%	163	120	24
Trade Processing	33	1741	Minimal	12	0.69%	192	146	24
Wealth Management	103	> 4 million	Minimal	47	0.01%	1200	900	25
Payments process 2*	52	> 300 million	Minimal	128	NA	NA	NA	NA
Check Processing*	53	> 30 million	Minimal	226	NA	NA	NA	NA
Payments process 3*	112	> 1 billion	Minimal	190	NA	NA	NA	NA

Total count of test cases not available since being very large model, processed in parts as described in section on Scalability below.* ***NA**: Effort data not available yet; data collection still underway.*

5.2 Productivity

The last three columns in Table 5 show the efforts for manual test creation, automated generation using MBT and the percentage reduction or overall productivity gain achieved, which ranges from 22-27%. The effort figures have been taken from a tracker in which the time spent on each activity by each person per release is manually captured.

The MBT effort includes time taken for initial creation and validation of the model, auto-generation and validation of tests. Average time for complete scenario generation is under 4 minutes. Average time taken by the constraint solver for data generation per test case (single data set) is between 1-2 minutes.

Change Management. Effort comparison between change impact analysis in the manual process and MBT approach for one of the projects is shown in Table 6.

Effort for understanding the change is the same in both cases. Impact computation in MBT is done using Selective generation (3.2) for the changed items in the model and takes 40% less time than in the manual case, as seen in Table 6. The overall productivity gain in total effort is 30%, as seen from the table.

Table 6. Effort comparison for Change Impact Analysis

Manual process (Effort in hrs)		MBT (Effort in hrs)	
Understand CR	16	Understand CR	16
Gap/ Impact analysis	40	Process flow creation/ modification in model	24
Discussion with stakeholders / SMEs	18	Discussion with stake holders/ SMEs	18
Update BRD	8	BRD/ FRD generation	Neglig-ible
Total effort	82	**Total effort**	58

Effort needed to make modifications in the test suite in manual and MBT approaches is shown in Table 7. Effort for scenario and test case modification is shown separately. Whether the change is an addition or update of functionality, the effort to modify/ re-write tests impacted by changes is very high in the manual case. In the MBT approach, change is made centrally in the model and automated re-generation of scenarios and tests resulted in 20-30% savings.

Table 7. Effort comparison for Test suite maintenance for a Change Request

Type of Change	Team Size	Scenarios			Test Cases		
		# of Scenarios to create/ update	Manual effort (Person hrs)	MBT Effort (Person hrs)	# of Test Cases to create/ update	Manual effort (Person hrs)	MBT effort (Person hrs)
Addition of New Functionality	30	10	40	32	45	200	156
Update Existing Functionality		6	30	23	120	360	260

It is interesting to note that for scenarios, effort needed was more for update of existing functionality than for addition, in both manual as well as MBT, since one needs to understand existing functionality before making the change. Conversely for test cases, effort was more for addition of new functionality in both approaches, since a lot of detail has to be added, while update takes less time due to availability of detail.

Activity-Wise Productivity. The graph in Fig. 5 gives an activity wise comparison of the approaches.

It shows that MBT gives a productivity advantage of about 30-40% over the manual approach for each activity. Having knowledge captured in BPM brings down the time taken to train new members of the team. The time taken for requirement analysis is reduced due to structured models and automated impact computation. Productivity in scenario/ test case creation has been discussed above, where the overall productivity was found to be 20-30% as opposed to activity-wise 30-40% shown here, as a result of factoring in modeling effort.

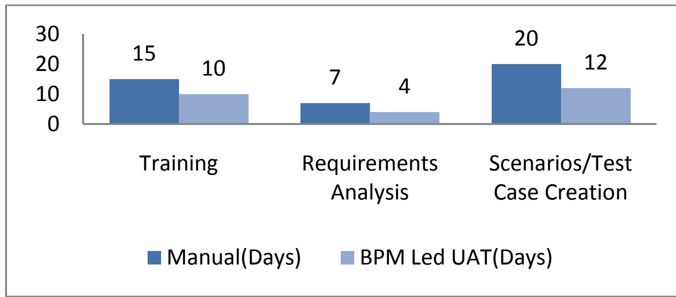


Fig. 5. Activity wise productivity comparison of Manual vs. MBT approach

The resultant test cases being detailed to keystroke level enables quick execution. Creation and maintenance of such detailed tests manually would be extremely inefficient. Breaking down maps into multiple levels has helped in gaining detailed tests and better test coverage.

5.3 Scalability

Securities Trading was a large and complex process with 67 sub-processes. The total number of possible paths was 17 million as seen in the table. The toolset encountered a scalability issue and ran out of memory in trying to apply selection on the entire path set.

To address this problem, the graph was divided into sub-graphs and selection applied on sub-graphs. Selected paths from the first sub-graph were connected to paths from the next sub-graph to form a bigger sub-graph on which selection was applied. This was iteratively done until all sub-graphs covered, yielding the minimal set of 33 paths.

5.4 Quality

The quality problems stated in Section 2 are addressed by MBT as discussed here.

Lack of Optimized Test Coverage. In the existing UAT process, Test scope had to be determined by the Test Lead who had to ensure coverage as well as efficiency. Achieving this balance and designing tests keeping business rules in mind was an effort-intensive process. This effort had to be put in for every change to the test suite as well. Automated test selection allowed the Test lead to focus on just specifying test scope, while the actual test creation and optimization is automated, giving productivity and accuracy at no extra cost once models were in place. Simple Minimal selection provided a basic test set with guarantee of element coverage, which could be built upon.

Standardization in Test Design. The approach defines a process for test design that is systematic and repeatable since it uses machine-manipulable models. Validation

against end-user requirements is the goal of testing, which is achieved through generation from end-user models. All teams can now follow the same process and generated test formats as opposed to differing processes and formats earlier.

Unavailability of End-to-End Process Knowledge. Standardization in the modeling approach has enabled digitization of domain as well as process knowledge and made it available to test teams. Impact on upstream and downstream processes can be ascertained by querying the model.

5.5 Planning

Accuracy in Test Planning and Estimation. Test generation from the requirement model for the specified scope gives the exact number of tests to be run, enabling accurate planning as opposed to the Test lead having to estimate manually using his knowledge and experience. Automated assessment of change impact helps estimate the testing effort for changes and plan more accurately.

6 Discussion and Conclusion

Here we discuss our experience in the context of the problems cited in Section 1 that make adoption of MBT difficult. [2] gives a number of evaluation criteria for MBT. We have covered a majority of these in our evaluation.

6.1 Qualitative Findings, Lessons Learnt and Open Issues

Changing to the Modeling Paradigm. Modeling needed test teams to begin visualizing requirements in the form of process maps. Business Analysts were doing this earlier, now entire test teams had to be trained on modeling.

Modeling guidelines had to be formulated to avoid modeling errors that lead to problems in test generation. Model validations were programmed into the toolset to automate them to the extent possible. Training test teams to write business rules in OCL would not have been viable. A user interface had to be developed to simplify specification, however, specifying complex rules is still hard.

Cost of Deployment. Adoption of MBT needs significant lead time before the new testing process can be rolled out. As discussed in Section 4, transition to the new process and a modeling strategy need to be worked out. Procuring a BPM tool, training personnel, creation of the as-is-models, evaluation against existing tests all contribute to initial cost of deployment although the investment is later expected to pay back through productivity gain and easier maintenance.

The UAT organization spent 2 elapsed months to come up with model architecture. For the initial roll-out, 300 people were trained on modeling and spent 6 months creating as-is process maps and generating tests, covering 60% of the total processes in various core areas. They later trained the remaining team. Area-wise summary of MBT deployment is detailed in table below.

Table 8. Area wise MBT pilot statistics

Business Area	Process maps	Scenarios (Post selection)
Corporate Banking	212	1650
Consumer Banking	140	2800
Capital Markets	260	1800
Insurance	180	900

Limitations of Existing Tools and Technologies. There is a need for standardization of model interchange formats across BPM tools. Currently every modeler exports in different format and does not export all of the information in the model, making it difficult for generative tools like ours to support models from multiple tools.

Limitations of Our Approach. Our data generation approach has limitations - it generates data from the valid values provided to it. These need to be populated with real data from project databases for effective testing. It is a hard problem since data integrity between elements will need to be preserved. In the current approach, users had to manually replace some data in the tool generated output before executing the tests.

Scalability of Constraint Solving. Scalability problems in scenario generation were resolved. The extremely large models like Wealth Management, Payments process 3 etc. confirmed the approach scales for very large process models. However, scalability issues crop up in constraint solving when the size of the data model, particularly number of attributes and their value ranges increases substantially. Currently upto 100 attributes and integer ranges of upto 1000 values are handled by our toolset. Number of entities, associations or constraints do not pose a problem. Easier scripting interface, capability to handle complex data types and higher value ranges are needed in model checkers so that they can be more widely used with business applications.

6.2 Conclusion

Although there is an associated cost of deploying MBT, the modeling formalism enables automation of the test design and change impact assessment, bringing accuracy and efficiency to the process. This has the potential to pay back in the long term not only through productivity gain but through improved quality of testing due to a qualitative improvement in the process and a systematic, repeatable process. The user organization in our case, apart from using models to obtain productivity in Test planning, leveraged their investment by using the models for analysis, training, process optimization and transition.

Extending test generation to include creation of scripts for automated execution would give much greater productivity benefit since as discussed in Section 2, test execution is time-consuming. We have implemented this in our approach but its application poses problems due to large variation in user interface controls, platforms and coding styles.

The main difficulty in proliferation of MBT is high learning curve, cost of deployment, lack of data and process knowledge regarding its use e.g. modeling and optimization strategies. Reuse of models would lower the cost of MBT deployment. Emergence of supporting tools and methods and interoperability between tools as described above would make MBT adoption easier. The advantages of MBT make it worthwhile to work on reducing costs through improved methods and putting models to better use so as to make its application in industry viable.

References

1. Kholkar, D., Goenka, N., Gupta, P.: Automating Functional Testing using Business Process Flows. In: 4th ISEC 2nd Workshop on Advances in Model-Based Software Engineering (2011)
2. Sarma, M., Murthy, P.V.R., Jell, S., Ulrich, A.: Model-Based Testing in Industry: A Case Study with Two MBT Tools. In: ACM Proceedings of 28th International Conference on Software Engineering (ICSE), Cape Town, South Africa, May 7-9 (2011)
3. ARIS: http://www.ariscommunity.com/test-designer
4. Smartesting, http://www.smartesting.com/index.php/cms/en/solution/why-smartesting
5. Conformiq, http://www.conformiq.com/cases/
6. Craggs, I., Sardis, M., Heuillard, T.: AGEDIS Case Studies: Model-Based Testing in Industry. In: Proc. 1st Eur. Conf. on Model Driven Software Engineering, pp. 129–132 (2003)
7. Hartman, A.: Adaptation of Model Based Testing to Industry, Agile and Automated Testing Seminar, Tampere University of Technology (2006)
8. Symbolic Analysis Laboratory (SAL) model checker, http://sal.csl.sri.com
9. Combinatorial and Pairwise Testing, http://csrc.nist.gov/groups/SNS/acts/index.html
10. Hartmann, J., Vieira, M., Foster, H., Ruder, A.: A UML Based Approach to System Testing. Innovations in Systems and Software Engineering 1(1), 12–24 (2005)
11. Briand, L.C., Labiche, Y.: A UML-based approach to system testing. Software Syst. Model 1(1), 10–42 (2002)
12. Bakota, T., Beszédes, Á., Gergely, T., Gyalai, M.I., Gyimóthy, T., Füleki, D.: Semi-Automatic Test Case Generation from Business Process Models SPLST 2009 and NW-MODE 2009 (2009)

A Case Study in Evidence-Based DSL Evolution

Jeroen van den Bos[1,2] and Tijs van der Storm[1]

[1] Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
[2] Netherlands Forensic Institute, Den Haag, The Netherlands
jeroen@infuse.org, storm@cwi.nl

Abstract. Domain-specific languages (DSLs) can significantly increase productivity and quality in software construction. However, even DSL programs need to evolve to accomodate changing requirements and circumstances. How can we know if the design of a DSL supports the relevant evolution scenarios on its programs? We present an experimental approach to evaluate the evolutionary capabilities of a DSL and apply it on a DSL for digital forensics, called DERRIC. Our results indicate that the majority of required changes to DERRIC programs are easily expressed. However, some scenarios suggest that the DSL design can be improved to prevent future maintenance problems. Our experimental approach can be considered first steps towards evidence-based DSL evolution.

1 Introduction

Domain-specific languages (DSLs) can increase productivity by trading generality for expressive power [17,5]. Furthermore, DSLs have the potential to improve the practice of software maintenance: routine changes are easily expressed. More substantial changes, however, might require the DSL itself to be changed [4]. How can we find out whether the relevant maintenance scenarios will require routine changes or not?

In this paper we present a test-based experimental approach to answer this question and apply it to a domain-specific language for describing file formats: DERRIC [2]. DERRIC is used in the domain of digital forensics to generate software to analyze, reconstruct, and recover file-based evidence from storage devices. In digital forensics it is common that such file format descriptions need to be changed regularly, either to accomodate new file format versions, or to deal with vendor idiosyncrasies.

As a starting point, we have assembled a large corpus of image files to trigger failing executions of the file recognition code that is generated from DERRIC descriptions. Each failing execution is attempted to be corrected through a modification of the DERRIC code, until all image files are correctly recognized. The required changes are accurately tracked, categorized and rated in terms of complexity. This set of changes provides an empirical baseline to assess whether the design of DERRIC sufficiently facilitates necessary maintenance.

The results show that all of the required changes were expressible in DERRIC; the DSL did not have to be changed to resolve all failures. The majority of harvested changes consists of multiple, inter-dependent modifications. The second most common change consists of a single, simple, local modification. Finally, a minority of changes is more complex. We discuss how the DERRIC DSL may be changed to make these changes

P. Van Gorp, T. Ritter, and L.M. Rose (Eds.): ECMFA 2013, LNCS 7949, pp. 207–219, 2013.
© Springer-Verlag Berlin Heidelberg 2013

```
1  format PNG
2  extension png
3  strings ascii
4  sequence Signature IHDR Chunk* IEND
5
6  structures
7  Signature {
8      marker: 137,80,78,71,13,10,26,10;
9  }
10
11 Chunk {
12   length: lengthOf(chunkdata) size 4;
13   chunktype: type string size 4;
14   chunkdata: size length;
15   crc: checksum(
16          algorithm="crc32-ieee",
17          init="allone",start="lsb",
18          end="invert",store="msbfirst",
19          fields=chunktype+chunkdata)
20       size 4;
21 }
22
23 IHDR = Chunk {
24   chunktype: "IHDR";
25   chunkdata: {
26     width: !0 size 4;
27     height: !0 size 4;
28     bitdepth: 1|2|4|8|16;
29     colourtype: 0|2|3|4|6;
30     compression: 0;
31     filter: 0;
32     interlace: 0|1;
33   }
34 }
35
36 IEND {
37   length: 0 size 4;
38   chunktype: "IEND";
39   crc: 0xAE,0x42,0x60,0x82;
40 }
```

Fig. 1. Simplified PNG in DERRIC

expressed more easily. Thus, the experiment has provided us with empirical data to improve the design of DERRIC.

The contributions of this paper can be summarized as follows:

– We describe and apply an experiment in DSL-based maintenance in the context of DERRIC, and provide a detailed description including its parameters.
– We present empirical results on how the DERRIC DSL supports the maintenance process in the domain of digital forensics.
– We discuss the usefulness of this approach and how it has helped us to both evaluate and improve the design of DERRIC.

These contributions can be considered first steps towards evidence-based DSL evolution.

2 Background

DERRIC is a DSL to describe binary file formats [2]. It is used in digital forensics investigations to construct highly flexible and high performance analysis tools. One example is the construction of file carvers [1], which are used to recover possibly damaged evidence from confiscated storage devices (e.g., hard disks, cameras, mobile phones etc.). DERRIC descriptions are used to generate some of the software components, called *validators*, that check whether a recovered piece of data is a valid file of a certain type.

An example DERRIC description for a simplified version of the PNG file format is shown in Fig. 1. The structure of a file format is declared using the **sequence** keyword.

The sequence consists of a regular expression that specifies the syntax of a file format in terms of basic blocks, called *structures*. In this case, a PNG file starts with a `Signature` block, an `IHDR` block, zero-or-more `Chunks` and finally an `IEND` block.

The contents of each structure is defined in the following **`structures`** section. A structure consists of one or more fields. The contents and size of each field are constrained by expressions. The simplest expression is a constant, that directly specifies the content, and hence length, of a field. This is the case for the `marker` field of the `Signature` structure. Another common type of constraint only restricts the type and/or length of a field. For instance, the `chunktype` field of structure `Chunk` is constrained to be of type `string` and size 4. Constraints may involve arbitrary content analyses. For example, consider the `crc` field. To recognize this field a full checksum analysis following the `crc32-ieee` algorithm should be performed.

3 Observing Corrective Maintenance

To study the maintainability characteristics of DERRIC, we need a way to inspect and evaluate actual maintenance scenarios. In other words: we need to observe how DSL programs are changed. For the purpose of this paper, we focus on *corrective* maintenance [10], which is maintenance in response to observed failures (“bug fixing”).

To realize this, a large corpus of representative and relevant inputs to a DSL program is needed, which allows us to automatically generate failures, which in turn trigger corrective maintenance actions. The approach is similar to *fuzzing* where a program is run on large quantities of invalid, unexpected or even random input data [19]. For maintenance evaluation, however, it is of paramount importance that the data is representative of what would be encountered in practice.

In the case of DERRIC we have assembled a large, representative corpus of image files (JPEG, GIF and PNG) for which DERRIC descriptions are available. The exact nature of these descriptions and the corpus is described in detail in Section 4.

For each file format f, the initial DERRIC D^i_f description is compiled to a validator and subsequently run on the corpus files of type f. This results in an initial set of files for which validation fails[1]. The set of failures is then divided over equivalence classes which are sorted by their size. This allows us to focus on the most urgent problems first. Next, D^i_f is edited to obtain a new version D^{i+1}_f which resolves at least one of the failures in the largest equivalence class. As soon as the set of failures is observed to decrease, D^{i+1}_f is committed to the version control system. Before committing we ensure that the set of correctly validated files (the true positives) strictly increases, as a form of regression test. The process then repeats, now using D^{i+1}_f as a starting point.

After all failures have been resolved, the changes, as stored in the version control, are categorized in *change complexity classes*. A change may thus be interpreted as being more complex than another change. This provides an empirical base line to qualitatively assess to what extent DERRIC supports maintenance of format descriptions.

[1] Technically, both false positives and false negatives are failures. However, since the corpus only contains real files, we cannot detect when a validator would incorrectly validate a file.

4 Experiment

4.1 DSL Programs and Corpus

The three DSL programs that have been used are DERRIC descriptions of JPEG, GIF and PNG. These file formats are well-known, very common and highly relevant to the practice of digital forensics. An impression of the sizes of these descriptions is given in Table 1. From the table it can be inferred that the descriptions are significantly different. Both GIF and PNG have a richer syntactic structure than JPEG. Structure inheritance is heavily used in JPEG and PNG but only once in GIF. Finally, GIF has a lot more fields per structure (58 per 12). Summarizing, we claim that the three file format descriptions cover a wide range of DERRIC's language features, in different ways.

Table 1. Initial DERRIC descriptions

	JPEG	GIF	PNG
Sequence tokens	14	29	30
Structures	15	12	20
Uses of inheritance	10	1	17
Field definitions	32	58	27

Table 2. Initial validator results

	Data Set		Failures	
Format	#	size	#	%
JPEG	930,386	327GB	5,485	0.6%
GIF	36,524	3GB	389	1.1%
PNG	236,398	27GB	5,789	2.4%
Total	1,203,308	357GB	11,663	1.0%

The second important component of the experiment, is a representative corpus. We have developed such a corpus for the evaluation of our earlier work on model-transformation of DERRIC descriptions [3]. This data set contains JPEG, GIF and PNG images found on Wikipedia, downloaded using the latest available static dump list, which dates from 2008[2]. Around 50% of the files on that list were still available and included in the set. An overview of the data set is shown in Table 2. The corpus contains a total of 1,203,410 images, leading to a total size of 357 GB. As the last two columns show, not all images in the data set are recognized by the validators generated from the respective JPEG, GIF and PNG descriptions: between 0.6% and 2.4% of the files in the data set are not recognized using the base descriptions of the respective file formats.

The Wikipedia data set can be considered representative, since the files uploaded to it originate from many different sources (e.g., cameras, editing software, etc.). We have verified this diversity by inspecting the metadata of the files and aggregating the results.

This shows that the set contains files from a large number of different cameras (e.g., Canon, Nikon, etc.) Furthermore, many images have been modified using a multiplicity of tools (e.g., Photoshop, Gimp, etc.) Original computer images such as diagrams and logos have been created using many different tools (e.g., Dot, Paintshop Pro, etc.)

The diversity is depicted graphically in Fig. 2, showing the distribution of files over values of the EXIF *Software* tag present in 28.4% of the images. The most common tool is Photoshop 7.0, used on 3.4% of the corpus; Photoshop CS2 and CS (Windows) are used on 2.3% and 1.8% respectively. ImageReady covers 1.6%. After that the percentages rapidly decrease: no specific version of any application was used in more than 1% of the files. The number of different values is 4,024.

[2] Available at `https://github.com/jvdb/derric-eval`

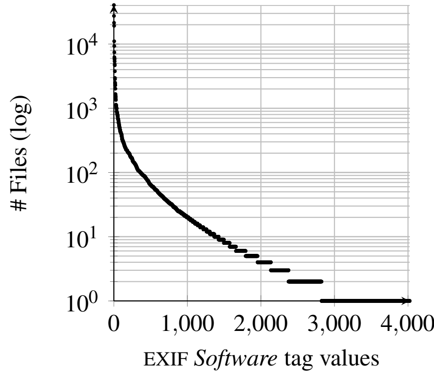


Fig. 2. Distribution of EXIF *Software* tag values over 28.4% of the corpus

4.2 Classifying and Ordering Failures

To improve productivity and handle the most relevant issues first, the set of failures is divided over equivalence classes, according to their *longest normalized recognized prefix*: this is the sequence of DERRIC structures that has been successfully recognized before recognition failed. Classification is repeated after each iteration, because after each change to a description, files might now fail with another prefix.

The prefix is normalized to eliminate the common effect of repeating structures. For instance, if the recognized prefix consists of the structures `A B B C`, then the normalized prefix is `A B+ C`. The plus-sign indicates one-or-more occurrences. As a result, files that failed recognition with prefixes `A B C`, `A B B C`, `A B B B C`, etc. all end up in the same bucket. The equivalence classes thus obtained are then sorted according to size in order to first improve those parts of the description that generate the most failures.

4.3 Evolving the Descriptions

The next step in the experiment is to manually fix the descriptions until all failures have been resolved. After each change, we recorded how many *edits*—additions, modifications and deletions—were needed to reduce the number of failures. An edit captures an atomic delta to a description. Edits can be applied to either the sequence or the list of structures. The semantics of edits is summarized in Table 3.

The simplest edits are addition/removal of a structure to/from the structures section of a DERRIC description, and adding/removing a referenced structure from the sequence expression (cf. Fig. 1). Furthermore, a structure itself can be modified by adding, modifying or removing fields. The sequence can be modified by changing the regular expression without adding or removing a structure reference.

Each change has been tracked in the Git version control system[3] to allow full traceability and reproducability of the results of this paper. In fact, a single change corresponds to a single commit. After each change the DERRIC compiler was rerun with the modified descriptions. The process was repeated until all failures were resolved.

[3] Available at `https://github.com/jvdb/derric-eval`

Table 3. Edit semantics: a DERRIC description's two main sections can be edited in three ways

	Structures	Sequence
Add	Add new structure	Insert structure symbol
Modify	Add, modify, or delete field	Change regular grammar
Delete	Remove structure definition	Remove structure symbol

4.4 Change Complexity Classes

After all failures have been resolved, the resulting set of changes is divided over equivalence classes according to their *change complexity*. Change complexity is intuitively defined in terms of the number of edits in a change, their interrelatedness and how much they are scattered across a source file: more edits, more interrelatedness and more scattering, means higher complexity.

A change consisting of a single edit has very low change complexity. On the other hand, a change involving many logically related edits, scattered over the whole program, has a high change complexity. Simple, low complexity changes leave the structure of the original program mostly intact. At the opposite end, high complexity changes might well create future maintenance problems.

Just like code smells [7] might be indicators of software design problems, in the case of DERRIC, we conjecture, high complexity changes might indicate *language design problems*. For the purpose of our experiment we have identified 3 change complexity classes. Below we briefly describe each class, rated as *Low*, *Medium* or *High*.

- *Single, localized edit (Low)* The ideal situation is where a change requires a single modification of the program. By implication, such a change is always localized. Example: a single edit of the sequence, or the change of a single field in a structure.
- *Multiple, but dependent edits (Medium)* In this case, a change requires multiple, inter-dependent edits. For instance, defining a new structure, then adding a reference to it in the sequence section.
- *Cross-cutting changes (High)* Cross-cutting changes require many (more than two) similar edits scattered across the program. Such changes always involve some form of duplication. This kind of changes is very bad, since they affect the program in a way that is dependent on the size of the program.

The changes, categorized in the change complexity classes, provide an empirical base line to start discussing to what extent DERRIC supports maintenance.

5 Results

The results of the experiment are summarized in Table 4, 5 and 6 for the file formats JPEG, GIF and PNG respectively. The first column of each table identifies the change (i.e. set of edits). In the following, we will identify changes by using a combination of file format name and Id, like so: PNG 11 denotes the eleventh change of the PNG description

in Table 6. Columns 2-5 display how many edits of that particular type were required in order to decrease the number of failures. For instance, change JPEG 1 involved two edits: a structure definition was added, and a reference was added to the sequence expression. Note that deletions are omitted from these tables since they never occurred.

The actual decrease in failures is shown in the “Errors Resolved” column. Finally, the last column shows how a change was categorized with respect to change complexity. Revisiting change JPEG 1 we see that it is ranked as *Medium*, which means that the change contains multiple, dependent edits. Hence we can conclude that the reference inserted into the sequence expression has to be a reference to the newly added structure.

Table 4. Modifications to the JPEG description

	Structure		Sequence		Errors	
Id	Add	Mod	Add	Mod	Resolved	CC
1	1		1		520	*Medium*
2			1		284	*Low*
3	1		1		245	*Medium*
4	1		1		821	*Medium*
5				1	3395	*Low*
6				1	138	*Low*
7	1		2		46	*High*
8	1	4	21		26	*High*
9	1		4		5	*High*
10	1		19		3	*High*
11	1		2		2	*High*

Table 6. Modifications to the PNG description

	Structure		Sequence		Errors	
Id	Add	Mod	Add	Mod	Resolved	CC
1	5		5		3136	*Medium*
2	1		1		1819	*Medium*
3	1		1		332	*Medium*
4	1		1		63	*Medium*
5	1		1		73	*Medium*
6	2		2		112	*Medium*
7	1		1		144	*Medium*
8	1		1		24	*Medium*
9			1		20	*Low*
10			1		18	*Low*
11				1	20	*Low*
12	1		1		10	*Medium*
13	1		1		2	*Medium*
14	1		1		9	*Medium*
15	2		2		2	*Medium*
16			1		3	*Low*
17	1		1		1	*Medium*
18			3		1	*Medium*

Table 5. Modifications to the GIF description

	Structure		Sequence		Errors	
Id	Add	Mod	Add	Mod	Resolved	CC
1		1			9	*Low*
2			1		115	*Low*
3			1		137	*Low*
4		3			36	*Medium*
5			1		39	*Low*
6				1	48	*Low*
7				1	3	*Low*
8			2		2	*Medium*

Table 7. Changes per change complexity class

Level	Name	#
Low	Single localized	13
Medium	Multiple dependent	19
High	Cross-cutting	5
	Total	37

6 Analysis

To summarize the results of our experiment, Table 7 shows the total number of changes per complexity level. The table shows that the majority of changes are easily supported by DERRIC: 13 are simple, localized edits (*Low*), and 19 changes require multiple, dependent edits. The dependency between edits in these changes is a direct consequence of separating sequence from structure definition. In other words: this dependency is anticipated by the design, and hence unavoidable.

Only 5 changes are categorized as cross-cutting (*High*). While in the experiment these changes did not occur very frequently, they still might indicate there is room for improving the design of DERRIC. Moreover, looking at the results for JPEG, we seem to observe a pattern of deterioration. Investigating the actual changes reveals that, indeed, duplication introduced by earlier changes, has a detrimental effect on the required subsequent changes. The fact that cross-cutting changes may amplify each other, is exactly the evolutionary effect we would like to avoid. Three language features could be introduced to DERRIC to eliminate such cross-cutting changes completely:

- Abstraction: a language construct to declare subsequences so that duplicate subsequences can be referred to by name.
- Padding: a construct to automatically interleave certain bytes inbetween structure references in the sequence declaration.
- Precedence: declaring that a particular structure has priority over another one.

Below we motivate these language features based on the results of the experiment.

Abstraction. In JPEG 7, a newly discovered data structure `SOF1` is added to the description. It was discovered that it is part of a sub-sequence of structures that may occur both before and after a mandatory `SOS` structure. As a result, a reference to `SOF1` had to be inserted in two places. The relevant part of the original sequence reads as follows:

```
sequence ...
(DQT DHT DRI SOF0 SOF2 APPX COM)*
SOS
(SOS DQT DHT DRI SOF0 SOF2 APPX COM)*
```

Note that the sequence `DQT DHT DRI SOF0 SOF2 APPX COM` is duplicated. An abstraction construct would allow the description to be refactored as follows:

```
def Seq = DQT DHT DRI SOF0 SOF2 APPX COM;
sequence ... Seq* SOS (SOS Seq)*
```

To accomodate the new `SOF1` structure, only the definition of `Seq` would have to be adapted. Such an abstraction mechanism feature would not only reduce the severity of such changes, it would also clearly communicate to readers of the description that the sequences before and after the `SOS` reference are always the same.

Padding. The JPEG 8 change clearly signals a problem: padding bytes are allowed everywhere in between structures. Every change that modifies the sequence will explicitly make sure that padding is maintained. The duplication introduced by JPEG 7 makes the way this change is expressed even less desirable. A (domain-specific) padding construct allows padding to be expressed in a single place in the configuration section:

```
padding 0xFF
```

The compiler would then weave the generic padding element into the sequence.

Precedence. The cross-cutting change JPEG 10 signals another language feature that could be added to DERRIC. A new structure `COMElanGmk` was identified, which functions as an alternative to the standard `COM` structure. The only difference from `COM` is that `COMElanGmk` redefines the contents of a single field using DERRIC's support for structure inheritance. We would, however, like to also express that `COMElanGmk` has precedence over `COM`: if it is there, consume it, *otherwise* attempt to match `COM`.

The current resolution involves duplicating large parts of the sequence to move the choice between either structure to a higher level. A proper solution would be to extend the set of sequence operators (?, *, etc.) with a new binary operator $<$. The precedence ordering could then be expressed simply as `COMElanGmk < COM`.

7 Discussion

7.1 Lessons Learned

Based on this case study, we can draw a number of conclusions that are generally applicable to the area of DSL development and model-driven development at large. First of all, in order to do evidence-based DSL evolution, the existence of a large, representative corpus is of paramount importance. Given such a corpus, it becomes possible to apply our test-based experimental approach. Our results show that such an experiment indeed provides useful feedback on the design of a DSL.

The corpus of files used in our experiment in essence represents a very large and comprehensive test suite. In other domains, such a test suite has to be designed up front. Nevertheless, the existence of test suites for (legacy) code, could thus be instrumental in deciding whether to adopt a model-driven approach. For instance, in [14] the authors perform a study whether the Mod4J framework is suitable to build web applications following a reference architecture. In this case, the organization had ample experience building such web applications. If (evolving) test suites for a representative sample of non-Mod4J applications exist, they can be run against Mod4J replicas to find out whether Mod4J supports the necessary evolution facilities to fix the failing tests.

Second, to our surprise, the experiment showed that even a simple DSL such as DERRIC requires abstraction facilities in order to mitigate future maintenance. Maybe DSLs and modeling languages are much more like programming languages than we might think. As such, our results provide a cautionary tale, which may be taken into consideration when designing a DSL or modeling language. Furthermore, it might suggest that,

if such a feature is to be avoided, that graph-like, visual concrete syntax is preferrable, since it would allow the direct representation of sharing of sub-structures.

Finally, since our experiment requires the accurate tracking and classification of changes to source models, textual syntax seems to be an advantage. The textual syntax of DERRIC allowed us to use standard `diff` tools to get insight into what was changed inbetween revisions. A visual modeling language would most certainly require custom, domain-specific difference algorithms [20]. Generic difference algorithms (on trees or graphs) would likely contain irrelevant noise, and hence would be hard to interpret.

7.2 Threats to Validity

Even though our classification of changes is informal, we contend that it is sufficiently intuitive. Proficient users of computer languages (domain-specific or general purpose) use similar reasoning to distinguish "good" changes from "bad" changes. Most programmers are familiar with the principles of Don't-Repeat-Yourself (DRY) and Once-and-Only-Once (OAOO). These are precisely the principles that were violated in the cross-cutting changes.

The changes were performed by the first author (the designer of DERRIC) who has ample experience in digital forensics. As such, he could have tended towards the smallest and simplest changes. However, in order to evaluate the way a language supports maintenance it is essential to analyze *optimal* changes; only then can the language aspect be isolated. A subject who is less versed in the domain of digital forensics or DERRIC, would probably have added noise to the results (i.e. unneeded complexity in the changes), and consequently, the results would have been harder to interpret.

As shown in Section 4, we consider the set of image files from Wikipedia a suitable test set for generating failures and harvesting changes. First, the set of images is constructed by thousands of users of Wikipedia, so there is no selection bias. Second, there is a high variability in the origin of the images and how the images were processed in user programs (Fig. 2). Finally, the data set is large enough to generate realistic failures; any of the observed failures could have occurred in practice.

It could be argued that neither JPEG, GIF nor PNG are rich enough to cover the full expressivity or expose the lack thereof of DERRIC. This might be true, however, the DERRIC language is designed precisely for this kind of file formats. In Section 4 we have argued that the DERRIC descriptions of these file formats are sufficiently different to cover the whole language.

7.3 Related Work

Mens et al. [16] define evolution complexity as the computational complexity of a metaprogram that performs a maintenance task, given a "shift" in requirements. Our classification of changes is comparable since we consider small and local edits (fewer "steps") to be easier than multiple, dependent and scattered edits (requiring more steps). Making this relation more precise, however, is an interesting direction for further research. This would involve formalizing each change as a small metaprogram, and then using its computational complexity to rank the changes.

Hills et al. [9] do a similar experiment but use an imaginary virtual machine for "running" maintenance scenarios encoded as simple process expressions. Since the changes and programs investigated in this paper are relatively small, writing them as *actual* metaprograms might be practically feasible. Even more so since DERRIC is implemented using the metaprogramming language RASCAL [11], which is highly suitable for expressing the changes as source-to-source transformations.

The work presented in this paper can be positioned as an experiment in language evaluation. Empirical language evaluation is relatively new since, as pointed out by Markstrum [15], most language features are introduced without evidence to back up its effectiveness or usefulness. In the area of DSL engineering, however, there is work on evaluating the effectiveness of DSLs with respect to program understanding [17], key success factors [8], and maintainability [12]. Our experiment can be seen in this line of work, but focusing on how a DSL *as a language* supports evolution.

Corpus-based language analysis dates at least from the '70s, but is getting more attention recently; see [6] for a comprehensive list of references. A recent study is performed by Lämmel and Pek. [13]. The authors have collected over 3,000 privacy policies expressed in the P3P language in order to discover how the language is used and which features are used most. Morandat et al. [18] gather a corpus of over 1,000 programs written in R to evaluate some of the design choices in its implementation. A difference with respect to our work, however, is that corpus-based language analysis focuses on a corpus of *source files*. Instead, in this paper we used a corpus of *input files* to trigger realistic failures, *not* to analyze the usage of language features, but to analyze how these features fare in the face of evolution.

8 Conclusion

DSLs can greatly increase productivity and quality in software construction. They are designed so that the common maintenance scenarios are easy to execute. Nevertheless, there might be changes that are impossible or hard to express. In this paper we have presented an empirical experiment to discover whether DERRIC, a DSL for describing file formats, supports the relevant corrective maintenance scenarios.

We have run three DERRIC descriptions of image formats on a large and representative set of image files. When file recognition failed, the descriptions were fixed. This process was repeated until no more failures were observed. The required changes, as recorded in version control, were categorized and rated according to their complexity.

Based on the results we have identified to what extent DERRIC supports maintenance of file format descriptions. The results show that most of the changes are easily expressed. However, the results also show there is room for improvement: three features should be added to the language. The most important of those is a mechanism for abstraction to factor out commonality in DERRIC syntax definitions.

Our experimental approach can be applied in the context of other DSLs. The only requirement is a representative corpus of inputs that will trigger realistic failures in the execution of DSL programs and a way to classify and rank the changes required to resolve the failures. By fixing the DSL programs, tracking and ranking the required changes, it becomes possible to observe how seamless (or painful) actual maintenance

would be. We consider the experiment presented in this paper as a first step towards evidence-based DSL evolution.

References

1. Aronson, L., van den Bos, J.: Towards an Engineering Approach to File Carver Construction. In: 2011 IEEE 35th Annual Computer Software and Applications Conference Workshops (COMPSACW), pp. 368–373. IEEE (2011)
2. van den Bos, J., van der Storm, T.: Bringing Domain-Specific Languages to Digital Forensics. In: 33rd International Conference on Software Engineering (ICSE 2011), pp. 671–680. ACM (2011)
3. van den Bos, J., van der Storm, T.: Domain-Specific Optimization in Digital Forensics. In: Hu, Z., de Lara, J. (eds.) ICMT 2012. LNCS, vol. 7307, pp. 121–136. Springer, Heidelberg (2012)
4. van Deursen, A., Klint, P.: Little Languages: Little Maintenance? Journal of Software Maintenance 10(2), 75–92 (1998)
5. van Deursen, A., Klint, P., Visser, J.: Domain-Specific Languages: An Annotated Bibliography. SIGPLAN Notices 35(6), 26–36 (2000)
6. Favre, J.M., Gasevic, D., Lämmel, R., Pek, E.: Empirical Language Analysis in Software Linguistics. In: Malloy, B., Staab, S., van den Brand, M. (eds.) SLE 2010. LNCS, vol. 6563, pp. 316–326. Springer, Heidelberg (2011)
7. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring. Addison-Wesley (1999)
8. Hermans, F., Pinzger, M., van Deursen, A.: Domain-Specific Languages in Practice: A User Study on the Success Factors. In: Schürr, A., Selic, B. (eds.) MODELS 2009. LNCS, vol. 5795, pp. 423–437. Springer, Heidelberg (2009)
9. Hills, M., Klint, P., van der Storm, T., Vinju, J.J.: A Case of Visitor versus Interpreter Pattern. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 228–243. Springer, Heidelberg (2011)
10. ISO/IEC 14764: Software Engineering–Software Life Cycle Processes–Maintenance (2006)
11. Klint, P., van der Storm, T., Vinju, J.: Rascal: A Domain Specific Language for Source Code Analysis and Manipulation. In: Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2009), pp. 168–177. IEEE (2009)
12. Klint, P., van der Storm, T., Vinju, J.J.: On the Impact of DSL Tools on the Maintainability of Language Implementations. In: 10th Workshop on Language Descriptions, Tools and Applications (LDTA 2010). ACM (2010)
13. Lämmel, R., Pek, E.: Vivisection of a Non-Executable, Domain-Specific Language – Understanding (the Usage of) the P3P Language. In: IEEE 18th International Conference on Program Comprehension (ICPC 2010), pp. 104–113. IEEE (2010)
14. Lussenburg, V., van der Storm, T., Vinju, J.J., Warmer, J.: Mod4J: A Qualitative Case Study of Model-Driven Software Development. In: Petriu, D.C., Rouquette, N., Haugen, Ø. (eds.) MODELS 2010, Part II. LNCS, vol. 6395, pp. 346–360. Springer, Heidelberg (2010)
15. Markstrum, S.: Staking Claims: A History of Programming Language Design Claims and Evidence: A Positional Work in Progress. In: 2nd ACM SIGPLAN Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU 2010), p. 7:1–7:5. ACM (2010)
16. Mens, T., Eden, A.H.: On the Evolution Complexity of Design Patterns. Electronic Notes in Theoretical Computer Science 127(3), 147–163 (2005)

17. Mernik, M., Heering, J., Sloane, A.M.: When and How to Develop Domain-Specific Languages. ACM Computing Surveys 37(4), 316–344 (2005)
18. Morandat, F., Hill, B., Osvald, L., Vitek, J.: Evaluating the Design of the R Language - Objects and Functions for Data Analysis. In: Noble, J. (ed.) ECOOP 2012. LNCS, vol. 7313, pp. 104–131. Springer, Heidelberg (2012)
19. Oehlert, P.: Violating Assumptions with Fuzzing. IEEE Security and Privacy 3(2), 58–62 (2005)
20. Xing, Z., Stroulia, E.: UMLDiff: An Algorithm for Object-Oriented Design Differencing. In: 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), pp. 54–65. ACM (2005)

Model Driven Software Development

A Practitioner Takes Stock and Looks into Future

Vinay Kulkarni

Tata Consultancy Services, Pune, India
vinay.vkulkarni@tcs.com

Abstract. We discuss our experience in use of models and model-driven techniques for developing large business applications. Benefits accrued and limitations observed are highlighted. We describe possible means of overcoming some of the limitations and experience thereof. A case for shift in focus of model driven engineering (MDE) community in the context of large enterprises is argued. Though emerging from a specific context, we think, the takeaways from this experience may have a more general appeal for MDE practitioners, tool vendors and researchers.

Keywords: Modeling, meta modeling, separation of concerns, model transformation, software product lines, model driven engineering workbench, model driven enterprise.

1 Introduction

We are in the business of delivering custom business applications for various verticals such as banking, financial services, insurance, retail etc. This paper describes our journey in use of models and model-driven software development techniques since their emergence till date. Chronological sequence of the narration, we think, might help bring out progression of our understanding of the problem and also evolution of home-grown MDE technology. We then discuss capabilities of minimal tooling infrastructure for easy adoption of MDE by industry practice leading to effective use. Then we take a sneak peek at possible future uses of models in the context of large enterprises. The paper concludes with a summary.

2 Model Driven Software Development

Way back in '94, our organization decided to come up with a focused offering in banking and financial services space. As system requirements for the same functional intent such as retail banking, payments, securities trading etc., vary from one financial institution to another, it was felt that developing a shrink-wrapped product wouldn't do. Instead, it was felt that developing a set of functionality components having high internal cohesion and low external coupling would be more pragmatic [1]. A relevant

P. Van Gorp, T. Ritter, and L.M. Rose (Eds.): ECMFA 2013, LNCS 7949, pp. 220–235, 2013.
© Springer-Verlag Berlin Heidelberg 2013

subset of predefined components would be suitably modified to deliver the desired business application. The jump start courtesy of components would shorten time to market, it was felt. The offering was to have object-oriented nature in line with market expectations. It was also felt that the offering should be so designed and architected as to keep pace with technology advance.

2.1 Ground Reality

Experience in delivering business-critical software systems had led to a small team of technical architects having expertise in distributed architecture and relational databases. Though C++ had emerged as dominant object oriented programming language, programmers found its complexity daunting. Also, there was no proven method available for developing industry-strength OO applications back in early '90s. As a result, the immediate principal challenge facing management was *how to quickly make the large team of fresh developers productive?*

2.2 Eliminating Accidental Complexity

Business applications typically conform to a layered architecture wherein each layer encapsulates a set of concerns and interfaces with adjoining architectural layers using a well-defined protocol. For instance, *Graphical user interface layer* deals with which widget to use for displaying which data and what the layout of the screen should be; *Business process layer* deals with ordering of profess steps/tasks and who should be performing which task and when; *Application services layer* deals with which functionality to be exposed to the external world; and *Database access layer* deals with concerns such as how to construct an hierarchical object from flat tables and vice versa. Typically, the architectural layers are wired together by middleware infrastructure that support message passing in a variety of architectures such as synchronous, asynchronous, publish-subscribe etc. As a result, developing a distributed application demands wide-ranging expertise in distributed architectures and technology platforms which is typically in short supply. Large size of application further exacerbates the problem.

We addressed this problem through specification-driven code generation [2]. Keeping the layered nature of application in mind, we came up with domain specific languages (DSL), one for each layer, for specifying the application at a higher level of abstraction. Each DSL was a view over a Unified DSL which enabled specification of constraints spanning across two DSLs. For instance, a button on a screen must map to a service in the application services layer; a screen must have necessary data widgets so that input parameters of the service being invoked can be populated etc. By keeping the DSLs free from all technology related concerns, we helped developers focus on specifying just the business functionality. A DSL processor encoded appropriate technology related details while transforming a concern specification to the desired implementation [3]. We implemented these DSL transformers using standard compiler-compiler techniques [4]. Fig 1 describes the model-driven code generation approach pictorially.

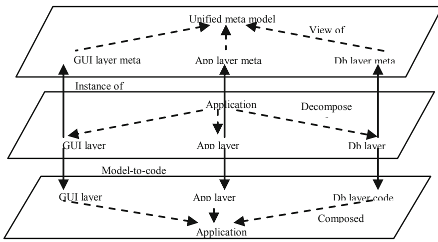


Fig. 1. Generating application from models

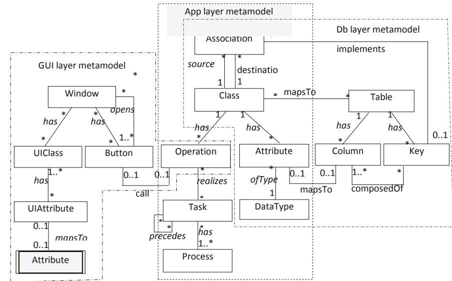


Fig. 2. A subset of the unified metamodel

We used UML [5] class diagrams to capture business entities and their relationships, UML use-case diagrams to describe business scenarios, and UML activity diagrams to describe process flows. We extended UML class models with additional properties and associations in order to capture architectural information, such as classes that make up a message, classes that need to be persisted in a database, classes that need to be displayed on a GUI screen, methods that need to be deployed as services having transactional behavior, class attributes that are mandatory etc. We designed a model-aware high-level language (Q++) to specify the business logic. Q++ treats the extended UML class models as its type system, provides constructs for navigating model associations, and allows for declarative specification of errors and exceptions. Also, the language abstracts out details such as memory management and exception handling strategy. Model-aware nature of Q++ guarantees that business logic specifications will always be consistent with the models.

We extended UML to specify the presentation layer in terms of special abstractions, namely windows and windowtypes. A windowtype specifies a pattern such as a form screen, a list screen, and so on. A window is an instance of a windowtype. It specifies which data elements of which business entity to display using which controls and which buttons should invoke which business services and/or open which windows. Our presentation layer model was independent of the implementation platform except for the event code.

We extended UML to specify relational database schemas and complex database accesses. Object-relational mapping was realized by associating object model elements to the schema model elements, i.e. class to table, attribute to column, association to foreign key etc. We defined a Query abstraction to provide an object façade over SQL queries with an interface to provide inputs to, and retrieve results from the query. We used a slightly modified SQL syntax in order to bind input/output parameters.

We came up with a unified meta model to specify the above mentioned models and their relationships. Fig. 2 highlights the associations spanning these models. These associations help keep the three specifications consistent with respect to each other and thus ensure that the generated platform-specific implementations are also consistent [2]. For example, the association Button.call.Operation can be used to check if a window is capable of supplying all the input parameters required by an operation being called from the window.

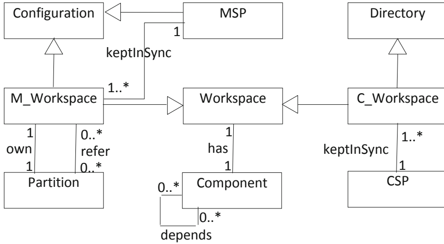


Fig. 3. Metamodel for workspaces

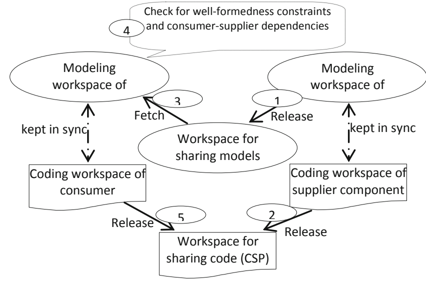


Fig. 4. Synchronizing components

Similarly, we were also able to model other facets such as batch functionality, reports etc. From these various models and high level specifications, we were able to generate a complete application implementation.

Shifting the focus of software development from code to higher level of abstraction led to many advantages:

- Effective separation of concerns in specification provided a good handle on reducing the inherent complexity. DSLs being closer to the problem domain enabled functional experts to play a more significant and operational role in software development life cycle (SDLC). Higher level of abstraction also meant reduced size of the specification and hence reduced time for creating it.
- Greater structure courtesy of models led to application specifications being more amenable for rigorous analysis. As a result, certain kind of errors got caught early in SDLC and certain kind of errors were eliminated altogether. For instance, error of not mapping a screen button to an application service would otherwise get caught at the time of integrating independently developed screens with independently developed application services. In model-driven approach, this error gets caught at model validation stage itself before either the screen or the service is implemented.
- Model-driven development also helped in uniform application-wide implementation of policies such as date data type should always be displayed using, say, dateWidget control and for computation purposes be treated as a string conforming to “mmddyyyy:hh:mm:ss” format. In code-centric development, one has to rely on manual code inspection for enforcing such policies – an error prone and time- and effort-intensive endeavour. On the other hand, in model-driven approach such policies can be enforced either at model validation time or at model creation time itself.
- Code generators helped application-wide uniform implementation of key design decisions. For instance, query-intensive nature (as opposed to update-intensive) of application demands object-relational mapping strategy whereby the table corresponding to the derived class must have columns corresponding to the attributes of its base class all the way up to the root of the class

hierarchy. This schema definition then dictates implementation of create(), get(), modify() and delete() methods which can be automatically generated from the class model. A change in design decision, say, switching over to update-intensive behaviour only needs transformation of the same database layer model using a different model transformer. On the other hand, correct and consistent implementation of this change would be a huge challenge for code-centric approach.

- We extended class and process models to capture testcase and testdata specifications for unit and system testing [6]. This helped us generate testdata with assurance of path coverage. Automation of test execution speeded up application testing process.
- Keeping application specification totally devoid of technology concerns, we could deliver the same specification into multiple technology platforms. This was possible largely because the target technology platforms had more or less similar capabilities. For instance, we could easily switch across databases (Oracle, DB2, Sqlserver), programming languages (C++, Java, C#), middleware (Tuxedo, CICS, Encina, Websphere), presentation managers (ASP, JSP, winforms) and operating system (Unix, Open MVS, Windows).

However, the shift to model driven development also raised some unique problems.

- Though application was specified at a higher level of abstraction, debugging remains at code level. As a result, one needs to carry in mind a map from specification to implementation for effective debugging. Higher the level of abstraction and the number of concerns abstracted, the more complex is the map and hence more difficult is the debugging.
- As modelling is not covered by majority of academic institutes as a part of their curricula, model-driven development had a steep learning curve for fresh developers who constituted a large portion of the project team.
- Modelling and model-based code generation tools needed to follow certain usage discipline. For instance, different concerns of the application being modelled independently needed to be validated for well-formedness and consistency – both within a concern and across the concerns – as only valid models lead to generation of correct implementation in code. To cope with the large size, different concerns had to be modelled separately and synchronized on a need basis as concerns can be interrelated. This was a new way of software development for the project team. Just the availability of tools was not enough. Lack of a method for developing large applications using model-driven techniques proved to be the most significant hurdle in executing project in a coordinated manner.
- Onsite-offshore development model added a new dimension to the problem of coordinated project execution. Application specifications developed at different geographical locations needed to be synchronized from time to time.

With models and high level languages being the primary SDLC artefact, they needed to be amenable for versioning, configuration management, diff-n-merge etc.

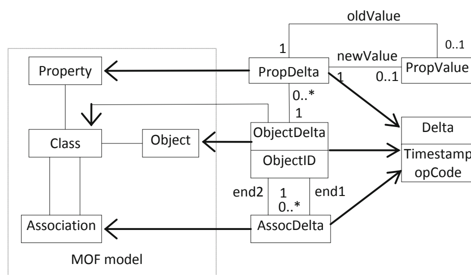


Fig. 5. Delta metamodel

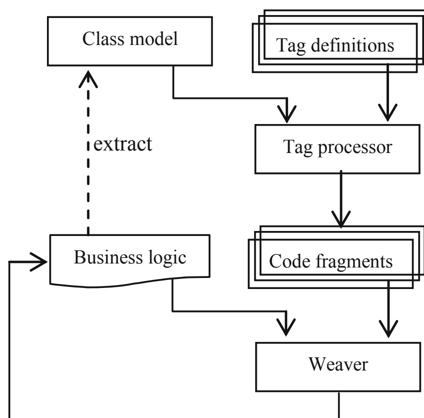


Fig. 6. Metadata-driven aspect-oriented

2.3 Achieving Scale-Up

Component Abstraction: Large business applications needed large development teams. Ensuring coordinated development with large teams became a big issue. Partitioning development effort along functional modules having high internal cohesion and low external coupling seemed intuitive. There was a need to make dependencies between these modules explicit so that independent development was possible.

We introduced a *component* abstraction as a unit of development to manage these dependencies at both model and code level. A component specifies its interface in terms of model elements such as *Classes*, *Operations and Queries*. The consumer-supplier relationship between components is explicitly modelled through *depends* association between the components. A component can only use the model elements specified in the interface of the components it depends upon. As Q++ is model-aware, these model-level dependencies can be honoured automatically in code as well. A component has two associated workspaces, a model workspace and a code workspace as shown in Fig. 3. The model workspace is a configuration comprising of own Partition and the Partitions of the components it depends on. In a component workspace, one is only allowed to change the contents of own Partition. As workspaces provide change isolation, a consumer component is not immediately affected by the changes introduced in its supplier components. A special workspace, configuration MSP (i.e. pool for sharing of models between components) of Fig. 3, is provided for exchanging models between components. A (supplier) component releases its model to this special workspace for sharing, from where its consumer components pick it up as shown in Fig. 4. Model well-formedness constraints and consumer-supplier dependencies are then automatically checked in the consumer component workspace. A similar workspace, directory CSP (i.e. pool for sharing of code between components) of Fig. 3, is provided for sharing code between components. Components are allowed to share code only after they share their models. Model-awareness of Q++ ensures consistency across consumer-supplier components at code-level as well. The process is realized

through a set of roles, each responsible for performing a set of well-defined tasks on a component in a workspace. A role essentially identifies a set of logically coherent process steps. For instance, all modelling related tasks are grouped into the modeller role, all coding related tasks are grouped in the programmer role, all workspace synchronization related tasks are grouped in the manager role, all setup related tasks are grouped in the administrator role etc.

Explicit modelling of interfaces and dependencies provided better control over integration of components being developed independently and in parallel. This enhanced structure was used to compute change impact leading to significantly reduced testing effort. An extension of synchronization protocol of Fig 4 sufficed for multi-site development also.

Change-Driven Development: The performance of various model processing operations such as model validation, diff/merge, model transformation and code generation would deteriorate with increasing model sizes. This in turn would affect turn-around times for change management. Ideally, these operations should only consume time that is proportional to the size of the change and remain unaffected by the total size of the model. We devised a pattern-based approach for implementing incremental execution of common model-driven development process tasks such as model comparison and merging, model validation, and model transformation. We also developed a metamodel for recording changes and integrated it into the model repository for efficient change processing.

Models are well-structured graphs, and many model processing operations can be formulated in terms of graphs. Also, most models are processed in clusters of related elements; for instance, a class, its attributes and operations are usually processed together. Each such cluster has a primary element that identifies the cluster; for instance 'Class' in the above example. We used metamodel patterns to specify such clusters with the root identifying the primary element of the cluster. We then specified the model processing operations in terms of these patterns. For example, when we want to compare class models of two UML models, we want the comparison to be conducted on clusters of model elements centred around class objects; in pattern model terms we want to treat class as the primary object, with its attributes, operations and associations making up the rest of the connected elements of the cluster. In execution terms, a diff operation can be seen as being invoked repeatedly for each matching root element of the pattern from both the source and target models. Fig. 5 shows the metamodel for recording model changes that occur in a model repository.

A Delta is a record of a single change; it has a timestamp property that records the time of the change and an opCode property that records the operation causing the change, namely, one of ADD/MODIFY/DELETE. We developed a separate meta model, as shown in Fig. 5, so that changes to application models can also be captured in a model form. ObjectDelta records changes to objects; PropDelta records changes to properties; and AssocDelta records changes to associations. ObjectDelta has an association to Object to identify the object that has changed; it also stores the ID of the object (ID is required because that is the only way to identify an object that has been

deleted from the repository). PropDelta has an association to Property to identify the property that has changed, and records two values – new and old (if any). AssociationDelta has an association to Association to identify the association that has changed, and two links to ObjectDelta corresponding to the two end objects. The associations between ObjectDelta, PropDelta and AssocDelta mirror the associations between Class, Property and Association in the meta meta model, and thus record the same structure. We devised an algorithm that, given a model pattern, computes the impacted root objects from a given model and its delta model for the set of changes recorded in a given time period.

Thus, we could identify which root objects in the model have changed in a given change cycle and apply the necessary model processing operations only on these root objects. This resulted in minimal code generation for the model changes. We used 'make' utility, which is time-stamp sensitive, to make the subsequent compilation → build → test → deploy steps also incremental.

2.4 Towards Product-Lines

In our experience, no two solutions even for the same business intent such as straight-through-processing of trade orders, back-office automation of a bank, automation of insurance policies administration etc., were identical. Though there existed a significant overlap across functional requirements for a given business intent, the variations were manifold too. Moreover, management expected delivery of subsequent solutions for the same business intent to be significantly faster, better and cheaper. We witnessed that business applications tend to vary along three dimensions:

- Functionality dimension which can be further divided into Business rules and Business logic sub-dimensions
- Business process dimension which can be further divided into Process tasks, Organizational policies, and Organizational structure sub-dimensions
- Solution architecture dimension which can be further divided into Design decisions, Technology platform, and Implementation architecture sub-dimensions

With code generators encoding the choices corresponding to the Solution architecture in transforming application specification into an implementation, we were forced to implement the code generators afresh for every project as no two projects had the same solution architecture even for the same business intent. We observed an interplay between the set of choices wherein a choice along a dimension eliminates (or forces) a set of choices along other dimensions. For instance, choice of 'rural India' geography for a banking system forced 'hosted services platform' choice of Implementation architecture; Choice of Java programming language and Oracle database as persistent store forced 'Object relational mapping' choice for design strategy etc. We witnessed that a choice along a dimension can impact multiple program locations (i.e. scattering) and choices along a set of dimensions can impact the same program location (i.e. tangling). For instance, choices for a set of strategies such as concurrency of a database table row

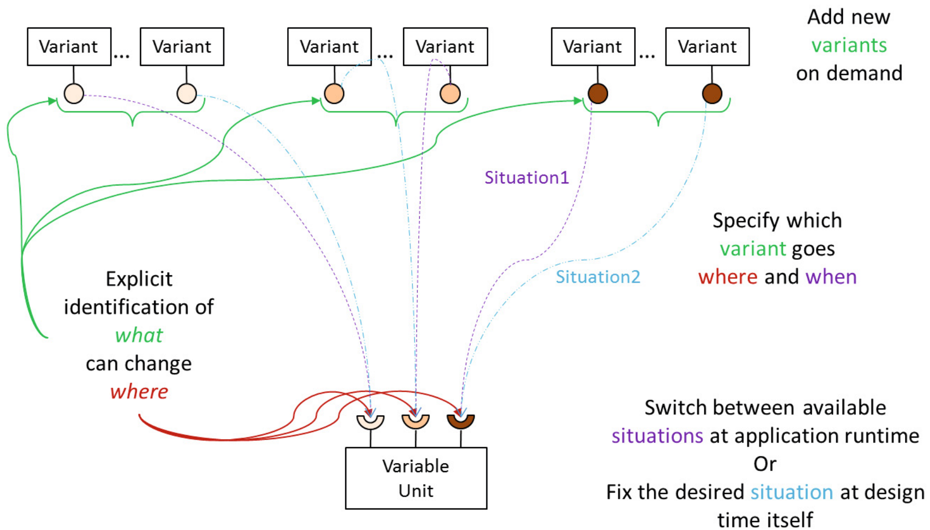


Fig. 7. Architecting for ease of configuration and extension

(corresponding to a persistent object), object-relational mapping, and preserving audit trail in a database table all impact the create() method implementation for a persistent class. And object-relational mapping strategy impacts definitions of all persistent classes in the hierarchy.

These observations led us to visualize model-based code generation system as a set of composable Variable Units each having a set of well-defined Variation Points (VPs) as shown in Fig. 6. The variation points of a variable unit denote the places *where* changes are expected to occur thus reflecting current level of understanding of the domain. A Variation (V) denotes *what* can change at a variation point so as to cater to a specific *Situation*. A *situation* helps to describe the context, i.e. *when* a specific change can occur. Addition of a new variation enriches *system configurability* i.e. ability to address more situations [7]. Also, the variation being added can have variation points of its own thus introducing new paths for extension and configuration. Thus, the system begins to take the shape of a product line wherein a member corresponds to a set of variable units and variations such that no variation point is left unbound, and the variations are *consistent* among themselves.

We came up with a modelling abstraction (*building block*) for specifying the architecture patterns of Fig 6 [8]. A building block encapsulates contribution of a given choice along Design strategy / Architecture / Technology platform sub-dimension to the eventual implementation. Thus, supporting new solution architecture is either novel composition of existing building blocks or addition of a new building block. This step towards model-based code generators product line led to several advantages:

- Model-based generation of model-based code generators reduced the time and effort required for maintaining the code generators.
- Building block abstraction facilitated reuse thus resulting in reduction in size of model-based code generator team.

- Separation of technology platform and MDE concerns led to specialization within code generator development team.
- Boot-strapping gave us confidence about functional completeness of our home-grown MDE infrastructure.

We are in the final stage of moving towards business application product line [10]. The core MDE infrastructure comprising of metamodels, model processors and a method for operationalizing a product line is in place. The central idea is vetted by implementing a model-based code generators product line. The product line idea has been proven in laboratory setting by implementing a near real-life example from Banking domain. We are about to start with a real-life product-line implementation through the restructuring and refactoring of a related set of existing purpose-specific solutions. We will be able to comment on robustness and usability of our MDE infrastructure only after completion of this exercise.

Early experience shows that models, through better separation of multi-dimensional concerns, seem to provide a better handle for implementing business application product lines. Separation of solution architecture from business functionality concerns enables business domain experts to focus solely on specifying the variations in business functionality and technology architects to focus solely on specifying the variations in technology platform, design strategies and implementation architecture. Ability to resolve the specified variations either at code generation time or at application run time leads to increased flexibility. Model-based generation of model-based generators leads to agile product line evolution process. The Variation Points also double up as extension points for introduction of as-yet-unforeseen changes – a reality for industry practice. Though early signs are encouraging, several significant issues remain to be addressed:

- Multi-level resolution of variability i.e. at class level, component level, application level etc., seems to suffice but is posing usability challenge even for the small laboratory case-study we implemented. In the least, more intuitive GUI seems a must for resolving variability.
- Business-critical applications need to evolve through extension and mutation. Our variability metamodel is adequate for addressing extension only i.e. add as-yet-unseen variant part or add as-yet-uncalled-for common part. But, a mutation may lead to fusion of existing variation points and commensurate fusion of a set of variant parts etc. Intuitive refactoring support is essential.
- Maintenance / evolution effort far exceeds the development effort for a successful business application [10]. Precise computation of impact of a change and optimal testing (i.e. what to test when) is a must.
- Effective management of a product line demands coordination of multiple stakeholders such as Domain experts, Solution architects, Developers, Testers, Product line managers etc., across the various SDLC phases. Should there be a feature model for every stakeholder? But a stakeholder might be interested in a set of [sub] dimensions leading to overlap of feature models and feature dependency. It calls for a method (and the relevant tooling) to help what to do when and by whom. There is a need to build further on the proposed multi-level resolution model and the staged resolution approach described in [11].

Definition of a new mutual fund offering or an insurance policy or a financial product varies from the existing ones in a well-defined manner even though the variations need to be introduced at many places. A declarative mechanism aided by suitable [de]composition architecture is missing.

2.5 Code Is the Model

Though the model-driven software development approach has delivered on the promises of improved productivity, better quality and platform independence, especially in development of large enterprise applications, majority of the small to medium sized projects found it difficult to adopt for several reasons. Steep learning curve for modelling meant a considerable chunk of project execution time was spent in creating application specifications. Insistence on models being the primary SDLC artefacts further exacerbated this problem. Project teams wanted bi-directional synchronization between model and the generated code for quick turnaround of changes introduced. An application generated from models is best maintained using the generators. This didn't augur well with many customers who were sensitive to technology and vendor lock-in related risks. Also any change in the solution architecture of the application to be delivered necessitated modifying the code generators which demanded expertise in MDE technology – a skill in excessive short supply. As a result, small and medium sized development projects found model-driven software development approach too heavy and the associated development process too restrictive.

To overcome these limitations, we came up with a metadata-driven aspect-oriented approach for developing J2EE applications (and later extended it for .Net applications). The key idea is to implement just the business logic (i.e. Functionality dimension mentioned earlier) using a reflexive and extensible programming language like Java / C# and annotate it with *tags* [12]. A tag encapsulates choice along one of the many dimensions of Solution architecture such that the desired solution architecture can be viewed as a hierarchical composition of tags. Annotated business logic is parsed to create an internal representation in the form of class model and its metadata annotations. The class model is transformed to generate skeleton or partial class definition. The tag hierarchy is transformed to generate a set of code fragments, each corresponding to the choice along one of the many solution architecture dimensions, and a specification for composing these code fragments. We used AspectJ [13] like syntax for specifying code composition and implemented a tree-transformation based composer which turned out to be sufficient for our needs. Fig 7 gives a pictorial description of this approach.

The approach was supported through an open extensible Eclipse-based toolset resulting in a development process that turned out to be flexible and lightweight as compared to model-driven development process. Separation of architect role (to specify tags) from developer role (to specify business logic and use pre-defined tags) led to effective utilization of expertise. The composable nature of building blocks enabled reuse even across projects. Template based code generation strategy enabled easy adherence to customer-specific standards, guidelines, best practices etc. The composable nature of building blocks, the ability to define purpose-specific metamodels and

plug-in architecture due to Eclipse resulted in easy customizability, extensibility and improved maintainability of the toolset. The approach facilitated creation of a repository of reusable artefacts. A library of reusable, easy-to-adapt solution accelerators enabled even inexperienced teams to deliver high quality code on schedule. Low or no learning curve, adherence to industry standards and dependence on freeware further accelerated acceptance of the approach within developer community.

Class model centricity turned out to be a severe limitation of this approach. Other concerns such as business process, graphical user interface etc could not be addressed easily. In effect, it amounted to coming up with new join point models and developing suitable implementation machinery for every join point model which is a significant effort. Model-driven approach turned out better when multiple DSLs were needed for implementing the required solution. Therefore, we developed a mechanism that enabled interoperation between model-driven and metadata-driven approaches. The most observed use of this interoperability bridge was to graduate from metadata-driven approach to model-driven approach. Although possible, we didn't observe reverting back to metadata-driven approach after having settled into model-driven approach.

3 Tools for Model Driven Development

Model-driven software development has been around since mid-90s. Launch of OMG's MDA [14] in 2000 generated widespread interest in model-driven development. Today it can justifiably be said that model-driven development has proved beneficial in certain niche domains if not all. There is ample evidence of models being used in many ways viz., as pictures, as documentation aids, as jump-start SDLC artefacts, as primary SDLC artefacts etc [15, 16]. Many tools exist that provide automation support at various levels of sophistication for model-driven development. The majority of these tools are highly effective when used in a shrink-wrapped manner but tend to lose effectiveness rapidly whenever the tool needs to be customized or extended. This is a serious drawback.

The demand for intuitiveness on models dictate they be domain-specific. Since there can be infinitely many domains with each domain possibly ever-expanding, it is impossible to think of a universal modelling language that can effectively cater to them all. Furthermore, models are purposive and hence it is impossible to conceive a single modelling language that can cater to all possible purposes. Therefore, multiplicity of modelling languages is a reality. Separation of concerns principle makes the need for a cluster of related modelling languages (one for each concern in a domain) and a mechanism to relate the separately modelled concerns (say to compose a unified model) apparent. The need to relate otherwise separate models demands an ability to express one model in terms of the other. Thus emerges the need for a common language capable of defining all possible modelling languages of interest. There are multiple stakeholders for a model each possibly having a limited view being presented in the form a suitable diagramming notation. From the above discussion, it follows there could be as many diagramming notations as there are modelling languages. And thus emerges the need for a language to define all possible visualizations of a model.

For models to be used as primary SDLC artefacts, there needs to be an execution engine for the models – say an interpreter or a transformer to (say) text format that is executable e.g. a programming language. Plus, separation of concerns leading to a cluster of relatable models indicates the need for transforming one model into another and another and so on. Therefore, to comprehensively address the needs of model-driven software development the MDD tool needs to support:

- A language to define all possible modelling languages
- A language to define all possible visualizations of a model
- A language to specify transformation of one model into another
- A language to specify transformation of a model into text artefacts

As a result, software development gets transformed into a language engineering endeavour wherein the focus is on defining the most intuitive and expressive modelling language[s] for a given purpose and the necessary execution machinery. Since there cannot be a bound on the desired purposes, a configurable extensible modelling language engineering workbench seems required for greater adoption of model-driven software development. In a sense, something on the lines of Eclipse but for language engineering is called for [17].

4 What Next

Economic and geo-political uncertainties are putting increasingly greater stress on frugality and agility for enterprises. Large size and increasing connectedness of enterprises is fast leading them to a system of systems which is characterized by high dynamics and absence of a know-all-oracle. Multiple change drivers are resulting in increasingly dynamic operational environment for enterprise IT systems, for instance, along Business dimensions the change drivers are dynamic supply chains, mergers and acquisitions, globalization pressures etc., along Regulatory compliance dimension the change drivers are Sarbanes oxley, HiPAA, Carbon footprint etc., and along Technology dimension the change drivers are Cloud, smartphones, Internet of things etc. At the same time windows of opportunity for introducing a new service/product/offering and/or for adapting to a change are continuously shrinking. Furthermore, business-critical nature of IT systems means the cost of incorrect decision is becoming prohibitively high and there is very little room for later course-correction. Therefore it is important that we look beyond the traditional model-based generative/SPLE based techniques that we have been using in the past and put more emphasis on understanding of the target organizational environment including its business, IT systems, and stakeholder perspectives. In other words, model the whole enterprise. Towards formal and precise enterprise architecture modelling is an important step towards realizing this goal.

Enterprise Architecture (EA) is a technique used in the process of translating business vision and strategy into effective enterprise change by creating, communicating and improving the key requirements, principles and models centred on business and IT that describe the enterprise's future state and enable its evolution [18]. A number of

EA techniques are used by companies looking for business-IT alignment and transformation with desired properties, for instance, Zachman Framework, TOGAF (Open Group Architecture Framework), FEA (Federal Enterprise Architecture), Gartner, and ArchiMate, etc [19].

Applying these EA techniques to an enterprise is a highly person dependent activity with complete reliance on the enterprise architect's knowledge and experience. Furthermore, validation of goals, such as business-IT alignment, is carried out in a blueprint way in current EA techniques [20]. It means that if the enterprise architect feels, based on his knowledge and experience, that an enterprise has been architected according to principles laid out by these EA techniques; then goals such as business-IT alignment have been accomplished *by definition*. An enterprise may also strive for other goals such as adaptability or cost optimality, for which no mechanism is provided by current EA techniques to prove that a property is satisfied across the enterprise. We believe the ability to specify enterprises in terms of high-level machine-manipulable models that are amenable to analysis and simulation is critical.

The more closely enterprise models reflect reality, the more applicable the inferences from simulation and analysis will be. Today, data describing structural and behavioural aspects of an enterprise is available – typically from multiple and possibly overlapping perspectives. We believe it should be possible to make use of this data to automatically arrive at first-cut purpose-specific models. Results from the well-studied field of log analysis seem readily applicable here [21]. These models will typically be further refined (and glued together into a unified model) by the experts. Results of what-if / if-what analysis and simulation of a purpose-specific model can help arrive at appropriate response to the problem under consideration. A model-centric approach will enable problems to be addressed in a pro-active manner long before they manifest. A mapping or a faithful representation from the models-for-analysis ("simulated" enterprise) to enterprise system models ("real" enterprise) is required to translate inferences from analysis and simulation into an action plan for the underlying enterprise systems. The action plan will typically describe a partially ordered list of changes to be introduced in the operating environment, or in software intensive systems, or in the manner the software intensive systems interact with each other and environment, or any combination of these.

5 Summary

We discussed our experience of delivering large business applications using model driven development approach supported by home-grown standards compliant MDD toolset. In our experience, large application development projects benefitted from this approach in terms of technology independence, enhanced productivity and uniformly high code quality. We observed that without a method imparting discipline to their use, the modelling tools and model-based code generators create more problems for the development team. As a consequence, model-driven development approach leads to less agile if not inflexible development process. We have tried incorporating agile manifesto in model-driven development but it is too early to say anything concrete [22].

Steep learning curve and high upfront investment in the form of creating high level specifications were principal deterrents for small and medium sized software development projects adopting model-driven approach. A metadata-driven code-centric generative approach turned out more appropriate. But, this approach turned out to be too limited to be considered for complex software development endeavours. To overcome this limitation, we came up with a lightweight model interpretation based approach [23]. This worked very well as long as non-functional requirements such as throughput, transaction time were not stringent. As these concerns were effectively addressed in model-driven code generative approach, we developed a migration bridge from interpretive to code generative approach. But this bridge was sporadically used and with mixed response which we haven't fully analysed yet.

Managers agreed to the qualitative benefits of model-driven development but inability to translate them in quantitative terms tended to be the biggest stumbling block for adoption of model-driven approach. We have taken a baby step in this regard but lot needs to be done [24].

We think the basic technological pieces for supporting model-driven development are in place. Many tools with a varying degree of sophistication exist. Other important aspects such as usability, learnability, performance need to be improved which in essence is a continuous process. However, full potential of model-driven development cannot be realized in absence of ready-to-use models supported by domain ontologies providing the semantic and reasoning basis. This aspect is poorly addressed at present.

Focus of MDE community has been on developing technologies that address *how to model*. Barring the domain of safety-critical systems, these models are used only for generating a system implementation. Rather, modelling language design/definition is influenced very heavily by its ability to be transformed into an implementation that can be executed on some platform. Modern enterprises face wicked problems most of which are addressed in ad hoc manner. Use of modelling can provide a more scientific and tractable alternative. For which, modelling community needs to shift the focus on analysis and simulation of models. Results from random graphs, probabilistic graphical models, belief propagation and statistics seem applicable here. We believe, it is possible to model at least a small subset of modern complex enterprises so as to demonstrate that model *is* the organization [25].

Acknowledgment. Author would like to acknowledge Sagar Sunkle, Suman Roychoudhury, Sreedhar Reddy and Tony Clark for direct and indirect help leading to this manuscript.

References

1. Clements, P.C.: From subroutines to subsystems: Component-based software development. American Programmer 8, 31–31 (1995)
2. Kulkarni, V., Venkatesh, R., Reddy, S.: Generating Enterprise Applications from Models. In: OOIS Workshops, pp. 270–279 (2002)
3. Kulkarni, V., Reddy, S.: Integrating Aspects with Model Driven Software Development. In: Software Engineering Research and Practice, pp. 186–197 (2003)
4. Aho, A.V., et al.: Compilers: principles, techniques, and tools, vol. 1009. Pearson/Addison Wesley (2007)

5. UML – Unified Modeling Language, http://www.omg.org/spec/UML
6. Sreenivas, A.: Panel discussion: is ISSTA testing research relevant to industrial users? ACM SIGSOFT Software Engineering Notes 27(4) (2002)
7. Parnas, D.L.: Designing software for ease of extension and contraction. In: ICSE, pp. 264–277 (1978)
8. Kulkarni, V., Reddy, S.: An abstraction for reusable MDD components: model-based generation of model-based code generators. In: GPCE, pp. 181–184 (2008)
9. Kulkarni, V., Barat, S., Roychoudhury, S.: Towards Business Application Product Lines. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 285–301. Springer, Heidelberg (2012)
10. Boehm, B.: A Spiral Model of Software Development and Enhancement. ACM SIGSOFT Software Engineering Notes 11(4), 14–24 (1986)
11. Czarnecki, K., Helsen, S.: Staged configuration using feature models. In: Nord, R.L. (ed.) SPLC 2004. LNCS, vol. 3154, pp. 266–283. Springer, Heidelberg (2004)
12. Kulkarni, V.: Metadata-driven aspect-oriented software development. Technical Architects Conference (2004)
13. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An Overview of AspectJ. In: Lindskov Knudsen, J. (ed.) ECOOP 2001. LNCS, vol. 2072, pp. 327–354. Springer, Heidelberg (2001)
14. MDA – Model Driven Architecture, http://www.omg.org/mda
15. Hailpern, B., Tarr, P.: Model-driven development: The good, the bad, and the ugly. IBM Systems Journal 45(3), 451–461 (2006)
16. Hutchinson, J., Rouncefield, M., Whittle, J.: Model-driven engineering practices in industry. In: 2011 33rd International Conference on Software Engineering (ICSE), pp. 633–642. IEEE (2011)
17. Kulkarni, V., Barat, S., Sunkle, S.: Model driven development – where to from here? A practitioner's perspective. In: Advances in Model Based Engineering Workshop of India Software Engineering Conference (2012), http://www.infosys.com/infosyslabs/publications/infoyslabs-briefings/Pages/software-engineering-approaches.aspx
18. IEEE recommended practice for architectural description of software intensive systems, http://standards.ieee.org/findstds/standard/1471-2000.html
19. Sessions, R.: A comparison of the top four enterprise-architecture methodologies, MSDN-Enterprise Architecture Trends (2007), http://msdn.microsoft.com/en-us/library/bb466232.aspx
20. Wagter, R., Proper, H.A(E.), Witte, D.: A practice-based framework for enterprise coherence. In: Proper, E., Gaaloul, K., Harmsen, F., Wrycza, S. (eds.) PRET 2012. LNBIP, vol. 120, pp. 77–95. Springer, Heidelberg (2012)
21. Margara, A., Cugola, G.: Processing flows of information: from data stream to complex event processing. In: DEBS 2011, pp. 359–360 (2011)
22. Kulkarni, V., Barat, S., Ramteerthkar, U.: Early Experience with Agile Methodology in a Model-Driven Approach. In: Whittle, J., Clark, T., Kühne, T. (eds.) MODELS 2011. LNCS, vol. 6981, pp. 578–590. Springer, Heidelberg (2011)
23. Shroff, G., Agarwal, P., Devanbu, P.T.: InstantApps: A WYSIWYG model driven interpreter for web applications. In: ICSE Companion 2009, pp. 417–418 (2009)
24. Sunkle, S., Kulkarni, V.: Cost Estimation for Model-Driven Engineering. In: France, R.B., Kazmeier, J., Breu, R., Atkinson, C. (eds.) MODELS 2012. LNCS, vol. 7590, pp. 659–675. Springer, Heidelberg (2012)
25. Clark, T., Kulkarni, V., France, R., Frank, U., Barn, B.: Towards model driven organization. In: NIER Manuscript Submitted to ICSE 2013 (2013)

Author Index

Abid, Mohamed 101
Ambert, Fabrice 134

Babau, Jean-Philippe 101
Barbier, Franck 37
Bouquet, Fabrice 134
Buezas, Nicolás 179

Cariou, Eric 37
Castelo Branco, Moises 20
Combemale, Benoit 118
Conallen, James 165
Czarnecki, Krzysztof 20

de Lara, Juan 179
De Queiroz Santos, Davi 54
Derrien, Steven 118
Deshpande, Ajay 192
Didonet Del Fabro, Marcos 54

Elaasar, Maged 165

Favre, Cédric 20
France, Robert B. 118

Gogolla, Martin 1
Gray, Jeff 86
Guerra, Esther 179

Kholkar, Deepali 192
Kleiner, Mathias 54
Kulkarni, Vinay 220
Küster, Jochen 20

Lasalle, Jonathan 134
Legeard, Bruno 134
Le Goaer, Olivier 37

Manns, Paul 70
Martín, Javier 179
Mehiaoui, Asma 101
Monforte, Miguel 179
Mori, Fiorella 179
Mraidha, Chokri 101
Mzid, Rania 101

Ogallar, Eva 179

Patzina, Lars 70
Patzina, Sven 70
Pérez, Oscar 179
Peureux, Fabien 134
Pierre, Samson 37
Piper, Thorsten 70

Ritter, Daniel 152

Sánchez Cuadrado, Jesús 179
Shetye, Aditya 192
Störrle, Harald 3
Sun, Wuliang 118
Sun, Yu 86

Tiwari, Harshit 192
Tucci-Piergiovanni, Sara 101

van den Bos, Jeroen 207
van der Storm, Tijs 207
Völzer, Hagen 20

Yelure, Pooja 192