

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

University of Dortmund, Germany

Madhu Sudan

Massachusetts Institute of Technology, MA, USA

Demetri Terzopoulos

University of California, Los Angeles, CA, USA

Doug Tygar

University of California, Berkeley, CA, USA

Gerhard Weikum

Max-Planck Institute of Computer Science, Saarbruecken, Germany

Ina Schieferdecker Alan Hartman (Eds.)

Model Driven Architecture – Foundations and Applications

4th European Conference, ECMDA-FA 2008
Berlin, Germany, June 9-13, 2008
Proceedings

Volume Editors

Ina Schieferdecker
TU Berlin/Fraunhofer FOKUS
Kaiserin-Augusta-Allee 31, 10589 Berlin, Germany
E-mail: ina.schieferdecker@fokus.fraunhofer.de

Alan Hartman
IBM Haifa Research Laboratory
Haifa University Campus, Mt. Carmel, Haifa 31905, Israel
E-mail: hartman@il.ibm.com

Library of Congress Control Number: 2008928260

CR Subject Classification (1998): C.2, D.2, D.3, F.3, C.3, H.4

LNCS Sublibrary: SL 2 – Programming and Software Engineering

ISSN	0302-9743
ISBN-10	3-540-69095-6 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-69095-5 Springer Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.

Springer is a part of Springer Science+Business Media

springer.com

© Springer-Verlag Berlin Heidelberg 2008
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12280377 06/3180 5 4 3 2 1 0

Preface

The fourth edition of the European Conference on Model-Driven Architecture – Foundations and Applications (ECMDA-FA 2008) was dedicated to furthering the state of knowledge and fostering the industrialization of the model-driven architecture (MDA) methodology. MDA is an initiative proposed by the Object Management Group (OMG) for platform-generic software development. It promotes the use of models in the specification, design, analysis, synthesis, deployment, and evolution of complex software systems.

ECMDA-FA 2008 focused on engaging key European and international researchers and practitioners in a dialogue which will result in a stronger, more efficient industry, producing more reliable software on the basis of state-of-the-art research results. ECMDA-FA is a forum for exchanging information, discussing the latest results and arguing about future developments of MDA.

It is a pleasure to be able to introduce the proceedings of ECMDA-FA 2008. ECMDA-FA addresses various MDA areas including model management, executable models, concrete syntaxes, aspects and concerns, validation and testing, model-based systems engineering, model-driven development and service-oriented architectures, and the application of model-driven development.

There are so many people who deserve warm thanks and gratitude. The fruitful collaboration of the Organization, Steering and Program Committee members and the vibrant community led to a successful conference: ECMDA-FA 2008 obtained excellent results in terms of submissions, program size, and attendance.

The Program Committee accepted, with the help of additional reviewers, research papers and industry papers for ECMDA-FA 2008: We received 87 submissions. Of these, a total of 31 were accepted including 21 research papers and 10 industry papers. We thank them for the thorough and high-quality selection process.

The Steering Committee members helped with various issues – we enjoyed continuous and constructive interactions with them. Birgit Benner, the Organization Chair, is to be warmly thanked for her important role in making this conference a well-organized event. In addition, ECMDA-FA 2008 was complemented by a varied set of workshops: for this we would like to thank the Workshop Chair Marc Born.

We would like to thank for the strong and consistent support received from Fraunhofer FOKUS, Technical University Berlin, and the Alfried Krupp von Bohlen und Halbach-Stiftung, without whom ECMDA-FA 2008 would not have been possible. We are also very grateful to the European Association of Software Science and Technology (EASST) and to the Joint Interest Groups on Modelling (JIGMOD) of GI for their trust and support.

The proceedings were produced by Springer LNCS: we were assisted by Anna Kramer and Erika Siebert-Cole, who made everything straightforward for us.

Finally, we would like to thank all the authors who spent valuable time in preparing and submitting papers to ECMDA-FA 2008 and the sponsors of ECMDA-FA 2008.

June 2008

Ina Schieferdecker
Alan Hartman

Organization

ECMDA-FA 2008 was organized by the Competence Center Modeling and Testing of Fraunhofer FOKUS, Berlin, Germany, the Chair on Design and Testing of Communication-Based Systems, Technical University Berlin, and the Model-Driven Engineering Technologies Group at IBM Haifa Research Laboratory, Israel.

Executive Committee

Conference Chair	Ina Schieferdecker (TU Berlin/Fraunhofer FOKUS, Germany)
Program Chair	Alan Hartman (IBM, Israel)
Workshop Chair	Marc Born (IKV++, Germany)
Panel Moderator	Andy Schürr (University of Dortmund, Germany)
Tools and Poster Chair	Philippe Desfray (Softeam, France)
Organizing Chair	Birgit Benner (Fraunhofer FOKUS, Germany)
Publication Chair	Tom Ritter (Fraunhofer FOKUS, Germany)
Webmaster	Andrea Techlin (Fraunhofer FOKUS, Germany)

Steering Committee

Philippe Desfray (Objecteering Software, France)
Alan Hartman (IBM, Israel)
Richard Paige (University of York, UK)
Arend Rensink (University of Twente, The Netherlands)
Andy Schürr (Darmstadt University of Technology, Germany)
Regis Vogel (IHG, France)
Jos Warmer (Ordina, The Netherlands)

Program Committee

Jan Aagedal (Telenor, Norway)
Dave Akehurst (Thales, France)
Uwe Assmann (TU Dresden, Germany)
Terry Bailey (Fundación European Software Institute, Spain)
Mariano Belaunde (France Telecom)
Xavier Blanc (University of Paris VI, France)
Zhen Ru Dai (Philips, Germany)
Miguel de Miguel (Universidad Politecnica de Madrid, Spain)

Jean-Luc Dekeyser (University of Lille 1, France)
Philippe Desfray (Objecteering Software, France)
Jürgen Dingel (Queen's University, Canada)
Gregor Engels (University of Paderborn, Germany)
Joachim Fischer (Humboldt-Universität zu Berlin, Germany)
Jeff Gray (University of Alabama, USA)
Alan Hartman (IBM, Israel)
Frederic Jouault (University of Nantes, France)
Gabor Karsai (Vanderbilt University, USA)
Olaf Kath (IKV++, Germany)
Torsten Klein (Carmeq, Germany)
Ingolf Krüger (UC San Diego, USA)
Zhang Li (Beihang University, China)
Eda Marchetti (CNR, Italy)
Tiziana Margaria (University of Potsdam, Germany)
Parastoo Mohagheghi (SINTEF, Norway)
Richard Paige (University of York, UK)
Christoph Pohl (SAP AG, Germany)
Arend Rensink (University of Twente, The Netherlands)
Laurent Rioux (Thales Group, France)
Tom Ritter (Fraunhofer FOKUS, Germany)
Bernhard Rumpe (University of Braunschweig, Germany)
Ina Schieferdecker (TU Berlin/Fraunhofer FOKUS, Germany)
Doug Schmidt (Vanderbilt University, USA)
Rudolf Schreiner (Objectsecurity, Germany)
Andy Schürr (Darmstadt University of Technology, Germany)
Bran Selic (IBM Canada)
Juha-Pekka Tolvanen (Metacase, Finland)
Andreas Ulrich (Siemens AG, Germany)
Marten Van Sinderen (University of Twente, The Netherlands)
Regis Vogel (IHG, France)
Jos Warmer (Ordina, The Netherlands)
Albert Zündorf (University of Kassel, Germany)

Additional Referees

Martin Assmann	Barry Demchak
Jan-Christopher Bals	Zekai Demirezen
Iovka Boneva	Birgit Demuth
Javier F. Briones	Dolev Dotan
Frank Brüseke	Frederic Doucet
Philippa Conmy	Cédric Dumoulin
Michelle L. Crane	Hajo Eichler
Cesar de Moura Filho	Vina Ermagan
Vegard Dehlen	Anne Etien

Emilia Farcas
Abdoulaye Gamatié
Hans Grönniger
Andrei Kirshin
Holger Krahn
Shiri Kremer-Davidson
Ivan Kurtev
Andreas Leicher
Hongzhi Liang
Michael Meisinger
Massimiliano Menarini
Alix Mougenot
Tor Neple
Michael Piefel
Dick Quartel

Dirk Reiss
Xavier Renault
Harald Rbig
Julia Rubin
Tim Schattkowsky
Boris Shishkov
Juan Pedro Silva
Tom Staijen
Mark Stein
Yu Sun
Robert Tairas
Hendrik Voigt
Steven Völkel
Anjelika Votintseva

Supporting Institutions

Alfried Krupp von Bohlen und Halbach-Stiftung
European Association of Software Science and Technology (EASST)
Joint Interest Groups on Modeling of GI

Sponsoring Institutions

European Project MODELPLEX (MODELLing solution for comPLEX
software systems)
International Business Machines Corp.
Netfective Technology SA
Objecteering Software SA
Testing Technologies IST GmbH

Table of Contents

Research Session

Model Management

The Epsilon Generation Language	1
<i>Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A.C. Polack</i>	
Constructing and Visualizing Transformation Chains	17
<i>Jens von Pilgrim, Bert Vanhooft, Immo Schulz-Gerlach, and Yolande Berbers</i>	
Towards Roundtrip Engineering – A Template-Based Reverse Engineering Approach	33
<i>Manuel Bork, Leif Geiger, Christian Schneider, and Albert Zündorf</i>	
Annotation Framework Validation Using Domain Models	48
<i>Carlos Noguera and Laurence Duchien</i>	

Executable Models

Prototyping Visual Interpreters and Debuggers for Domain-Specific Modelling Languages	63
<i>Daniel A. Sadilek and Guido Wachsmuth</i>	
Data Flow Analysis of UML Action Semantics for Executable Models ...	79
<i>Tabinda Waheed, Muhammad Zohaib Z. Iqbal, and Zafar I. Malik</i>	
From UML Activities to TAAL - Towards Behaviour-Preserving Model Transformations	94
<i>Gregor Engels, Anneke Kleppe, Arend Rensink, Maria Semenyak, Christian Soltenborn, and Heike Wehrheim</i>	
A Practical MDA Approach for Autonomic Profiling and Performance Assessment	110
<i>Fabio Perez Marzullo, Rodrigo Novo Porto, Divany Gomes Lima, Jano Moreira de Souza, and José Roberto Blaschek</i>	
Ladder Metamodeling and PLC Program Validation through Time Petri Nets	121
<i>Darlam Fabio Bender, Benoît Combemale, Xavier Crégut, Jean Marie Farines, Bernard Berthomieu, and François Vernadat</i>	

Array OL Descriptions of Repetitive Structures in VHDL	137
<i>Stephen Wood, David Akehurst, Gareth Howells, and Klaus McDonald-Maier</i>	

Concrete Syntaxes

Textual Modelling Embedded into Graphical Modelling	153
<i>Markus Scheidgen</i>	
Classification of Concrete Textual Syntax Mapping Approaches	169
<i>Thomas Goldschmidt, Steffen Becker, and Axel Uhl</i>	
Metamodel Syntactic Sheets: An Approach for Defining Textual Concrete Syntaxes	185
<i>Javier Espinazo-Pagán, Marcos Menárguez, and Jesús García-Molina</i>	
Graphical Concrete Syntax Rendering with SVG	200
<i>Frédéric Fondement</i>	

Aspects and Concerns

Semantics Preservation of Sequence Diagram Aspects	215
<i>Jon Oldevik and Øystein Haugen</i>	
Generic Reusable Concern Compositions	231
<i>Aram Hovsepian, Stefan Van Baelen, Yolande Berbers, and Wouter Joosen</i>	
Modeling Human Aspects of Business Processes – A View-Based, Model-Driven Approach	246
<i>Ta'ïd Holmes, Huy Tran, Uwe Zdun, and Schahram Dustdar</i>	
A Semantics-Based Aspect Language for Interactions with the Arbitrary Events Symbol	262
<i>Roy Grønmo, Fredrik Sørensen, Birger Møller-Pedersen, and Stein Krogdahl</i>	

Validation and Testing

Model-Driven Platform-Specific Testing through Configurable Simulations	278
<i>Thomas Kuhn and Reinhard Gotzhein</i>	
Testing Metamodels	294
<i>Daniel A. Sadilek and Stephan Weißleder</i>	
A Metamodeling Approach for Reasoning about Requirements	310
<i>Arda Goknil, Ivan Kurtev, and Klaas van den Berg</i>	

Industrial Session

Model-Based Systems Engineering

Model-Driven Security in Practice: An Industrial Experience	326
<i>Manuel Clavel, Viviane da Silva, Christiano Braga, and Marina Egea</i>	
Supporting the UML <i>State Machine Diagrams</i> at Runtime	338
<i>Franck Barbier</i>	
Model-Based Generation of Interlocking Controller Software from Control Tables.....	349
<i>Cédric Chevillat, David Carrington, Paul Strooper, Jörn Guy Süß, and Luke Wildman</i>	
Model-Driven Simulation of a Maritime Surveillance System	361
<i>Martin Monperrus, Fabre Jaozafy, Gabriel Marchalot, Joel Champeau, Brigitte Hoeltzener, and Jean-Marc Jézéquel</i>	

Model-Driven Development and Service-Oriented Architectures

Towards Utilizing Model-Driven Engineering of Composite Applications for Business Performance Analysis	369
<i>Mathias Fritzsche, Wasif Gilani, Christoph Fritzsche, Ivor Spence, Peter Kilpatrick, and John Brown</i>	
From Business Architecture to SOA Realization Using MDD	381
<i>Avivit Bercovici, Fabiana Fournier, and Alan J. Wecker</i>	
Realizing an MDA and SOA Marriage for the Development of Mobile Services.....	393
<i>Mariano Belaunde and Paolo Falcarin</i>	

Surveys on Applying Model-Driven Development

A Survey about the Intent to Use Visual Defect Annotations for Software Models	406
<i>Jörg Rech and Axel Spriestersbach</i>	
MDA-Based Methodologies: An Analytical Survey.....	419
<i>Mohsen Asadi and Raman Ramsin</i>	
Where Is the Proof? – A Review of Experiences from Applying MDE in Industry	432
<i>Parastoo Mohagheghi and Vegard Dehlen</i>	
Author Index	445

The Epsilon Generation Language

Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos,
and Fiona A.C. Polack

Department of Computer Science, University of York, UK
{louis,paige,dkolovos,fiona}@cs.york.ac.uk

Abstract. We present the Epsilon Generation Language (EGL), a model-to-text (M2T) transformation language that is a component in a model management tool chain. The distinctive features of EGL are described, in particular its novel design which inherits a number of language concepts and logical features from a base model navigation and modification language. The value of being able to use a M2T language as part of an extensible model management tool chain is outlined in a case study, and EGL is compared to other M2T languages.

1 Introduction

For Model-Driven Development to be applicable in the large, and to complex systems, mature and powerful *model management* tools and languages must be available. Such tools and languages are beginning to emerge, e.g., model-to-model (M2M) transformation tools such as ATL [8] and VIATRA [19], workflow architectures such as oAW [17], and model-to-text (M2T) transformation tools such as MOFScript [15] and XPand [17].

Whilst there are some mature model management tools, most such tools are stand-alone, or are loosely integrated through their ability to manipulate and manage the same kind of models, for instance via Eclipse EMF. (An exception is oAW, which supports model management workflows). These limitations mean that development of new tools often entails substantial effort, with few opportunities for reuse of language constructs and tools [13]. However, model management tasks have many common requirements (e.g., the need to be able to traverse models), share common concepts (e.g., the ability to query models) and have a common logic. There is substantial value, for developers and users, in integrating model management tools, to share features and facilitate construction of support for new model management tasks. Integrated tools improve our ability to provide rich automated support for model management in the large.

M2T transformation is an important model management task with a number of applications, including model serialisation (enabling model interchange); code and documentation generation; and model visualisation and exploration. In 2005, the OMG [9] recognised the lack of a standardised approach to performing M2T transformation with its M2T language RFP [16]. Various MDD tool vendors have developed M2T languages, including JET [5], XPand and MOFScript. None of

these M2T languages has been built from other model management languages, and none directly exploits existing M2M support – the new languages have been developed either from scratch (e.g., JET and MOFScript), or as a component that can be applied within a modelling workflow (e.g., XPand).

Our approach is to create tools that are components in an *extensible* and *integrated* model management tool chain. This paper introduces the *Epsilon Generation Language (EGL)*, a language for specifying and performing M2T transformations. We describe EGL’s basic and distinctive features, with particular emphasis on the advantages of building EGL as part of an extensible, integrated platform for model management; we illustrate the minimalist derivation needed, from the existing EOL [13] that supports model navigation and modification.

We start with an overview of key concerns for M2T transformation tools. In Section 3, we discuss the features and tool support provided by EGL. We discuss EGL’s unique features, and explain its development from EOL, focusing on how EOL’s design has been reused in EGL. In Section 4, a case study demonstrates the use of EGL to perform model visualisation. In Section 5, we compare EGL to other M2T transformation tools. Finally, in Section 6, we discuss future work.

2 Background

In this section, we briefly outline the key concerns of an effective M2T transformation solution. We also briefly describe the Epsilon model management platform, and its support for building new languages and tools.

2.1 Concerns of Model-to-Text Transformation

There are four key concerns in any M2T transformation solution.

Repeatability. M2T transformations may need to be repeatable, so that changes made to models percolate through to generated text. However, repeated invocation of transformations may need to respect hand-written changes that have been made to generated artefacts [16].

Traceability. After performing a M2T transformation, it should be possible to determine the elements of the source model from which a portion of the text has been produced. Such traceability is particularly valuable when debugging a model or when auditing the development process.

Readability. An M2T solution must maintain readability aspects, such as layout and indentation.

Flexibility. M2T transformations, like M2M, need to be flexible; one approach is to support parameterised transformation definitions [10].

2.2 The Epsilon Platform

Epsilon, the Extensible Platform for Specification of Integrated Languages for mOdel maNagement [11], is a suite of tools and domain-specific languages for model-driven development. Epsilon comprises a number of integrated model management languages, based upon a common infrastructure, for performing tasks such as model merging, model transformation and intermodel consistency checking [12]. Whilst many model management languages are bound to a particular subset of modelling technologies, limiting their applicability [14], Epsilon is metamodel-agnostic – models written in any modelling language can be manipulated by Epsilon’s model management languages. (Epsilon currently supports models implemented using EMF, MOF 1.4, pure XML, or CZT.)

Figure 1 illustrates the various components of Epsilon.

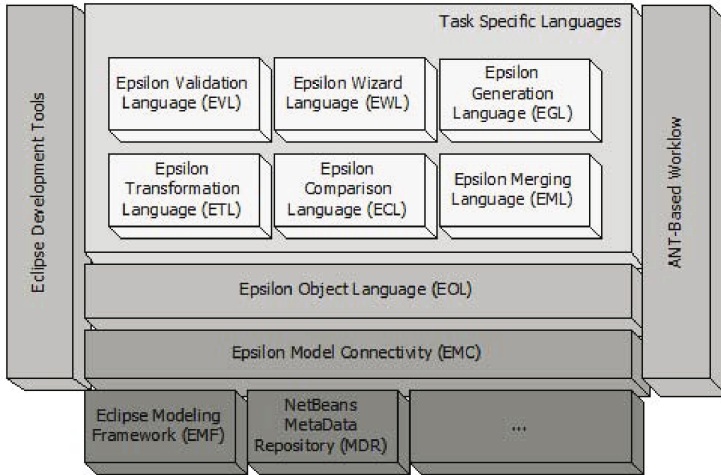


Fig. 1. The architecture of Epsilon

The design of Epsilon promotes reuse when building task-specific model management languages and tools. Each individual Epsilon language (e.g., ETL, ECL, EGL) can be reused wholesale in the production of new languages. Ideally, the developer of a new language only has to design language concepts and logic that do not already exist in Epsilon languages.

EGL follows this principle, and inherits concepts and logic from Epsilon’s base language, EOL, as described in Section 3. First, we outline EOL’s features, particularly as they pertain to EGL.

2.2.1 The Epsilon Object Language. The core of the platform is the Epsilon Object Language (EOL) [13]. EOL’s scope is similar to that of OCL. However, EOL boasts an extended feature set, which includes the ability to update models, conditional and loop statements, statement sequencing, and access to

standard output and error streams. Every Epsilon language re-uses EOL, so improvements to this object language enhance the entire platform.

A recent enhancement to EOL is the provision of constructs for profiling. EOL also allows developers to delegate computationally intensive tasks to extension points, where the task can be authored in Java. This now allows developers using any Epsilon language to monitor and fine-tune performance – in EGL, this allows fine-tuning of M2T transformations.

EOL itself provides the most basic M2T transformation facilities, because every EOL type provides `print` and `println` methods, which append a textual representation of the instance to the default output stream. However, native EOL is insufficient for M2T in the large – transformation specifications littered with explicit `print` statements become unreadable, and EOL alone does not support the sorts of features, specific to M2T transformation, which address the concerns identified in Section 2.1.

3 The Epsilon Generation Language (EGL)

EGL provides a language for M2T in the large. EGL is a model-driven template-based code generator, built atop Epsilon, and re-using all of EOL. In this section, we discuss the design of EGL and its construction from existing Epsilon tools.

3.1 Abstract Syntax

Figure 2 depicts the abstract syntax of EGL’s core functionality.

In common with other template-based code generators, EGL defines *sections*, from which templates may be constructed. Static sections delimit sections whose contents appear verbatim in the generated text. Dynamic sections contain executable code that can be used to control the generated text.

In its dynamic sections, EGL re-uses EOL’s mechanisms for structuring program control flow, performing model inspection and navigation, and defining custom operations. EGL provides an EOL object, `out`, for use within dynamic

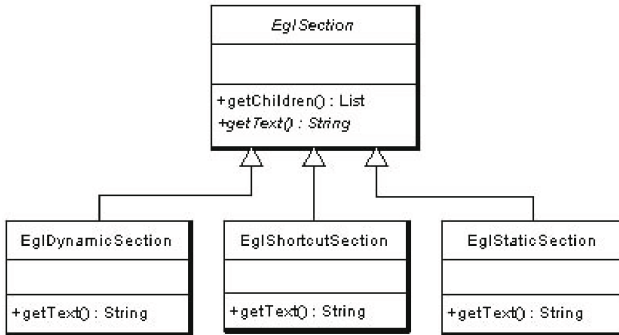


Fig. 2. The abstract syntax of EGL’s core

sections. This can be used to perform operations on the generated text, such as appending and removing strings and specifying the type of text to be generated.

EGL also provides syntax for defining *dynamic output* sections, which provide a convenient shorthand for outputting text from within dynamic sections. Similar syntax is often provided by template-based code generators.

3.2 Concrete Syntax

The concrete syntax of EGL mirrors the style of other template-based code generation languages. The tag pair [% %] is used to delimit a dynamic section. Any text not enclosed in such a tag pair is contained in a static section. Listing 1.1 illustrates the use of dynamic and static sections to form a basic EGL template.

Listing 1.1. A basic EGL template

```
1 [% for (i in Sequence{1..5}) { %]  
2 i is [%=i%]  
3 [% } %]
```

The [%=expr%] construct is shorthand for [% out.print(expr); %], which appends `expr` to the output generated by the transformation. Note that the `out` keyword also provides `println(Object)` and `chop(Integer)` methods, which can be used to construct text with linefeeds, and to remove the specified number of characters from the end of the generated text.

EGL exploits EOL's model querying capabilities to output text from models specified as input to transformations. For example, the EGL template depicted in Listing 1.2 may be used to generate text from a model that conforms to a metamodel that describes an object-oriented system.

Listing 1.2. Generating the name of each Class contained in an input model

```
1 [% for (class in Class.allInstances) { %]  
2 [%=class.name%]  
3 [% } %]
```

3.3 Parsing and Preprocessing

EGL provides a parser which generates an abstract syntax tree comprising static, dynamic and dynamic output nodes for a given template. A preprocessor then translates each section into corresponding EOL: static and dynamic output sections generate `out.print()` statements. Dynamic sections are already specified in EOL, and require no translation.

Consider the EGL depicted in Listing 1.1. The preprocessor produces the EOL shown in Listing 1.3 – the [% %] and [%= %] tag pairs have been removed, and the text to be output is translated into `out.print()` statements.

Listing 1.3. Resulting EOL generated by the preprocessor

```

1  for (i in Sequence{1..5}) {
2      out.print('i is ');
3      out.print(i);
4      out.print('\r\n');
5  }
```

When comparing Listings 1.1 and 1.3, it can be seen that the template-based syntax is more concise, while the preprocessed syntax is arguably more readable. For templates where there is more dynamic than static text, such as the one depicted in Listing 1.1, a template-based syntax is often less readable. However, this loss of readability is somewhat mitigated by EGL’s developer tools, which are discussed in Section 3.8. By contrast, for templates that exhibit more static than dynamic text, a template-based syntax is often more readable than its preprocessed equivalent.

3.4 Deriving EGL from EOL

In designing functionality specific to M2T transformation, one option was to enrich the existing EOL syntax with keywords such as `print`, `contentType` and `merge`. However, EOL underpins all Epsilon languages, and the additional keywords were needed only for M2T. Furthermore, the refactorings needed to support the new keywords affect many components – the lexer, parser, execution context and execution engine – complicating maintenance and use by other developers. Instead, we define a minimal syntax for EGL, allowing easy implementation of an EGL execution engine as a simple preprocessor for EOL.

The EGL execution engine augments the default context used by EOL during execution with two read-only, global variables: `out` (Section 3.2) and `TemplateFactory` (Section 3.5). The `out` object defines methods for performing operations specific to M2T translation, and the `TemplateFactory` object provides methods for loading other templates. The implementation for the latter was extended, late in the EGL development, to provide support for accessing templates from a file-system – a trivial extension that caused no migration problems for existing EGL templates, due to the way in which EGL extends EOL.

3.5 Co-ordination

In the large, M2T transformations need to be able to not only generate text, but also files, which are then used downstream as development artefacts. An M2T tool must provide the language constructs for producing files and manipulating the local file system. Often, this requires that the destination, as well as the contents, be dynamically defined at a transformation’s execution time [6].

The EGL co-ordination engine supplies mechanisms for generating text directly to files. The design encourages decoupling of generated text from output destinations. The `Template` data-type is provided to allow nested execution of

M2T transformations, and operations on instances of this data-type facilitate the generation of text directly to file. A factory object, `TemplateFactory`, is provided to simplify the creation of `Template` objects. In Listing 1.4, these objects are used in an EGL template that loads the the EGL template in Listing 1.2 from the file, `ClassNames.egl`, and writes out to disk the text generated by executing `ClassNames.egl`.

Listing 1.4. Storing the name of each Class to disk

```
1 [%  
2   var t : Template := TemplateFactory.load('ClassNames.egl');  
3   t.process();  
4   t.store('Output.txt');  
5 %]
```

This approach to co-ordination allows EGL to be used to generate one or more files from a single input model. Moreover, EGL's co-ordination engine facilitates the specification of platform-specific details (the destination of any files being generated) separately from the platform-independent details (the contents of any files being generated). The approach is compared to that in other M2T transformation tools in Section 5.

3.6 Merge Engine

EGL provides language constructs that allow M2T transformations to designate regions of generated text as *protected*. The contents of protected regions are preserved every time a M2T transformation generates text to the same destination.

Protected regions are specified by the `preserve(String, String, String, Boolean, String)` method on the `out` keyword – based on the `PROTECT` construct of the `XPand` language [18]. The first two parameters define the comment delimiters of the target language. The other parameters provide the name, enable-state and content of the protected region, as illustrated in Listing 1.5.

Listing 1.5. Protected region declaration using the `preserve` method

```
1 [%=out.preserve('/**', '*/', 'anId', true,  
2               'System.out.println(foo);')  
3 %]
```

A protected region declaration may have many lines, and use many EGL variables in the contents definition. To enhance readability, EGL provides two additional methods on the `out` keyword: `startPreserve(String, String, String, Boolean)` and `stopPreserve`. Listing 1.6 uses these to generate a protected region equivalent to that in Listing 1.5.

Listing 1.6. Protected region declaration

```

1  [%=out.startPreserve('\*', '\*/', 'anId', true)%]
2  System.out.println(foo);
3  [%=out.stopPreserve()%]

```

Because an EGL template may contain many protected regions, EGL also provides a separate method to set the target language generated by the current template, `setContentType(String)`. By default, EGL recognises Java, HTML, Visual Basic, Perl and EGL as valid content types. An alternative configuration file can be used to specify further content types. Following a call to `setContentType`, the first two arguments to the `preserve` and `startPreserve` methods can be omitted, as shown in Listing 1.7.

Listing 1.7. Setting the content type

```

1  [% out.setContentType('Java'); %]
2  [%=out.preserve('anId', true, 'System.out.println(foo);')%]

```

Because some languages define more than one style of comment delimiter, EGL allows mixed use of the styles for `preserve` and `startPreserve` methods.

Once a content type has been specified, a protected region may be declared entirely from a static section, using the syntax in Listing 1.8.

Listing 1.8. Declaring a protected region from within a static section

```

1  [% out.setContentType('Java'); %]
2  // protected region anId [on|off] begin
3  System.out.println(foo);
4  // protected region anId end

```

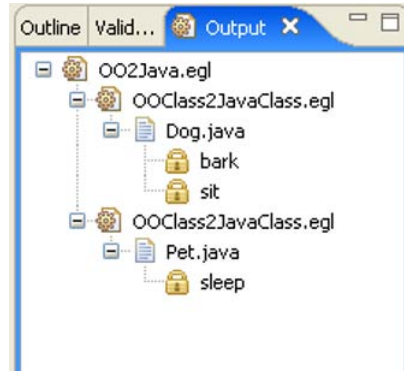
When a template that defines one or more protected regions is processed by the EGL execution engine, the target output destinations are interrogated and existing contents of any protected regions are preserved. If either the output generated by from the template or the existing contents of the target output destination contains protected regions, a merging process is invoked. Table 1 shows the default behaviour of EGL's merge engine.

3.7 Readability and Traceability

Conscientious developers apply various *conventions* to produce readable code. EGL encourages template developers to prioritise the readability of templates over the text that they generate. Like XPand [18], EGL provides a number of text post-processors – or *beautifiers* – that can be executed on output of

Table 1. EGL’s default merging behaviour

Protected Region Status	Generated	Existing	Contents taken from
On	On	On	Existing
On	On	Off	Generated
On	On	Absent	Generated
Off	On	On	Existing
Off	On	Off	Generated
Off	On	Absent	Generated
Absent	On	On	Neither (causes a warning)
Absent	On	Off	Neither (causes a warning)

**Fig. 3.** Sample output from the traceability API

transformations to improve readability. Currently, beautifiers are invoked via Epsilon’s extensions to Apache Ant [1], an XML-based build tool for Java.

EGL also provides a traceability API, as a debugging aid, and to support auditing of the M2T transformation process. This API facilitates exploration of the templates executed, files affected and protected regions processed during a transformation. Figure 3 shows sample output from the traceability API after execution of an EGL M2T transformation to generate Java code from an instance of an OO metamodel.

The beautification interface is minimal, in order to allow re-use of existing code formatting algorithms. Consequently, there is presently no traceability support for beautified text. However, due to the coarse-grained approach employed by EGL’s traceability API, this has little impact: clicking on a beautified protected region in the traceability view might not highlight the correct line in the editor.

3.8 Tool Support

The Epsilon platform provides development tools for the Eclipse development environment [4]. Re-use of Eclipse APIs allows Epsilon’s development tooling

to incorporate a large number of features with minimal effort. Furthermore, the flexibility of the plug-in architecture of Eclipse enhances modular authoring of development tools for Epsilon.

In addition to the traceability view shown in Figure 3, EGL includes an Eclipse editor and an outline view. In order to aid template readability, these tools provide syntax highlighting and a structural overview for EGL templates, respectively. Through its integration in the Epsilon perspective, EGL provides an Eclipse workbench configuration that is tailored for use with Epsilon’s development tools.

EGL, like other Epsilon languages, provides an Apache Ant [1] task definition, to facilitate invocation of model-management activities from within a build script.

4 Case Study

In this section, we demonstrate EGL’s capabilities and design with a case study. The example scenario requires analysis of the architecture and performance characteristics of a number of *systems*. Distinct metamodels are used to describe the way in which systems may be constructed and their response times.

The architecture metamodel, Figure 4, defines a *system* to comprise a number of *services*. A *Workflow* describes the combination of services needed to perform a complex task. The components of an example system are given in Table 2. In the system, the SearchForProperty workflow comprises the LookupDatabase, FilterUnsafeHouse and DisplayResults services; the BuyHouse workflow comprises SearchForProperty, and services, SelectProperty and PrintHouseDetails.

The metamodel defining system performance characteristics is shown in Figure 5. Each performance model comprises a number of *service implementations*, which have a name and a response time. The name attribute of the service implementation meta-class must correspond to the name attribute of the service meta-class in the architectural metamodel. The response time of workflows are

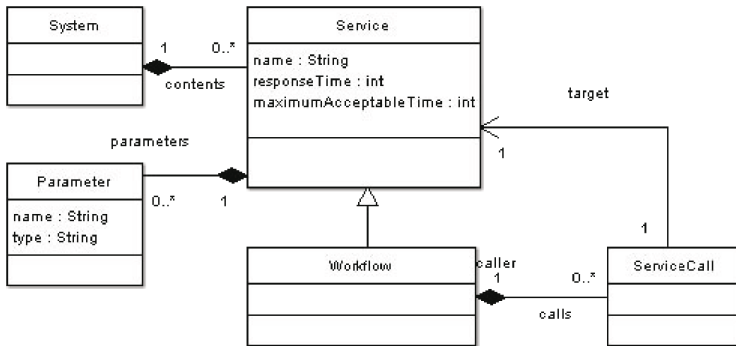


Fig. 4. The Service architecture metamodel

Table 2. An example instance of the Service metamodel

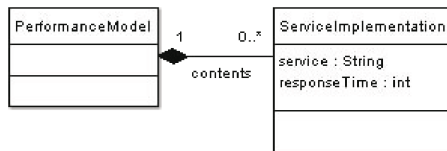
Type	Name	Max Response Time
Service	SelectProperty	3
Service	FilterUnsafeHouses	4
Service	DisplayResults	5
Service	PrintHouseDetails	6
Service	LookupDatabase	35
Workflow	SearchForProperty	
Workflow	BuyHouse	30

not included in the performance metamodel; these are derived from the response times of the services that make up the workflow.

Tooling was implemented using EGL and other Epsilon languages that allow the performance characteristics of a system to be calculated and visualised. Firstly, the Epsilon Comparison Language (ECL) is used to determine whether an instance of the architectural metamodel and an instance of the performance metamodel are compatible. A performance model, *p*, was deemed to be compatible with an architectural model, *a*, if, for each service implementation in *p*, there existed a service in *a* with name equal to the service implementation service.

Where instances have been shown compatible, the two are merged using the Epsilon Merging Language (EML). This creates an instance of the architectural metamodel that also contains response times. This allows the response times of each service and workflow to be compared with their maximum acceptable response times. The Epsilon Validation Language (EVL) is used to enforce this constraint and report any non-conformance.

Finally, EGL is used to produce a visualisation of the resulting model. The code, given in Listing 1.9, generates a table with a row for each service and workflow in the system, highlighting the performance characteristics of each. Example output is given in Figure 6. Of particular interest are the use of EOL's declarative functions on *collections* (a feature that EOL re-uses from OCL), which provide a concise means for expressing complex model inspections. For example, the use of `collect` on line 73 allows the total response time of a workflow to be calculated without explicit iteration.

**Fig. 5.** The ServicePerformance metamodel

Listing 1.9. The EGL code used to generate the visualisation (HTML for the table key is omitted)

```








1  <html>
2  <head>
3    <title>Service Model Visualisation</title>
4    <link title="default" rel="stylesheet" type="text/css"
5    href="Viz.css"/>
6  </head>
7  <body>
8
9  <table>
10    <tr>
11      <th>&nbsp;</th>
12      <th>Name</th>
13      <th>RT</th>
14      <th>MT</th>
15      <th>PE</th>
16      <th>SI</th>
17    </tr>
18    [%
19      services = Service.allInstances();
20      for (service in services.sortBy(s | s.getResponseTime())) {
21    %]
22      <tr>
23        <td>[%=service.getImage()%]</td>
24        <td>[%=service.name%]</td>
25        <td>[%=service.getResponseTime()%]</td>
26        <td>[%=service.maxAcceptableTime.toString()%]</td>
27        <td>[%=service.percentageExcess().toString()%]</td>
28        <td>[%=service.numberOfCalls()%]</td>
29      </tr>
30    [% } %]
31  </table>
32
33 </body>
34 </html>
35
36 [%
37   operation Any toString() : String {
38     if (self.isDefined()) {
39       return self;
40     } else {
41       return '&nbsp;';
42     }
43   }
44
45   operation Service percentageExcess() : String {
46     if (self.maxAcceptableTime.isDefined() and
47       self.maxAcceptableTime > 0) {
48
49       var percentage := 100 * self.getResponseTime() /
50                       self.maxAcceptableTime;
51       return (percentage - 100) + '%';
52     }
53   }
54
55   operation Service numberOfCalls() : Integer { return 1; }
56
57   operation Workflow numberOfCalls() : Integer {
58     return self.calls.collect(c|c.target.numberOfCalls())
59       .sum().ceiling();
60   }
61
62   operation Service getImage() : String {
63     return '';
64   }

```

```

65
66 operation Workflow getImage() : String {
67     return '';
68 }
69
70 operation Service getResponseTime() : Integer {
71     return self.responseTime;
72 }
73
74 operation Workflow getResponseTime() : Integer {
75     return self.calls
76         .collect(c|c.target.getResponseTime())
77         .sum();
78 }
79
80 operation Any getResponseTime() : Integer { return 0; }
81 %]

```

	Name	RT	MT	PE	SI
	SelectProperty	3			1
	FilterUnsafeHouses	4			1
	DisplayResults	5			1
	PrintHouseDetails	6			1
	LookupDatabase	35			1
	SearchForProperty	44			3
	BuyHouse	53	30	76.66667%	5

Key
RT Response Time
MT Maximum Acceptable Response Time
PE Percentage Excess
SI Number of Services Invoked

Fig. 6. Example output from the model visualisation phase

5 Related Work

5.1 JET 2.0

JET [5] is perhaps the most popular code-generation framework available for the Eclipse platform. JET’s dynamic sections employ custom XML tags in order to describe control flow. Attributes of these tags may include XPath [3] path expressions to support model interrogation. Developers may also include dynamic sections, written in Java, in JET templates.

Out of the box, JET can perform transformations only upon XML- and EMF-based models, and, unlike EGL, does not provide support for models implemented using MOF 1.4 or CZT. Furthermore, while XPath provides a concise means for specifying navigation of tree structures, it lacks some of the expressiveness of the OCL-like constructs for navigating collections (e.g. select, reject, forAll) that Epsilon provides through EOL.

JET provides support for active code generation, via the `c:userRegion` and `c:initialCode` constructs. This is slightly more flexible than EGL, where text used in EGL protected region markers has to conform to a simple grammar. However, this slight loss of flexibility enables EGL to provide constructs to simplify protected region demarcation and thus to reduce duplication in templates.

5.2 MOFScript

MOFScript [15] has influenced the OMG's MOF-based M2T transformation RFP. MOFScript's declarative style of syntax has similarities to the syntax style of M2M transformation languages such as ATL. The declarative approach allows transformation rules to use sophisticated mechanisms for abstraction and code re-use, such as inheritance. However, EGL provides much the same scope for reducing duplication of code using an imperative syntax plus facilities for code modularisation. Although the way in which abstraction and re-use is achieved is slightly less succinct in the imperative approach, the resulting templates are more readable.

A key problem with MOFScript is that it encourages transformations with tight coupling of destination and content: the MOFScript file type allows transformations to write generated text (content) directly to disk. This means that modification is needed to use the same transformation to generate content to a different type of destination (a socket, an HTTP stream, etc.). The EGL style encourages developers to separate destination and content of generated text, by restricting direct access to output destinations from within templates, as discussed in Section 3.5.

Unlike JET, MOFScript provides some OCL-like constructs for traversing and interrogating data structures (forEach and select keywords). MOFScript also provides a prototype implementation for aspect-oriented programming constructs, allowing transformations to be woven together at compile-time.

5.3 XPand

The openArchitectureWare platform, oAW [17], provides open-source, model-driven software development tools. It includes a M2T language, XPand [18], with a declarative template syntax. XPand meets many of the requirements outlined in Section 2.1. For instance, the language supports active code generation via the PROTECT construct, and provides beautifiers to enhance readability of both templates and generated text. However, like MOFScript, XPand encourages transformations to couple destination and content, which limits re-usability.

Unlike the other M2T languages considered, XPand templates can be invoked as part of a workflow, using oAW's proprietary workflow definition language. By contrast, Epsilon utilises Apache Ant to define workflows, which encourages reuse of existing tools, such as AntUtility [7], for profiling, and the Nurflugel AntScript Visualizer [2] for visualisation.

6 Conclusions and Further Work

In this paper, we have presented the Epsilon Generation Language, a template-based M2T transformation language for the Epsilon platform. Through its derivation from EOL, EGL provides features specific to the M2T transformation domain as well as having direct access to general model-management support, and to future enhancements to EOL and the Epsilon platform. Furthermore, by deriving EGL from EOL, we have been able to simplify the language design activity of EGL development, and re-use the EOL execution engine.

We are now working on combining EGL with Epsilon's languages for model comparison and transformation (ECL and ETL), to support incremental code generation. This would allow users to reflect in code all changes made to source models, by applying a minimal number of transformations. We are also investigating an alternative approach, using change information derived by model editing tools to perform impact analysis; by adding keywords to EGL this would allow direct checking of staleness of model elements.

An alignment of EGL with a web-server has a number of potentially interesting applications. The case study in Section 3.5 shows the potential of using EGL as a component in a web-based model repository. Extending this idea, the Epsilon languages could provide a scaffold for developing web-based applications: suppose the domain objects of such an application were encoded as EMF-compliant models – Epsilon's transformation language could be used to describe suitable transformations on the domain model (such as adding, editing or removing an instance); the Epsilon Validation Language could check that models remain valid and consistent subsequent to domain model transactions; and EGL could be used to produce HTML for viewing domain objects.

Information on EGL, Epsilon and associated languages is available from the Epsilon GMT project website, <http://www.eclipse.org/gmt/epsilon>.

Acknowledgement. The work in this paper was supported by the European Commission via the MODELPLEX project, co-funded by the European Commission under the “Information Society Technologies” Sixth Framework Programme (2006-2009).

References

1. Apache. The Apache Ant Project (2007), <http://ant.apache.org/>
2. Bullard, D.: Ant Script Visualizer (2005), <http://www.nurflugel.com/webstart/AntScriptVisualizer/>
3. World Wide Web Consortium. XML Path Language (XPath) Version 1.0 (1999), <http://www.w3.org/TR/xpath>
4. The Eclipse Foundation. Eclipse - an open development platform (2007), <http://www.eclipse.org>
5. The Eclipse Foundation. JET, part of Eclipse's Model To Text (M2T) component (2007), <http://www.eclipse.org/modeling/m2t/?project=jet#jet>
6. Frankel, D.: Model Driven Architecture: Applying MDA to Enterprise Computing. Wiley, New York (2003)

7. David Green. Ant Utility (2007), <https://antutility.dev.java.net/>
8. ATLAS Group. Atlas Transformation Language Project Website (2007), <http://www.eclipse.org/m2m/at1/>
9. The Object Management Group. OMG Official Website (2007), <http://www.omg.org>
10. Kleppe, A.G., Warmer, J., Bast, W.: MDA Explained: The Model Driven Architecture: Practice and Promise. Addison-Wesley Longman Publishing Co., Inc, Boston (2003)
11. Kolovos, D.S.: Extensible Platform for Specification of Integrated Languages for mOdel maNagement Project Website (2007), <http://www.eclipse.org/gmt/epsilon>
12. Kolovos, D.S., Paige, R.F., Polack, F.: Epsilon Development Tools for Eclipse. In: Eclipse Summit 2006, Esslingen, Germany (October 2006)
13. Kolovos, D.S., Paige, R.F., Polack, F.: The Epsilon Object Language (EOL). In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 128–142. Springer, Heidelberg (2006)
14. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: A Short Introduction to Epsilon (2007), <http://www-users.cs.york.ac.uk/~dkolovos/epsilon/Epsilon.ppt>
15. Oldevik, J., Neple, T., Grønmo, R., Agedal, J.Ø., Berre, A.-J.: Toward standardised model to text transformations. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 239–253. Springer, Heidelberg (2005)
16. OMG. MOF Model to Text Transformation Language RFP (2005), <http://www.omg.org/docs/ad/04-04-07.pdf>
17. openArchitectureWare. openArchitectureWare Project Website (2007), <http://www.eclipse.org/gmt/oaw/>
18. openArchitectureWare. XPand Language Reference (2007), http://www.eclipse.org/gmt/oaw/doc/4.1/r20_xPandReference.pdf
19. Varró, D., Balogh, A.: The model transformation language of the VIATRA2 framework. *Sci. Comput. Program* 68(3), 187–207 (2007)

Constructing and Visualizing Transformation Chains

Jens von Pilgrim¹, Bert Vanhooft², Immo Schulz-Gerlach¹,
and Yolande Berbers²

¹ FernUniversität in Hagen, 58084 Hagen, Germany

{Jens.vonPilgrim, Immo.Schulz-Gerlach}@FernUni-Hagen.de

² DistriNet, K.U.Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium*
{bert.vanhooft, yolande.berbers}@cs.kuleuven.be

Abstract. Model transformations can be defined by a chain or network of sub-transformations, each fulfilling a specific task. Many intermediate models, possibly accompanied by traceability models, are thus generated before reaching the final target(s). There is a need for tools that assist the developer in managing and interpreting this growing amount of MDD artifacts. In this paper we first discuss how a transformation chain can be modeled and executed in a transformation language independent way. We then explore how the available traceability information can be used to generate suitable diagrams for all intermediate and final models. We also propose a technique to visualize all the diagrams along with their traceability information in a single view by using a 3D diagram editor. Finally, we present an example transformation chain that has been modeled, executed and visualized using our tools.

1 Introduction

Monolithic transformations, like most non-modularized software entities have some inherent problems: little reuse opportunities, bad scalability, bad separation-of-concerns, sensitivity to requirement changes, etc. A number of these problems can be solved by decomposing a transformation into a sequence of smaller sub-transformations: a transformation chain or transformation network.

Each stage of a transformation chain produces intermediate models, which means that the total number of models can become very large. In the ideal case this all happens automatically and correctly so we do not have to care much about the intermediate models; only the final target models are to be considered. Unfortunately, we often have to study and possibly modify intermediate models manually in order to get the desired result, if only for debugging purposes. For example, if we detect an unexpected structure in the final output

* Some of the described work is part of the EUREKA-ITEA MARTES project, and is partially funded by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT), by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

model we must look at the intermediate models to identify the responsible transformation. In another case we might want to optimize the results of a complex sub-transformation manually. In these cases, it may also be useful to inspect the relations between the models under study – i.e. traceability. We have identified three main problems with the scenarios sketched above.

1. Defining, executing and maintaining transformation chains is not a straightforward task. This is particularly difficult if a mixture of different transformation languages and tools is used.
2. Relying on automatic layout mechanisms to generate diagrams for intermediate models is often not satisfactory. Since layout information is neither transformed nor preserved across transformation stages, each model’s diagram can look completely different even if subsequent models have only slightly changed.
3. Trying to interpret traceability information manually is difficult since there is no good graphical representation technique for this kind of information. Therefore, this information can often not be fully exploited.

In this paper we explain our approach to construct a transformation chain and present its output to a human user in a clear and comprehensible fashion.

We have extended UNITI (Unified Transformation Infrastructure) [1], a tool for transformation chain modeling and execution, with the ability to keep track of traceability models and their relation to other models in the chain. We discuss how the output of a transformation chain – an intricate network of models connected by traceability links – can be visualized. We explain how a proper diagram layout can be produced for the generated models by leveraging traceability information. Furthermore, we propose a technique to visualize the diagrams, together with traceability links in a 3D editor.

The remainder of this paper is structured as follows. In Section 2 we give a short overview of UNITI. Since traceability plays a crucial role in transformation chains, we discuss our perspective on that subject in Section 3. In Section 4 we describe how to automatically create the layout for generated models and introduce GEF3D, the framework used for implementing a 3D diagram editor. We demonstrate our solution with an example transformation chain in Section 5 and summarize related work in Section 6. Finally, we wrap up by drawing conclusions and identifying future work in Section 7.

2 Setting Up a Transformation Chain

Currently, most transformation technologies do not offer much support for re-using and composing sub-transformations as high-level building blocks. They focus on offering good abstractions and a clear syntax for implementation. Since many transformation languages are now reaching a certain maturity, we need to start focussing on reuse and composition of transformations.

We have developed an Eclipse plugin called UNITI (Unified Transformation Infrastructure) that manages building blocks of a transformation chain. In this

section we give a short overview of UNITI, for a more elaborate discussion we refer to [1].

The basic principles of UNITI are inspired on those of Component Based Software Engineering (CBSE) [2]. We have reformulated them to fit the model transformation domain:

Black-Box. The black-box principle indicates a strict separation of a transformation's public behavior (what it does) and its internal implementation (how it does it). Implementation hiding makes any technique an eligible candidate for implementing the transformation.

External specification. Each transformation should clearly specify what it requires from the environment and what it provides.

Composition. Constructing a transformation with reusable transformation building blocks should be considerably less work than writing it from scratch. A transformation should be a self-contained unit and composition should be easy and should require only a minimum of glue code.

Current transformation technologies do not focus on the principles outlined above; in order to reuse an existing transformation implementation in a chain, a very deep knowledge of that implementation is usually required [1]. This violates the black-box principle and is often consequence of the limited external specification. In the related work section we discuss that QVT does have limited support for the above principles.

In UNITI we try to not only adhere to the three CBSE principles, but also to MDD practices. A transformation chain in UNITI is therefore a model itself, making it a possible subject to MDD techniques such as model transformation.

A typical usage scenario of UNITI goes as follows (see Figure 1). A *Transformation Developer* who is specialized in one or more transformation languages provides a number of executable transformations. The *Transformation Specifier* is responsible for encapsulating these implementations in the the unified UNITI representation (*TFSpecification*). This entails specifying the valid context in which the transformation may be executed by defining pre- and post-conditions. Depending on the transformation language used for the implementation, it might also involve choosing concrete metamodels [3] or a transformation

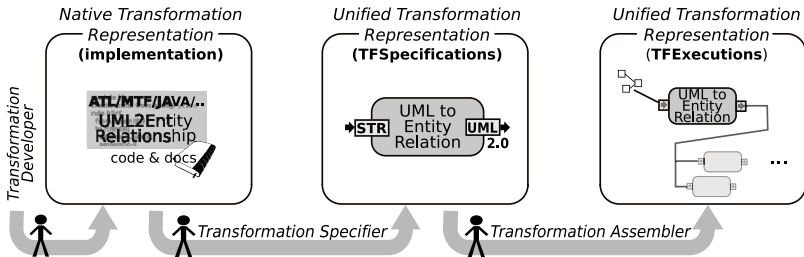


Fig. 1. Overview of UNITI principles

direction [4]. Currently we support semi-automatic generation of such transformation specifications for ATL [3], MTF [4] and Java transformations. Finally, the *Transformation Assembler* creates a transformation chain model by interconnecting an arbitrary number of transformation instances (*TFExecutions*). This can be done without any knowledge of implementation or technology details of the underlying transformations since these are completely hidden by UNITI at this point. Each transformation, be it ATL, MTF or Java, is represented in exactly the same way. Once the transformation chain is completely modeled, UNITI automatically executes all stages and produces the intermediate and final models.

3 Transformation Traceability

Traceability has many applications in software engineering. In requirements engineering, traces make the link between requirement documents and design models or code while program analysis focuses on traces between function calls. From an MDD point of view, traces can be any type of relation between model elements. Traceability information is typically expressed as a model of its own: traces are elements of a traceability model.

For this paper, we consider an MDD-specific kind of traceability: transformation traceability. In this case, traces are produced by automatic model transformations and denote how target elements are related to source elements and vice versa. In this section we discuss how traceability is supported in UNITI (see 3.1) and which traceability metamodels we use (see 3.2).

3.1 Traceability Support in UNITI

Since traceability information can be represented as a model itself, it can easily be produced as an additional output of a transformation. We assume that the creation of traceability models is explicitly implemented by the transformations and that traceability models are available as normal transformation outputs.

While regular models can usually be interpreted independently, traceability models contain cross-references to input and output models and are thus meaningless on their own. These references are usually unidirectional in order to prevent pollution of the regular models. This means that, given a traceability model, we can locate the models to which it refers, but not the other way around. Given only a model, we cannot locate the relevant traceability model(s).

Therefore, we have extended UNITI so it keeps track of the relations between all models and traceability models involved in a transformation chain. Tools such as GEF3D (see Section 4) can hence query the transformation chain model to determine how models and traceability models are related.

Figure 2 shows a part of the UNITI metamodel that introduces dedicated support for traceability. This is a new feature that was not yet described in [1]. A *TFExecution* (an instantiated transformation) has a number of input and output parameters (*TFParameter*), which can be considered as model containers.

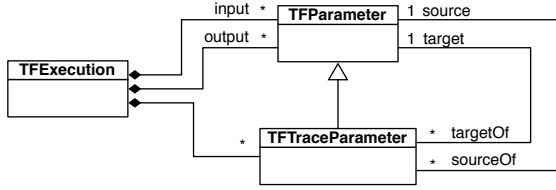


Fig. 2. Traceability Tracking in UNITI

Traceability parameters (*TFTraceParameter*) represent a special type of parameter that encapsulates a traceability model and keeps track of its *source* and *target* model; *TFParameters* have opposite references: *sourceOf* and *targetOf*. Notice that this structure (multiplicities of 1 for both *source* and *target*) enforces a separate traceability model for each input/output model pair. This decision is made to keep sub-transformations as independent as possible and to prevent sub-transformation implementation details from leaking (see also next subsection).

In summary, traceability models play a first class role a transformation chain modeled in UNITI, which facilitates traceability discovery by model navigation.

3.2 Traceability Metamodel

A lot of different metamodels for traceability information have already been suggested in literature, e.g. in the MDA foundation model [5]. Some of them are specialized, others propose a generalized model. At this point in time there does not seem to be a generally accepted model. One could wonder whether a generic model is necessary since traceability models can be transformed to conform to a different metamodel if required. For this paper we use this transformation approach since both UNITI and GEF3D use different traceability models.

The traceability models that we use are illustrated in Figure 3. On the left, we see the traceability metamodel that is used by UNITI and on the right, the metamodel required by GEF3D. Note that UNITI can also be used without traceability and supports any kind of traceability metamodel, but in order to enable some additional uses of traceability [6], this particular metamodel is required.

The **UNITI Traceability Metamodel** (left part of Figure 3) is designed to minimize exposing implementation details of individual transformations in order to prevent tight coupling within a transformation chain. A sub-transformation should never be able to depend on specific implementation details (not part of its specification) of a previous transformation. However, many transformation traceability models indirectly expose these kind of details by including the (implementation specific) transformation rule name in each trace. Even if this is not the case, an $m : n$ trace that links all elements involved in a single transformation rule, can still expose the rule structure. This means that subsequent transformations can depend on this structure, causing hard coupling within the transformation chain. As a result, sub-transformations cannot easily be replaced by others without breaking the chain.

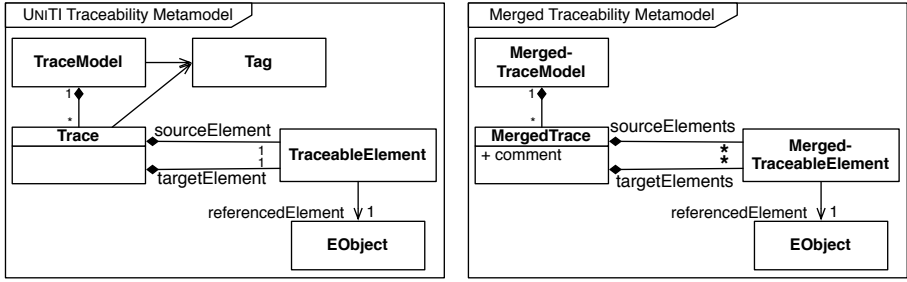


Fig. 3. UNiTI Traceability Metamodel and Merged Traceability Metamodel

In order to hide these implementation details, the UNiTI traceability metamodel only allows binary trace relations. A tagging mechanism is offered to annotate individual traces with a specific semantic meaning that is independent of the implementation. In the future, these tags will make part of the specification of a transformation. More details on these tags can be found in [6].

The **Merged Traceability Metamodel** (MTM – right part of Figure 3), used in GEF3D, is similar to the UNiTI traceability metamodel but has at least two important differences. First, a trace can contain a comment, which can be the transformation rule name. Second, a trace can contain multiple source and target elements. The main design consideration of the merged traceability model is to minimize the amount of traces (*MergedTraces*) that are given to the diagram layout algorithm (see Section 4.1). This is accomplished by creating only a single trace element for each group of related model elements; typically a single source element with a number of target elements.

It is clear that both traceability metamodels have different goals and therefore have a different structure. In order to overcome these differences, we have used a model transformation to translate UNiTI traceability models to merged traceability models. By using this technique, we ensure that both tools are loosely coupled and that their traceability metamodels can evolve independently. If one changes, it suffices to change the transformation.

In the next section we show how the merged traceability model is used to generate model diagrams and how traces between different models can be visualized in a 3D editor.

4 Visualization

When visualizing a transformation chain, two main issues can be identified: How to layout diagrams of generated models and how to visualize traces.

UML diagrams such as class diagrams can become quite complex. Often, much time is spent by the user to layout a single model. For example, related elements are grouped together or line crossings are minimized. While automatic layout algorithms may achieve good results with respect to some optimization criteria, users often try to visualize semantics in the layout which is not expressed in the

model itself. For example in Figure 6 important classes are put in the center of the diagram, something that cannot be mimicked by layout algorithms.

A big disadvantage of model transformations is that they don't transform layout information; diagrams are usually not transformed. If we want to take a look at the final output model or at any of the intermediate models, we have to rely on automatic layout algorithms. Since they only consider the current model, the spacial relation to the previous diagrams is often lost. That is, the user has created a mental map of the source diagram, knowing where on the diagram certain elements are located. Common layout algorithms do not take this mental map into account, therefore it can be very difficult to locate corresponding elements in the target diagram. This is especially true for chains in which intermediate models were used to simplify a single transformation – the structure of the different models is usually very similar and thus it is even more important to retain the mental map.

In [7] Kruchten writes: “Visual modeling [...] helps to maintain consistency among a system’s artifacts.” [7, p. 11]. A base feature to “maintain consistency” of models are traces, but traces are rarely visualized. Displaying traces between elements of two models makes the diagram even more complex. A first approach may be to draw the two models beneath or besides each other and to display the traces as lines connecting related elements. But in this case, the diagrams quickly become cluttered; many traces cross each other and other diagram elements and the whole diagram becomes quite large (difficult to display on a screen). Changing the diagram layout in order to minimize line crossings results in destroying the manual layout and the mental map.

While traces make the problem of visualizing a transformation chain more complex, they can help us in solving the layout problem. We show how to use the traceability models, produced by the transformation chain, to automatically layout intermediate diagrams based on the initial manually created model diagrams, hence preserving the mental map (see 4.1).

For solving the traceability visualization problem we “expand” the dimensions in which we draw the diagrams. Most software engineering models are drawn using two-dimensional diagrams. For drawing models with traces we can use three-dimensional diagrams, using the third dimension to display the traces. In most cases editors for single models already exist and we want to reuse these editors. We therefore adapt existing two-dimensional editors and combine them to a single 3D multieditor. This can be achieved by displaying the 2D diagrams on 3D planes. Traces can then be visualized as connections between elements of the different models displayed on these planes (see 4.2).

4.1 Derived Layout

The transformations used in a transformation chain usually only transform the domain models. In most cases these domain models do not contain any layout information. The layout of diagrams is defined using so called notation models, but these models are not transformed. This is why generated models do not have a layout and why we rely on layout algorithms in order to display these

$$\begin{array}{ccccccc}
DM_1 & \dots & \xrightarrow{t_{i-1}} & DM_i & \xrightarrow{t_i} & DM_{i+1} & \xrightarrow{t_{i+1}} \dots \\
\\
NM_1 & \dots & \xrightarrow{t_{NM}} & NM_i & \xrightarrow{t_{NM}} & NM_{i+1} & \xrightarrow{t_{NM}} \dots
\end{array}$$

Fig. 4. Transformation of Notation Models

models. What we suggest here is a layout algorithm which is indeed a kind of a transformation. Figure 4 illustrates the idea.

The first row in the figure shows the transformation chain as created by UNITI. Some domain models DM_n are related via transformations t_n . Additional notation models NM_n are used in order to display the domain models. To preserve the structure of a diagram, that is the layout created by a user in the very first notation model NM_1 , we have to make use of the information stored in the notation model of the predecessor. Therefore we map the source notation model to a target notation model. This is possible because a path $NM_{i+1} \rightarrow DM_{i+1} \rightarrow DM_i \leftarrow NM_i$ exists.¹ A detailed description of the algorithm can be found in [8].

4.2 2D Diagrams in a 3D Space

A lot of graphical editors are implemented today using the Eclipse Graphical Editing Framework (GEF)[9]. Examples for such editors are the TopCased Tools [10], or the Eclipse UML Tools. For visualizing models and traces in 3D, we do not want to re-implement these editors, even if they are 2D only. The main challenge here is to provide a technique allowing us to use existing editors in a 3D space. Besides minimizing implementation effort, we also retain the graphical syntax used by these editors. Since we actually use the original editor code, the 3D version does not only display the models but also enables us to modify the diagrams in this 3D view.

The concept of displaying models in a 3D space is to project common 2D diagrams on planes and combine these 2D diagrams with real 3D elements. Here, traces are visualized as 3D elements connecting 2D elements of the original diagrams. Figure 5 shows four planes on which 2D digrams are drawn. 3D connections (here traces) connect elements of these diagrams. The models of the figure are explained more detailed in Section 5.

While 3D editors (or visualizations) are used in many domains (e.g. CAD), they are rarely used in software engineering. Most software engineering diagrams are graphs, that is nodes and connections drawn as lines between these nodes. GEF is a framework for implementing editors for this kind of diagrams. Figure 6

¹ While the transformation is usually unidirectional, the trace is always bidirectional ($DM_{i+1} \rightarrow DM_i$). Since domain models are independent from the notation models, i.e. the connection is directed from the notation model to the domain model ($DM_i \leftarrow NM_i$), we have to use a reverse lookup, but this problem can easily be solved.

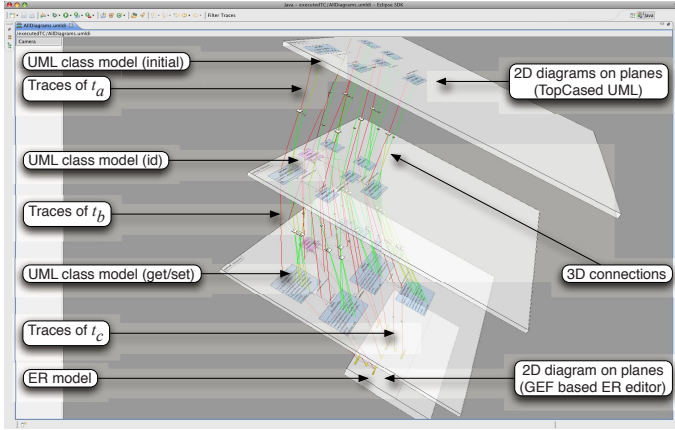


Fig. 5. 2D diagrams on planes and 3D connections

illustrates the main concepts of GEF and GEF3D. GEF is using a Model-View-Controller (MVC) architecture shown in the center of the figure. For each element drawn and edited in a GEF based diagram an MVC triple has to be implemented. The view for each element is called figure (and its class has to implement Draw2D's *IFigure* interface). The controller of a single element is called edit part (and its class has to implement GEF's *(Graphical)EditPart* interface). GEF is independent from any model implementation, but most often models are implemented using the Eclipse Modeling Framework (EMF) [11]. The figures and the edit parts are organized in a tree. The root of the figures is contained in a so called lightweight system which is a bridge between the graphical system of the platform and the figures. The edit parts are usually created by an edit part factory, using the factory pattern. All parts of a graphical editor are held together by an (edit part) viewer.

With GEF3D [12] we provide a 3D extension of GEF, making it possible to display 2D diagrams on planes in a 3D space or even to implement real 3D editors. It provides an OpenGL based implementation for figures, i.e. it provides a new interface for 3D figures (*IFigure3D*, which extends the original interface *IFigure*). The original lightweight system and the canvas, on which the figures are drawn, are replaced by 3D versions too. Instead of a 2D canvas an OpenGL canvas is used, that is we use OpenGL for rendering. These new 3D enabled view elements are instantiated by corresponding 3D versions of the graphical editor and viewer. The core of GEF can be used without further changes.

Adapting an existing GEF based editor is very simple: Instead of rendering the diagram directly on the screen, they are rendered as images which are used as textures for some 3D elements. Most parts of the 2D editors can be reused, only the top level container figure (i.e. the diagram itself) and its controller have to be replaced. In MDD, UML is often used for modeling. TopCased UML provides

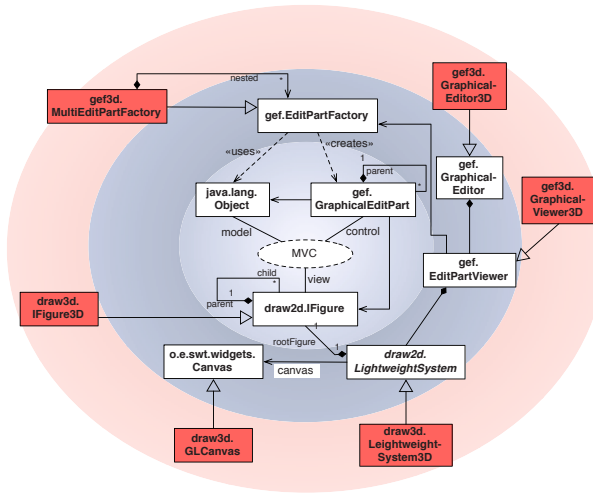


Fig. 6. GEF3D adds 3D to GEF

editors for most UML diagrams, and we adapted TopCased UML so that its diagrams can be rendered in a 3D editor. The result is not only a visualization of UML diagrams, but also a full-featured editor ².

Since we do not only display a single diagram, multiple editors have to be combined in a single editor called multi editor. This is possible due to GEF’s design of the controller components, the edit parts. These parts are created using a so called edit part factory. GEF3D extends this pattern by providing a multi edit part factory. This factory nests already existing factories and creates new elements using a nested factory based on the diagram element in which the new element is contained. As demonstrated in the following example, not only editors displaying the same type of diagrams can be combined, but also completely different editors.

The performance of the 3D visualization is quite well due to the usage of OpenGL. All 2D elements are currently rendered as images (one image per 2D diagram), so if the diagrams aren’t too large, this causes no problem. We have also implemented some performance tests, showing that 1000 real 3D nodes with textures can be handled by the engine.

5 Example

In this Section we demonstrate the usage of UNITI and the capabilities of GEF3D by creating and visualizing a typical transformation chain. The transformation chain starts with a simple UML model, creates several UML models, and finally an Entity-Relationship (ER) model. The models represent, even if slightly

² Currently not all editor features are implemented, see Section 7.

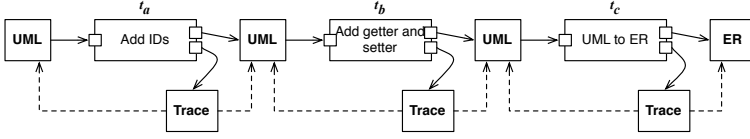


Fig. 7. Schematic of the example transformation chain

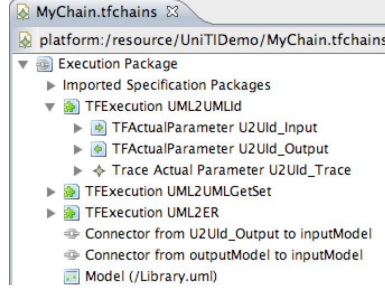


Fig. 8. Part of the transformation chain as modeled in UniTI

simplified, typical artefacts used when creating an information system. We first set up the transformation chain with UniTI and then visualize the whole chain with a GEF3D based editor.

The example transformation chain is schematically represented in Figure 7. It contains three transformation stages:

1. UML refinement – Id attribute (t_a): The first (UML) model represents a domain model of a library. Since it is a domain model, implementation details such as operations and persistency properties are omitted. The first refinement transformation t_a adds a unique identifier property to each class. For the initial model, we have manually created a model diagram so that further diagrams can be generated automatically (as described in 4.1).
2. UML refinement – getter and setter (t_b): For each public property, a public get and set method is added to control access to that property (which can also refer to an association). The property itself is then marked as private. This is useful if we want to add further transformation towards, e.g., a Java implementation.
3. UML to ER (t_c): Finally we transform the UML model to a corresponding ER model.

This transformation chain was modeled in UniTI (see Figure 8) and the results were visualized using GEF3D (see Figure 5).

The traces are displayed in different colors, depending on the level of the connecting elements. That is, traces connecting top level elements (e.g. classes or associations) are drawn red, while traces connecting nested elements (like operations) are drawn green. We have implemented a filter to hide traces of the

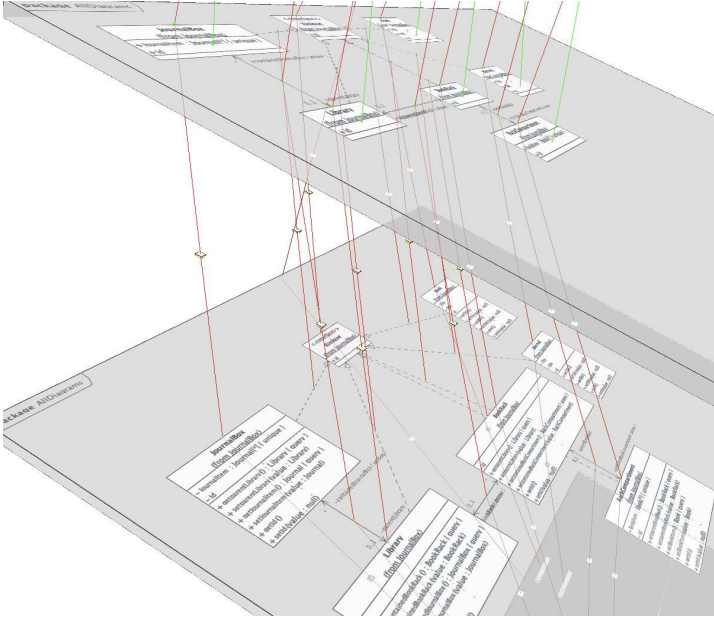


Fig. 9. Transformation chain output with filtered trace visualization

latter kind making the traceability much more clear. Figure 9 shows some filtered traces. Since this paper leaves no room for more pictures, we refer to GEF3D's³ and UNITI's⁴ website, which provide some screenshots and screencasts.

6 Related Work

The approach that was presented in this paper consists of three main areas: transformation chaining, traceability and visualization of models. In this section we identify related approaches in each of these areas.

6.1 Transformation Chains

There are a number of alternatives to UNITI that allow to build and execute transformation chains.

In [13], a transformation chain is seen as the composition of different tools that each support one or more transformations. They make a strong separation between transformation definition and execution. Transformation building blocks are represented as plugins to the Eclipse framework. Another approach similar to ours is described in [14]. They reuse classical component middleware as a

³ <http://gef3d.org/>

⁴ <http://www.cs.kuleuven.be/~bertvh/uniti.shtml>

Transformation Composition Framework (TCF); transformations, models and metamodels are all represented as components.

These systems have a similar goal as UNITI, but there are some significant differences. UNITI offers a model-based approach to specify and execute transformation chains, while the former systems use a scripting language. A model-based approach has the advantage that transformation chain models can easily be explored by other MDD tools (e.g. the GEF3D editor in Section 5) and can even be transformed themselves, for example to insert additional transformations. UNITI also integrates first class support for traceability models, which is not present in the other approaches.

Similar to UNITI, QVT [15] supports chaining of black-box transformations. However, it is up to each QVT implementation to provide concrete mechanisms to call out to external transformations. At the moment of writing, existing implementations of Borland and SmartQVT only enable Java to specify black-box transformations.

The AM3 Global Model Management approach [16] aims to build a generic model management infrastructure, including support for transformations, metamodels and other MDD artifacts. A similar approach, though more specialized towards model merging, is pursued in [17]. UNITI can be considered as one application of model management, focused on managing transformation related MDD resources.

6.2 Traceability

In our approach, we generate and visualize traceability models and use these models to generate diagram layouts. Related work concerning the generation of traceability and traceability in general is presented in this subsection, visualization related work is presented in the next subsection.

Traceability Metamodels. Many traceability metamodels have already been proposed for this purpose [18]. Some of them have a very specific purpose, others try to come to a generic model that is suited for every situation. We do not believe that such a generic model is feasible. Instead we were confronted ourselves with two different traceability metamodels when integrating UNITI and GEF3D. We have not attempted to adapt both tools to use a common traceability metamodel but rather used a model transformation to convert the traceability model when needed (see Section 3.2).

Traceability Generation. Most current transformation languages [15,3] build an internal traceability model that can be interrogated at execution time, for example, to check if a target element was already created for a given source element. Although this kind of traceability is generated without any additional required effort, there are some issues. Firstly, not all languages allow to export this information into an additional model, this is the case for ATL and MTF. Secondly, the traces cannot be adapted from within the transformation rules, which is required for our trace tagging approach. For these reasons, it is often

necessary to explicitly incorporate the generation of an addition traceability model into each transformation, as we did in our approach.

Manual implementation of traceability generation can be very cumbersome since very similar trace generating statements must be reproduced for each transformation rule. A solution for automating parts of this work has been proposed in [19]. They explain how trace generation statements can be added to an existing transformation by a model transformation – i.e. the transformation is transformed itself. This technique can be used for transformations in UNITI as well.

In [20], a distinction is made between traceability in the small (between model elements) and traceability in the large (between models). In Section 3.1, we make a similar distinction: traceability models contain links between model elements and UNITI keeps links between models and their traceability models.

6.3 Visualization

There are three kinds of related work: Work related to visualizing (software engineering) models in 3D, work related to visualizing diagrams on planes, and work related to visualizing traces.

Software engineering models like class diagrams are visualized in a 3D manner for example in [21]. This work is not using common notations, but invents new ones optimized for 3D visualization. The new notation syntax has to be learned and it differs from common notations like UML, hence existing editors cannot be reused. Often the visualization tools can only present, but not edit the diagrams, while GEF3D has the potential of a real full-feature editor based on a widely used framework.

The idea of visualizing 2D diagrams on planes is described in [22] already. In this work, 3D diagrams that look quite similar to the ones presented here, are shown. However, the diagrams here were produced by an existing tool. In [22] the diagrams are only illustrations of an idea, but a tool was not yet implemented. As far as we know there still is no tool, at least publicly, available.

Most work about visualizing traces examines requirement or execution traces. In the first case, often matrices are used, a column for each requirement and a row for each artifact [23]. In [24] execution traces are visualized in 3D diagrams, but again new 3D notation syntax is used. Since usually a lot of execution traces are produced in a software system, execution traceability becomes more a quantitative information. The effect of this is that a single trace cannot be identified anymore.

7 Conclusion and Future Work

Complex model transformations can often be split up into smaller sub-transformations in order to provide better robustness to changes, to allow more reuse opportunities and to simplify the implementation of individual transformations. A transformation chain thus produces many intermediate models before delivering the final output model(s).

While this approach seems very attractive, we have identified three issues. First, we need an approach to manage all the high level MDD artifacts in a transformation chain: models, metamodels and transformations. Second, model diagram information must also be preserved throughout the transformation chain to allow manual inspection of the intermediate and final models. Third, we need a good representation for traceability information between different models.

We have proposed a solution to these problems by integrating two tools. UNITI allows to model and execute transformation chains in a technology independent way. It also keeps track of traceability models and their relation to other models. A GEF3D based tool queries the transformation chain model to find models and traceability models. Based on the traceability models, it can generate a consistent diagram layout for all models throughout the chain. Furthermore, it visualizes all these diagrams in a 3D model editor where also the traceability links themselves can be shown.

This integrated approach hence facilitates both the specification of transformation chains and their visualization. The visualization provides a concrete representation of otherwise abstract traceability information. It can for example be used to quickly localize a problem in the transformation chain in order to solve it locally in the identified sub-transformation.

We want to keep improving our approach in the future. UNITI will be extended so that traceability information of the whole chain can be accessed transparently by subsequent transformations. GEF3D will be further developed in order to provide full-featured editors, i.e. the diagrams should not only be visualized but also be editable in the 3D view. Besides, the usability should be improved, e.g. by highlighting search results (or all elements connected by a trace).

References

1. Vanhooft, B., Ayed, D., Baelen, S.V., 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)
2. Szyperski, C.: *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Professional, Reading (1997)
3. Jouault, F., Kurtev, I.: Transforming models with atl. In: Bruel, J.-M. (ed.) *MoDELS 2005*. LNCS, vol. 3844. Springer, Heidelberg (2006)
4. IBM Alphaworks: Model transformation framework. Misc (2004), <http://www.alphaworks.ibm.com/tech/mtf>
5. OMG: A Proposal for an MDA Foundation Model. Object Management Group, Needham, MA. ormsc/05-04-01 edn. (2005)
6. Vanhooft, B., Van Baelen, S., Joosen, W., Berbers, Y.: Traceability as input for model transformations. In: *Proceedings of the European Conference on MDA Traceability Workshop*, Nuremberg, Germany (2007)
7. Kruchten, P.: *The Rational Unified Process*. Object Technology Series. Addison-Wesley, Reading (2004)
8. von Pilgrim, J.: Mental map and model driven development. In: Fish, A., Knapp, A., Störle, H. (eds.) *Proceedings of the Workshop on the Layout of (Software) Engineering Diagrams (LED 2007)*. Electronic Communications of the EASST, vol. 7, pp. 17–32 (2007)

9. Eclipse Foundation: Graphical Editing Framework (GEF), Project Website (2008), <http://www.eclipse.org/gef>
10. Topcased: Topcased Tools, Project Website (2008), <http://www.topcased.org/>
11. Eclipse Foundation: Eclipse Modeling Framework (EMF), Project Website (2008), <http://www.eclipse.org/modeling/emf/>
12. von Pilgrim, J.: Graphical Editing Framework 3D (GEF3D), Project Website (2008), <http://gef3d.org>
13. Kleppe, A.: Mcc: A model transformation environment. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 173–187. Springer, Heidelberg (2006)
14. Marvie, R.: A transformation composition framework for model driven engineering. Technical Report LIFL-2004-10, LIFL (2004)
15. Object Management Group: Qvt-merge group submission for mof 2.0 query/view/transformation. Misc (2005)
16. Allilaire, F., Bezivin, J., Bruneliere, H., Jouault, F.: Global model management in eclipse gmt/am3. In: Thomas, D. (ed.) ECOOP 2006. LNCS, vol. 4067. Springer, Heidelberg (2006)
17. Salay, R., Chechik, M., Easterbrook, S., Diskin, Z., McCormick, P., Nejati, S., Sabetzadeh, M., Viriyakattiyaporn, P.: An eclipse-based tool framework for software model management. In: Eclipse 2007: Proceedings of the 2007 OOPSLA workshop on eclipse technology eXchange, pp. 55–59. ACM, New York (2007)
18. Gills, M.: Survey of traceability models in it projects. In: ECMDA-TW Workshop (2005)
19. Jouault, F.: Loosely coupled traceability for atl. In: Proceedings of the European Conference on Model Driven Architecture (ECMDA) workshop on traceability, Nuremberg, Germany (2005)
20. Barbero, M., Fabro, M.D.D., Bézivin, J.: Traceability and provenance issues in global model management. In: 3rd ECMDA-Traceability Workshop (2007)
21. Alfert, K., Engelen, F., Fronk, A.: Experiences in three-dimensional visualization of java class relations. SDPS Journal of Design & Process Science 5, 91–106 (2001)
22. Gil, J., Kent, S.: Three dimensional software modelling. In: 20th International Conference on Software Engineering (ICSE 1998), Los Alamitos, CA, USA, p. 105. IEEE Computer Society, Los Alamitos (1998)
23. Duan, C., Cleland-Huang, J.: Visualization and analysis in automated trace retrieval. In: First International Workshop on Requirements Engineering Visualization (REV 2006 - RE 2006 Workshop), Los Alamitos, CA, USA, vol. 5. IEEE Computer Society, Los Alamitos (2006)
24. Greevy, O., Lanza, M., Wysseier, C.: Visualizing live software systems in 3d. In: SoftVis 2006: Proceedings of the 2006 ACM symposium on Software visualization, pp. 47–56. ACM Press, New York (2006)

Towards Roundtrip Engineering - A Template-Based Reverse Engineering Approach

Manuel Bork, Leif Geiger, Christian Schneider, and Albert Zündorf

Kassel University, Software Engineering Research Group,
Wilhelmshöher Allee 73, D-34121 Kassel, Germany

Abstract. Model driven development suggests to make models the main artifact in software development. To get executable models in most cases code generation to a “traditional” programming language like e.g. Java is used. To obtain customizable code generation template-based approaches are applied, commonly. So, to adapt the generated code to platform specific needs templates are modified by the user. After code generation, in real world application the generated code is often changed e.g. by refactorings. To keep the code and the model synchronous reverse engineering is needed. Many approaches use a Java parser and a mapping from the Java parse tree to the UML model for this task. This causes maintenance issues since every change to a template potentially results in a change to this parse tree - model mapping. To tackle this maintenance problem our solution does not use a common language parser but uses the templates as a grammar to parse the generated code, instead. This way changes to the templates are automatically taken into account in the reverse engineering step. Our approach has been implemented and tested in the Fujaba CASE tool as a part of the model and template-based code generator CodeGen2 [11].

1 Introduction

Transforming models into various kinds of text languages is common practice, nowadays. These textual languages are used to provide an executable mapping for different kinds of models. Therefore such a model-to-text translation, to a programming or description language, is subject to optimization, adaption and maintenance work.

To obtain the required flexibility in the transformation engine, text templates are a common means to facilitate the final transformation from model-to-text, cf. e.g. [1,2]. These templates can easily be tuned, adapted and maintained. Even users can edit templates to change the transformation results according to their needs.

Transforming models to text is not the only direction that is needed. There are still developers who edit source code directly, there are processes requiring people to change text artifacts, and sometimes pieces of generated text may have lost there corresponding models. Thus transformation from text to models are a requirement as well.

Common techniques for transforming text to models utilize a parser for the specific text language and operate on the parse tree afterwards. This approach is limited by the flexibility of the language parser (problems like syntax errors in the text document) and it easily causes a maintenance problem if the templates used for to-text-transformation are changed. In contrast to the template editing for the forward engineering, the reverse engineering - parsing of text - cannot be configured easily. To overcome these weaknesses our text-to-model transformation engine exploits the very same templates used for the model-to-text direction for reverse engineering.

The work presented here has mainly been done in the context of the bachelor and master thesis of Manuel Bork [6,7].

2 Related Work

Template based text generation has established itself as a standard technique for web pages e.g. based on PHP or Java server pages. Consequently, the same technique is frequently used for model-to-text code generation for example in eclipse by Java emitter templates [2] or in Fujaba [10] by velocity [1].

For text-to-model or reverse engineering one commonly uses parsing technologies. This means, based on e.g. some Java grammar and e.g. a compiler compiler like `javacc/jjtree` [3,4] one builds an abstract syntax tree and this tree is then transformed with a model-to-model transformation into the original model. In the Fujaba project we have been following this approach for several years, too [14,17,19,16,18,20]. Especially, the thesis of Thomas Klein [13] created a first reverse engineering component with reasonable capabilities for Fujaba. However, due to several changes to our code generation strategies, e.g. for association implementation, we frequently had to update this reverse engineering component and after only one year, the functionality was lost. Another approach [20] tried to overcome the maintenance problems with fuzzy reasoning. They relaxed the exactness of the pattern matching process in order to deal with minor code variations. This of course had the drawback of false positives and false negatives.

Other reverse engineering approaches use fact extractors. Facts are tuples of entities (i.e. classes, variables, methods) and relations (i.e. inheritance, function calls, instantiations). A fact extractor for Java is introduced in [12].

In order to reverse engineer dynamic models other approaches are used. Briand et al. generate UML sequence diagrams by protocolling execution traces at runtime [8]. Rountev et al. however analyse directly the control flow in the source code [21].

With the Columbus framework [9] it is possible to reverse engineer even large C/C++ projects. It generates class diagrams, an abstract syntax tree and call graphs. Furthermore, it supports design pattern recognition. CPP2XMI [15] is based on Columbus and extracts UML class, sequence and activity diagrams.

Due to our knowledge, there is no other approach that exploits code generation templates for parsing directly. One may argue that template files form some kind of language grammar. However this template file based grammar will most

likely not fulfill the constraints of an LALR1 or LL1 grammar, cf. [5]. Thus one has to use more general techniques for context free grammars as e.g. the Cocke-Younger-Kasami (CYK) algorithm. However, our templates contain local variables, expressions, and control structures which again complicate matters.

3 Code Generation

A template based code generation is a prerequisite of our new reverse engineering approach. This section gives an overview of the code generation software used for our prototype. The code generation process is split into three tasks, cf. Figure 1.

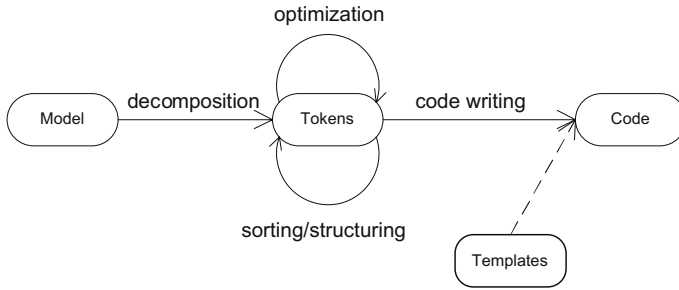


Fig. 1. Subtasks of the code generation with initial and resulting data

Our approach first transforms the original model into an intermediate token structure. This intermediate token structure defines a visiting order for model elements. This initial transformation step also handles a lot of conditional cases and alternatives, e.g. this step may choose a specific strategy for the implementation of associations. Note, one model element may create multiple tokens for different purposes, e.g. a model class may create a token structure for a Java interface class and another one for an implementation class. The result of the token creation task is a tree of tokens where the tokens may refer each other in several ways (thus forming a graph of tokens).

The generated token graph represents our intermediate language. The code generation for Fujaba’s graph rewrite rules performs additional structuring, sorting and optimizing on this intermediate language. Tokens from class diagrams usually need little to no further structuring.

Note, in reverse engineering, the token structure is extended by temporary string attributes that hold references to model elements. These string references are then resolved to real model references in a separate step.

In code generation, the final step generates code for the resulting token graph. Therefore, the underlying token tree is visited in postorder. Every visited token is passed to a chain of code writers. Usually, the responsible code writer opens a specific template file and passes the token and additional information as context to the template engine. This additional information includes the code generated for all children of the token in the tokens hierarchy.

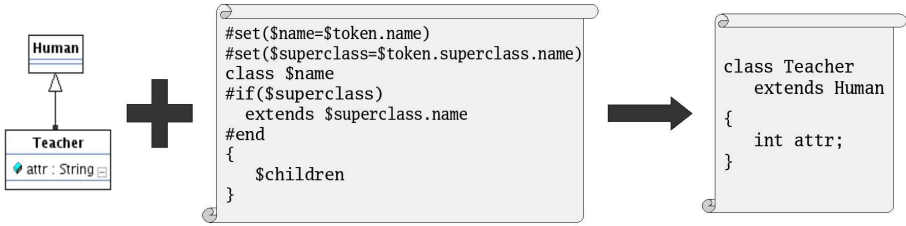


Fig. 2. Code generation example

In our implementation, we use the velocity template engine [1]. Figure 2 shows an example model and a simplified version of the template used for class tokens. Within the class template, the class token is stored in variable `$token`. From this variable, the template may access the original model via the token structure. For example, the first two lines of the template declare two local variables: `$name` and `$superclass`. The `$name` variable is read from the `name` attribute of the passed `$token` object. The `$superclass` variable gets its value via the access chain `$token.superclass.name`. Velocity allows attribute access or even method calls on every object passed to the template engine. That makes the templates highly customizable since every model element can be queried during code generation. In line 3 of the template finally code is generated: The constant string “class” followed by the current value of variable `$name` is produced as output. Velocity also allows control flow such as looping or branching. The code fragment in line 5 is only added to the output if the statement in line 4 evaluates to `true`. This means it is only added if the variable `$superclass` is not empty. After the opening bracket from line 7 the code for all subtokens (like methods or fields) is added to the class’ code. That code is passed to the engine in the `$children` variable. The code for the class is finished by the closing bracket from line 9.

Note, although the Velocity template engine provides control structures itself, we do most of the complex computation during the construction and optimization of the intermediate token structure. This reduces the complexity of our templates dramatically and makes them reasonably simple.

4 Reverse Engineering

It is common practice to use a parser which is based on the grammar of the target language to reverse engineer a piece of source code. But this common approach has several disadvantages: First, a separate parser for each language, which can be generated with the code generator, is needed. In contrast to that, it is sufficient to write a set of new templates for a template based code generator to support a new target language (with similar structure). Second, the parse tree - the result from parsing a piece of source code with a common language parser - is very fine grained. It is quite tedious to map this parse tree information to an application meta model (e.g. UML). To accomplish this task model elements

must be associated with more or less complex patterns, found in the parse tree. Afterwards a pattern recognition mechanism is used to identify pattern instances in the parse result and these matches are translated into model elements. As of the nature of the parse results these pattern need to be adapted according to the possible generated or hand written implementation style, found in the parsed source code.

In the template-based approach, presented in this paper, this mapping is still needed. Though, the result from the template-based parsing has a higher granularity and higher abstraction level than the parse tree from a common language parser. Thus the part of the pattern matching algorithms which is under major maintenance due to template- or code-style-changes is instead already covered in the template-based parsing.

Our text-to-model approach uses the very same templates that were used for code generation before. So, while adapting the templates for whatever reason one adapts the reverse engineering mechanism at the same time. This tackles the major part of the maintenance problem. Section 3 introduced our template based code generator. In a nutshell the application's model is transformed into an intermediate token tree and then passed to the template engine. The template engine generates code, specified in the the according template, afterwards. In Figure 2 an example for this procedure was given. For reverse engineering we invert this procedure. Figure 3 exemplifies the reversed procedure.

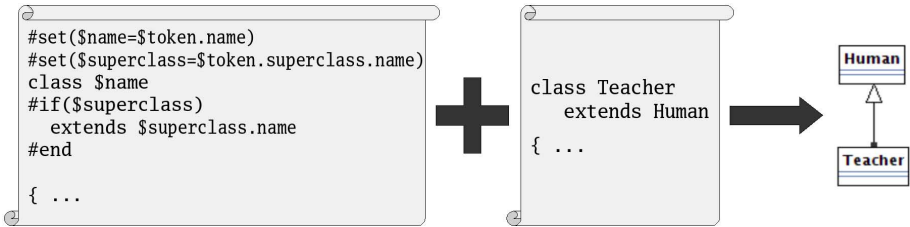


Fig. 3. Template and source code are parsed to reconstruct the model

Our approach works as follows. Given a piece of source code, the code generator states which template is used first. Then, while traversing template and source code the template's variables are assigned with textual fragments of the source code so that the given source code would be generated again. These assignments are utilized later on to reconstruct the intermediate token layer and finally the model itself. As there are several possible solutions for this first step some reasoning is performed next. Thereby boolean conditions that consist of multiple variables are split up and contradictory solutions are removed. Both tasks, template-based parsing and reasoning, are repeated for all included templates. After that the intermediate token layer is reconstructed generically. Then textual references are resolved to real objects and mapped to the model.

In the following we present each subtask in detail.

4.1 Template-Based Parsing

The goal of this subtask is to discover whether a given template has been used to generate the source code given and how the templates variables were assigned.

The left side of Figure 3 illustrates an excerpt of a template for a Java class. The right side shows two lines of source code. In order to reverse engineer this source code the values assigned to the template variables are inferred. So first of all the template itself must be parsed. Therefore the velocity template engine contains a template parser. This parser returns a parse tree that is traversed in the following step. To find the values in the example (i.e. for the classname `$name` and the superclass' name `$superclass`) the parser traverses the template and tries to match the terminals¹ of the template with text fragments from the source code, first. Thus, in the example, the parser starts at the beginning of the template and reads the terminal `class`. It finds the same text in the source code at the starting position, too. As this is a successful match, parsing process continues. After that the variable `$name` is processed. As the parser does not know, yet, which value is assigned to the variable, it just stores the variable name as pending. The next processed template part is a branching statement. As the parser does not know yet how to evaluate the branching condition (`$superclass`), it checks both possible paths through the template: a) assuming the condition to be true and b) assuming it to be false. So in case a) the parser tries to find the terminal `extends` in the source code. It is matched in the middle of the source code. It is now possible to assign a value to the pending variable: `$name` gets the assignment `Teacher`. After that the succeeding variable `$superclass` and the rest of the template is processed likewise.

By this manner the parser finds the following assignments for the variables from the template as a first possible solution: `$name` with `Teacher` and `$superclass` with `Human`. But one branch is still left to check. So the parser skips the complete branching statement in the other case (assuming `$superclass` to be false or empty) and tries to match the terminal `{`, which is found at the end of the source code snippet. So the parser can assign the previously read source code fragment `Teacher extends Human` to the variable `$name`. Obviously this assignment does not make much sense, but nevertheless it is a valid result concerning the parsing process. So this first subtask results in two possible solutions.

Trying all possible ways through the template is a very time consuming task. Additionally many possible assignments emerge, if a terminal occurs several times within a source code. For example, there are many opening braces in a piece of Java source code: This causes the same number of possible assignments for the first template variable, in the given example, as the number of opening braces in the source code. We address this problem by specifying constraints for the allowed values of variables. These constraints span from very simple constraints (i.e. "does not contain white spaces") to very expressive constraints (i.e. a complex regular expression). Commonly, most variables of a template cover only one single line of generated source code. Thus specifying a single line constraint as a default constraint for all variables of a template, helps decreasing

¹ Text fragments which are not substituted by the template engine.

the parsing time dramatically. In the example we would specify a constraint for both variables `$name` and `$superclass`, which denotes that the assigned value must not contain any whitespaces. This would discard the second solution.

In our implementation these constraints can be specified directly in the template. We introduced a simple comment syntax for this purpose. Figure 4 shows a constraint specification for the previous example.

```
## hints[ $name ] := SingleWordConstraint
## hints[ $superclass ] := RegularExpressionConstraint( "[^\s]+$" )
```

Fig. 4. Example of a specification of constraints for variables of a template

The constraint definition starts with the keyword `hints` followed by the name of the variable to constrain in brackets. An assignment operator follows and finally the a constraint name with parameters concludes the statement. With this syntax it is also possible to define multiple constraints for one variable by using `+=` as operator. If parameters are passed to the constraint, they are specified in braces after the name of the constraint surrounded by double quotes. In Figure 4 the second line shows such a case: A constraint specifying a regular expression that does not allow any whitespace characters within a string. The `SingleWordConstraint` in the upper line is a more convenient method for this same purpose.

4.2 Reasoning

After the subtask of parsing there are often multiple possible solutions how the template’s variables can be assigned with fragments of the source code. Several of these solutions might be contradictory if e.g. a variable is one time evaluated to `true` and another time evaluated to `false` in the same template application. The reasoning subtask addresses this issue by removing contradictory parse results. To retrieve all possible information complex expressions are used for reasoning, too. E.g. branching conditions that consist of multiple variables concatenated by an `AND` operator are split using constraint solving techniques. Additionally the reasoning subtask is responsible for the removal of local variables that do not originate from the context. The reasoning is discussed in this section.

Branching conditions often consist of multiple variables. We use a constraint solver to split up these compositions and infer the boolean value of previously not assigned variables wherever possible. The upper part of Figure 5 shows a template with two conditional statements. The parser result states that the first constant string “some text” was found but the second one “some other text” was not. So, the first condition was evaluated to `true` and the second one to `false`. This knowledge (shown in the lower left-hand site of Figure 5) is passed to the constraint solver. The right-hand site of the figure below shows the results returned by the constraint solver. The variable `$c` has to be `true` to fulfill the

```

    #if( ($a && $b ) || $c ) some text #end
    #if( $a ) some other text #end

```

<pre> (\$a && \$b) \$c == true \$a == false </pre>	\Rightarrow	<pre> \$c := true \$b := undef </pre>
--	---------------	---------------------------------------

Fig. 5. Above: Extract of a template. Left side: Conditions from the template and their boolean value, assigned to them by the parser. Right side: Values inferred by the reasoning mechanism.

constraints. For variable `$b` no information can be inferred. So, a new assignment `$c = true` is added to the set of assignments.

After the constraint solving is done, a check is performed that ensures that all assignments are consistent. First of all a variable that is only accessed read-only in the template (meaning there is no direct assignment to an value) must have the same source code fragment assigned each time it is used in the complete template. If a variable is assigned to a value (or another variable, or some calculated value), we distinguish if it is assigned only once before it is accessed or even afterwards. In the first case the variable is treated as if it was accessed read-only. In the second case the variable is marked as *mutable* and cannot be used for the reasoning subtask. If there are contradictory assignments the complete solution can be dismissed. This way the reasoning reduces the number of solutions - ideally only one solution will remain.

The last step of the reasoning is the removal of local variables. As many template languages, Velocity offers the possibility to specify local variables. Local variables are often used in templates e.g. to calculate a boolean value only once, if it is needed several times. While the parser only assigns these local variables with values, they should not be used to reconstruct the model, because the reconstruction of the model should not depend on implementation details of the templates. So, the reasoning mechanism has to remove the assignments of local variables. After the removal of the local variables, only the (textual) values of the attributes read from the intermediate layer are left. Figure 6 shows an example of this step.

While parsing the allocation of a local variable was found in the template. Then the parser was able to assign the local variable `$myLocal` with the logical value `true`. From the definition of the local variable `$myLocal` the reasoner is able to infer, that if `$myLocal` is `true`, both `$token.foo` and `$token.bar` must be `true`, too. So by reversing all allocation statements (`#set` directives in Velocity), local variable assignments are removed and the attribute values of the intermediate layer are inferred.

At the end of the reasoning subtask ideally one solution remains. But if the combination of template and source code is ambiguous, there may still remain several solutions. A simple example of this case is if there are two succeeding conditions with identical bodies but different condition statements. Then two valid solutions remain even after reasoning and it is not possible to say which solution has been used for code generation. Such problems should be avoided by

<pre>#set(\$myLocal = \$token.foo && \$token.bar) \$myLocal == true</pre>	\Rightarrow	<pre>\$token.foo := true \$token.bar := true</pre>
---	---------------	--

Fig. 6. Left side: Example of an allocation of a local variable within a template and an assignment found by the reasoner. Right side: Values inferred by the reasoning mechanism.

the template designer whenever possible. Anyhow, if several solution are found, our implementation needs user interaction to choose the right solution.

Parsing and reasoning are not necessarily subsequent tasks. In fact it makes much sense to combine both tasks for performance reasons. Since the reasoning excludes possible solutions, excluding those as soon as possible can really speeds up the parsing process. E.g. if the parser has assigned one variable and finds another value for the same variable later on, the current solution can be skipped and the parser does not have to match neither the rest of the current template nor nested templates.

4.3 Creating Tokens

After the subtask of reasoning there are lists of assignments for each pair of template and source code. These assignments are key-value assigning attributes of the intermediate layer of tokens to text extracted from the source code. Though it is possible to generate the model directly from the information included in these assignment, we create an intermediate layer first. This intermediate layer can be generated generically. The advantage of this approach is that the updating/creating of the model can now be described as a mapping between two models.

So, this subtask addresses the issue of creating an intermediate layer of tokens. Each token represents a single code fragment, for example an attribute of a class or a method declaration. So, there exists a token type for every template. As described in section 3, the token layer is linked as a tree: Source code generated for each parent token contains the source code that is generated for its child tokens. This structural information is obtained from CodeGen2.

The reconstruction of the token layer is done as follows: For each solution (consisting of template, source code, and assignments) a token object is created and then linked with previously reconstructed tokens according to the structural information given by CodeGen2. Afterwards the textual assignments are mapped to attributes of the created token. Figure 7 shows an example of this procedure.

The left side of figure 7 shows the assignments found for a template representing an attribute. So a token of type **AttributeToken** is created and linked in the tree of previously created tokens. Then the fields of the attribute token are accessed via reflection: The field **name** gets the value "example", **visibility** the value "public" and **type** gets **int**. In this way the complete layer of token is created.

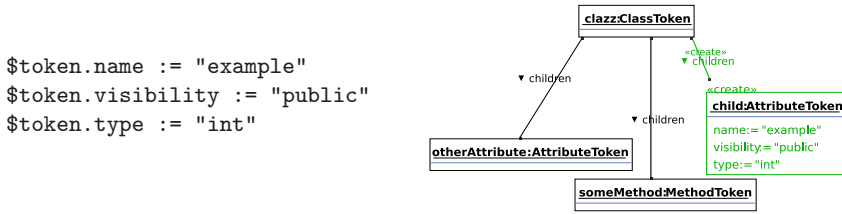


Fig. 7. Left side: Textual assignments found for a template of an attribute. Right side: Object diagram representing the token layer.

At the end of this task all textual assignments are transferred into the intermediate layer of tokens.

4.4 Updating of the Model in the CASE Tool

The last step is now to create/update the model in the application itself. Therefore the intermediate layer of tokens is traversed and the model is adapted accordingly. This cannot be done generically, because the UML model is a complex graph and not a plain tree like the token layer. So, there is a strategy for each token type that updates the model. Even in case of reverse engineering, when the existing model is empty, it is not sufficient to let the strategies simply create all model elements. In fact it is necessary to search the model element first, because it could have been created by another strategy before. Figure 8 shows an example of this situation.

In this example two classes with one connecting association have been parsed and the intermediate token layer was created. The model mapping mechanism starts with the package token and continues with either of both class tokens, for example with the one generated for the class **Teacher**. So a class **Teacher** is created in the model, if it does not yet exist. The next token in the token tree is the role token **toN**. In order to create the complete association in the model the association's target role and the corresponding class need to be created, too. It is known that the role's class must have the name **Course**, because this is the Type of the field in class **Teacher**. So **Course** is created in the model, then both roles and finally the association itself. These steps are done by the strategy for role tokens. Later on, the other class has to be mapped to the model. Because the class **Course** already exists, it is not created but updated. The class' visibility and some other properties are set that were not known while creating the class. The same is done for the other role-token which is processed next: As there already exists a role connected to an association with the name **gives** it must only be updated. After this task the model represents the parsed source code.

Note that association detection normally is a complex task because one must determine if two roles belong to the same association. As we generate the association's name into the source code (as comment of the roles) we can easily identify both roles belonging to one association.

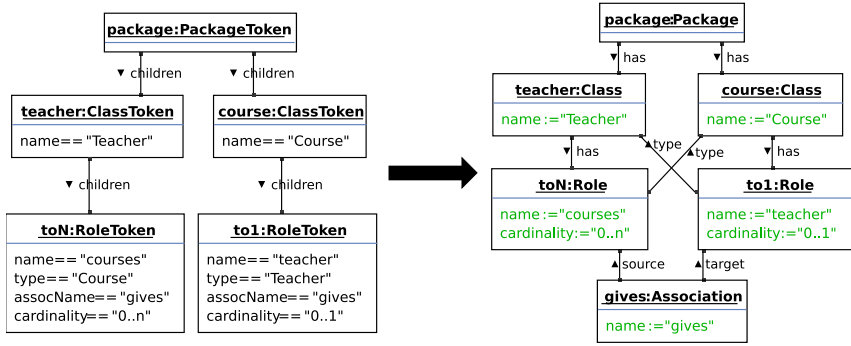


Fig. 8. Object diagram of token layer (left side) and corresponding model

4.5 Dealing with Nested Templates

In section 4.1 we introduced the parsing as the first step of reverse engineering. There we assumed that the template to use is predetermined. But in fact there are often several templates to be considered. Furthermore the code generator nests multiple templates to generate one single source code file. As described in section 3 the code generator traverses the token tree inorder and starts generating code for the leaves of that tree. The code generated for all child token is passed when generating code for their parent token which usually includes the child code in the parent code. In our implementation the parent templates include the children's code in the template variable `$children`.

So, the `$children` variable needs special treatment when reverse engineering a piece of source. It is not sufficient if one single template matches² a piece of source code, but potentially embedded templates have to fit, too. It is possible that a parent template matches but the children don't. So if the parser assigned some value to the variable `$children`, one or more child templates must fit at least once.

Figure 9 shows an example for nested templates.

The start template is predetermined by the code generator. So the parser starts parsing the complete Java class with the template for Java classes and assigns the class' body to the template's variable `$children`. Then multiple templates have to be considered: As a Java class can contain methods and attributes, the parser attempts to match the body with both templates. As in the second step the method template matches - which includes other templates, too - the parser attempts to parse the methods body before continuing parsing the rest of the class' body. In the example the template that matches the rest of the class' body is the template for attributes.

In general the order in which the parser attempts to match templates to source code does not matter. But if the parser tries to match the subsequent templates

² Matching in the sense of sections 4.1 and 4.2.

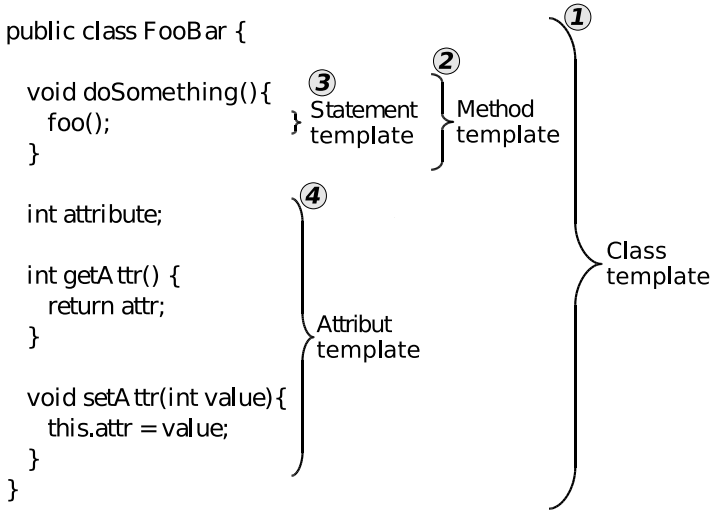


Fig. 9. A Java class and corresponding templates. The numbers identify the order in which the parser tries the nested templates. Note that the parser attempts to match all suitable templates in each step.

before he matches the child templates successfully (in the example steps three and four would be exchanged), the parsing time increases. As a method template only consists of a method declaration, opening braces, the `$children` variable and closing braces, this template is able to match less or more source code than one complete method. For example starting at the method declaration and ending at some closing braces in the middle of the method. As this does not make much sense the complete solution should be discarded as soon as possible and a longer match should be tried. But if the parser instead tries to match subsequent templates parsing time increases. So our parser attempts to match child templates before matching subsequent templates.

5 Conclusion

This paper presents an approach for reverse engineering of code generated by a template based code generator. Our approach uses the templates as a “grammar” to parse the given code. Thus, we have build some kind of compiler compiler. However, since our template based “grammars” do not conform to LL1 or LALR1 grammar restrictions, our approach has to fight several performance issues. With the help of some template extensions, we have achieved a reasonable performance for our examples. This still needs improvement.

The main advantage of our approach is that the reverse engineering mechanism is language independent since it does not rely on a specific language parser. Additionally, the very frequent changes that are made to the templates by our

developers are instantly taken into account by the reverse engineering component. Only if our meta model or the structure of our intermediate token layer changes, we have to adapt the model access parts of our templates. However, compared to the templates, our meta model and the token layer are quite stable. Thus, we have reduced the maintenance problem of keeping forward and reverse engineering synchronous, dramatically.

Note, if a template is modified, our parsing approach is adapted, instantly. Thus it is instantly able to recognize code generated with the new templates. However, it may now fail to recognize code generated with the earlier version of the template. Thus, in future work we will keep the earlier versions of our templates and use those as fallbacks if the new version fails.

Using our new approach, we observed that the template based parsing approach facilitates the recognition of quite complex Java code structures, dramatically. For example, our code generation implements a to-n association with the help of a container attribute and about 11 access methods. Using the old pattern matching on the Java parse tree it was quite tedious to identify all these parts of an association implementation, correctly, cf. [17]. Within our association template all these access methods are listed in a row. Thus, the association template provides exactly the pattern required to recognize all the access methods during parsing. However, this has the drawback, that the template expects the access methods in the given order and with exactly the given implementation. If e.g. some IDE reorders the methods, our recognition will fail. In our experiences this did not turn out to happen (to us) in practice.

We found that using the templates for parsing can result in a very slow parser since the possibilities of template applications can easily explode. Since the templates form some kind of a context free grammar, ideally, the generated parser should achieve a worst-case runtime complexity of $O(n^3)$ as the CYK algorithm. While this is already pretty inefficient, the CYK algorithm requires normalized grammar rules much simpler than our template structures and our parser faces the additional task of resolving template control structures and variable assignments. To overcome the performance problems, we add constraints to the templates that reduce the state space. An open problem is to what extent such constraints may be inferred directly from the meta-model. Additionally, to speed up the parsing, our reasoning step excludes paths from the state space, as soon as possible.

We have implemented our approach as a part of the code generation of the Fujaba Tool Suite [10,6,7]. We are now able to reliably reverse engineer every code generated by our code generation. In current work, we address manual changes to the source code. If such changes obey the coding rules or our templates, our reverse engineering works fine. However, a simple `System.out.println` in an attribute access method may suffice to disable the recognition of the corresponding template. Similarly, manual declarations of attributes or manual implementations of associations will most likely not be recognized by our usual template based parser. To address such manual code, we have added so called “legacy

templates” to our reverse engineering component. These legacy templates cover all basic Java elements. We use these legacy templates as fallbacks if the usual code generation templates fail. The result of such a legacy template recognition is usually rather low level, e.g. an attribute of some container type instead of a to-n association to some user defined class. However, we are able to reverse engineer every possible Java source code.

References

1. Velocity (2006), <http://velocity.apache.org/>
2. Java Emitter Templates Tutorial (2008),
http://www.eclipse.org/articles/Article-JET/jet_tutorial1.html/
3. JavaCC (2008), <https://javacc.dev.java.net/>
4. JJTree Reference Documentation (2008),
<https://javacc.dev.java.net/doc/JJTree.html>
5. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers, Principles, Techniques, and Tools. Addison-Wesley (1986)
6. Bork, M.: Reverse Engineering generierten Quelltexts durch Analyse von Velocity Templates. Master’s thesis, Kassel, Germany, Diploma I Thesis (2007)
7. Bork, M.: Reverse Engineering von Legacy Code: Optimierung des template-basierten Reverse Engineerings zu einem transparenten und flexiblen Erkennungsmechanismus. Master’s thesis, Kassel, Germany (2007)
8. Briand, L., Labiche, Y., Miao, Y.: Towards the Reverse Engineering of UML Sequence Diagrams. *wcre* 0, 57 (2003)
9. Ferenc, R., Magyar, F., Beszedes, A., Kiss, A., Tarkiaainen, M.: Columbus - Reverse Engineering Tool and Schema for C++. *icsm* 00, 172 (2002)
10. Fujaba Group. The Fujaba Toolsuite (1999), <http://www.fujaba.de/>
11. Geiger, L., Schneider, C., Record, C.: Template- and modelbased code generation for MDA-Tools (2005)
12. Kaastra, M., Kapser, C.: Toward a semantically complete java fact extractor. Department of Computer Science, University of Waterloo (April 2003)
13. Klein, T.: Rekonstruktion von uml aktivitats- und kollaborationsdiagrammen aus java quelltexten. Master’s thesis, Paderborn University (1999)
14. Klen, T., Nickel, U.A., Niere, J., Zündorf, A.: From uml to java and back again. Technical Report tr-ri-00-216, University of Paderborn, Paderborn, Germany (September 1999)
15. Korshunova, E., Petkovic, M., van den Brand, M.G.J., Mousavi, M.R.: CPP2XMI: Reverse Engineering of UML Class, Sequence, and Activity Diagrams from C++ Source Code. In: *WCRE*, pp. 297–298. IEEE Computer Society (2006)
16. Nickel, U.A., Niere, J.: Modelling and simulation of a material flow system. In: *Proc. of Workshop 'Modellierung' (Mod)*, Bad Lippspringe, Germany, Gesellschaft für Informatik (2001)
17. Nickel, U.A., Niere, J., Wadsack, J.P., Zündorf, A.: Roundtrip engineering with fujaba. In: Ebert, J., Kullbach, B., Lehner, F. (eds.) *Proc of 2nd Workshop on Software-Reengineering (WSR)*, Bad Honnef, Germany, Fachberichte Informatik, Universität Koblenz-Landau (August 2000)
18. Niere, J., Schäfer, W., Wadsack, J.P., Wendehals, L., Welsh, J.: Towards pattern-based design recovery. In: *Proc. of the 24th International Conference on Software Engineering (ICSE)*, Orlando, Florida, USA, pp. 338–348. ACM Press (May 2002)

19. Niere, J., Wadsack, J.P., Wendehals, L.: Design pattern recovery based on source code analysis with fuzzy logic. Technical Report tr-ri-01-222, University of Paderborn, Paderborn, Germany (March 2001)
20. Niere, J., Wendehals, L., Zündorf, A.: An interactive and scalable approach to design pattern recovery. Technical Report tr-ri-03-236, University of Paderborn, Paderborn, Germany (January 2003)
21. Rountev, A., Volgin, O., Reddoch, M.: Control flow analysis for reverse engineering of sequence diagrams. Technical Report OSU-CISRC-3/04-TR12, Ohio State University (March 2004)

Annotation Framework Validation Using Domain Models

Carlos Noguera and Laurence Duchien

Université des Sciences et Technologies de Lille,
INRIA Lille Nord-Europe - LIFL
40, avenue Halley, 59655 Villeneuve d'Ascq, France
{noguera, laurence.duchien}@lifl.fr

Abstract. Frameworks and libraries that use annotations are becoming popular. However, there is not enough software engineering support for annotation development. In particular, the validation of constraints in the use of annotations requires further support. In this paper we postulate that annotation frameworks are a projection of the domain model into a programming language model. Using this idea, we have developed a tool that allows the annotation programmer to specify, and then validate the constraints of the annotation framework regarding a given annotated application using a domain model. To validate our approach to the validation of constraints using models, we apply it to the Fraclet annotation framework and compare it to the previous implementation.

1 Introduction

Large annotation frameworks [6,16] are becoming more and more common. These kinds of frameworks, such as EJB3 [12], offer to the application programmer, in addition to classes and methods, annotations that provide an additional, declarative way to use the framework. Annotation framework design and implementation rises a number of challenges. Among them, the problem of validating that the application developers correctly use the annotations. It is interesting for the framework programmer to be able to express the constraints of its annotation framework, and to automatically check whether a program is valid with respect to these constraints.

Previously, we have developed a technique and a tool to express these constraints, called AVal, that relies on meta-annotations. Constraints are then implemented by the use of a meta-annotation framework, and are checked by an annotation processor. In this paper, we extend AVal by investigating the relationship between annotation frameworks and domain models. Based on this relationship, we show that the constraints of an annotation framework can be translated into constraints on a domain model. Furthermore, the validation of an annotated program corresponds to the validation of the instance of the domain model. We apply this technique to a case studie: Fraclet.

The paper is organized as follows: first, in the next section we motivate the need for annotation validation, and present our existing approach called AVal. Then, in Section 3 we present our proposal, by discussing the relation between annotations and models, and their usefulness in annotation validation. In Section 5 we present how models aid in the validation of a real-life annotation framework, Fraclet. Finally, in Sections 6 and 7 we compare our work to similar approaches, and conclude.

2 Annotation Validation

The Java type system for annotations is not expressive enough to assure that the use of annotations is correct. This type system allows the annotation framework developer to define the names, types and default values of (optional) properties, as well as the Java program elements to which it can be attached. It, however, leaves the responsibility of more complex checks to the annotation framework developer.

Complex annotation frameworks impose further restrictions on the use of annotations than those made available by the Java compiler. For example, in EJB3, the `@Id` annotation that marks a field in an entity class as its identifier, can only be placed in fields belonging to a class annotated as `@Entity`. Constraints such as these are common among annotation framework specifications. These kinds of constraints cannot be enforced by the Java compiler, and it is up to the annotation developer to check them as part of the annotation's processing phase.

Annotation frameworks imply a number of constraints on their usage. This is not different than for any other framework. However, in contrast to regular frameworks, annotation frameworks are static entities; that is, their usage can be checked during the compilation of the program. This is done so that the errors are provided to the final developer as soon as possible. Given the static nature of the semantics of annotations, their constraint checking is considerably easier than that of its regular counterparts because, in general, no complex static analysis must be performed.

We call the process of constraint checking *validation of an annotated program*. This process takes as inputs the set of annotation types and a program carrying the corresponding annotations. As output, a set of errors corresponding to the violations of the constraints as they are used in the program are returned. In this process we identify two actors: the developer of the annotation framework, i.e., the person that implemented the annotation types; and the program developer, i.e., the person that wrote and annotated the program.

Although the process flow for the validation of an annotated program is straightforward, the constraints actually checked strongly depend on the particular annotation framework. Each annotation framework imposes its particular set of constraints that derive from the domain in which they lay. In general, annotation validations are of two kinds, those dealing with the relationship between an annotation type and the code element on which it is placed, and those dealing the annotation type's properties, and its relationship with other annotation types. We call the former *code-wise validations*, while the later, *structural validations*.

2.1 AVal - Annotation-Based Validation

To perform the validation of annotation frameworks AVal [13] applies the concept of annotations itself by defining an annotation framework that contains a set of meta-annotations for the domain of *annotation constraints validation*. These validation meta-annotations are used to augment the definition of the annotation framework under development with meta-data relevant to validating a given constraint.

Annotating annotations with other annotations has the advantage to make the constraints *explicit* in the annotation framework's definition and *local* to the annotation they

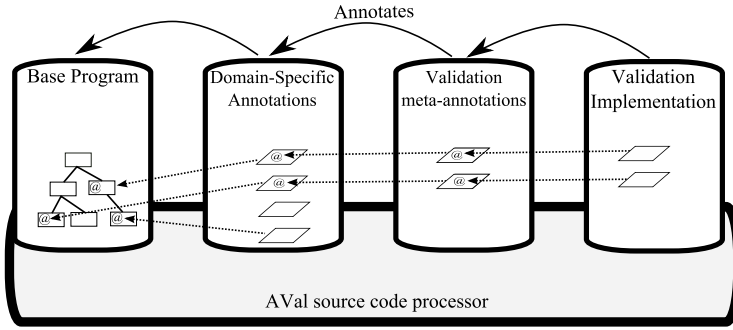


Fig. 1. Aval Architecture

apply to. Aval provides a number of built-in validations, and means to add custom ones. It uses Spoon [14] for compile-time reflection and annotation processing, and through it, provides integration to the Eclipse IDE.

2.2 Annotation Validation for Java

The concept of using meta-annotations to declare restrictions on use of Java annotations is already included in the JDK. Indeed, the Java Language Specification [10] defines a `Target` annotation that must be placed on annotation type definitions to restrict where instances of the annotation type can be placed. However, besides from `Target`, no other validation annotations are provided.

Aval's architecture is composed of four layers (Figure 1):

Base program: The (annotated) program that is to be validated. Elements of the program are annotated by annotations defined on the annotation framework layer.

Domain-Specific (Annotation) Language: The domain specific annotations. Each annotation is meta-annotated by an Aval meta-annotation that expresses the constraints for its validation.

Validation meta-annotations: Aval annotations that encode the constraints to validate domain specific annotations. Each meta-annotation represents a validation constraint, and is itself annotated with the class that is responsible for the implementation of it.

Implementation: A class per validation meta-annotation. The class must implement the `Validator` interface, and it uses the Spoon compile-time model of base the program, annotation framework, and meta-annotation in order to perform the validation.

Aval is implemented as a Spoon source code pre-processor that is executed before the code generation or compilation phase in an annotation framework. It traverses the base code looking for domain-specific annotations. Each time it finds an annotated element, it checks the model of the annotation's declaration to see if it has any meta-annotations. In case the annotation has one or more validators, the tool executes each implementation in the order in which they are defined. In order to ease the specification of constraints,

Table 1. Default constraint annotations in AVal

Annotation	Description
<code>Inside(AT)</code>	The annotation must be placed on elements which are inside elements annotated with <i>AT</i>
<code>Requires(AT)</code>	The annotation requires that the target of the annotation also is annotated with <i>AT</i>
<code>RefersTo(AT, N)</code>	The property of the annotation must carry the same value as the property called <i>N</i> belonging to <i>AT</i>
<code>AValTarget(TE)</code>	The annotation must have as target the program element <i>TE</i>

AVal provides a number of annotations that represent commonly used rules. A subset of these annotations, as well as their description is shown in Table 1, for a more complete discussion on AVal's annotations see [13].

In order to validate complex constraints, the annotation framework developer can extend AVal with new meta-annotations. For this, a new annotation type and its corresponding implementation (see Figure 1) must be provided. The annotation type serves to mark the context of the constraint, while the actual checking must be performed by traversing the AST of the program. This traversal, of course, requires an intimate knowledge of the model used by the tool to represent the program, and its associated API. Hence, the creation of new annotations for AVal can be a tedious task. In order to ease the extension of AVal's annotations, an abstraction over the AST of the program is needed. Annotation models is one possible abstraction.

3 Annotation Models

As we have seen in the previous sections, complex annotation frameworks require validations that concern both other annotations and the program on which they are used. But, where do these constraints come from? Consider the `Inside` validation, if an annotation type *A* is required to reside inside another one, *B*, this implies a relationship between them since it makes no sense for *A* to be present in the program without its corresponding *B*. Now, suppose that both *A* and *B* are classes in an UML class model, then the relationship induced by the `Inside` validation could be described by means of a containment association between them.

3.1 Annotation and Code Models

Extending this idea of *modeling* annotation types, we can see that structural validations can be mapped to relationships and invariants on a model that represents the annotation types. We will call this model derived from the annotation types an *annotation model*.

Code-wise validations, on the contrary, cannot be described in terms of the annotation model alone, since they deal with constraints on the relationship between the annotations and the program on which they are imposed. To integrate code-wise validations into the model we need a representation of the target language, in this case Java, so that an association between annotation models and code models can be reasoned on.

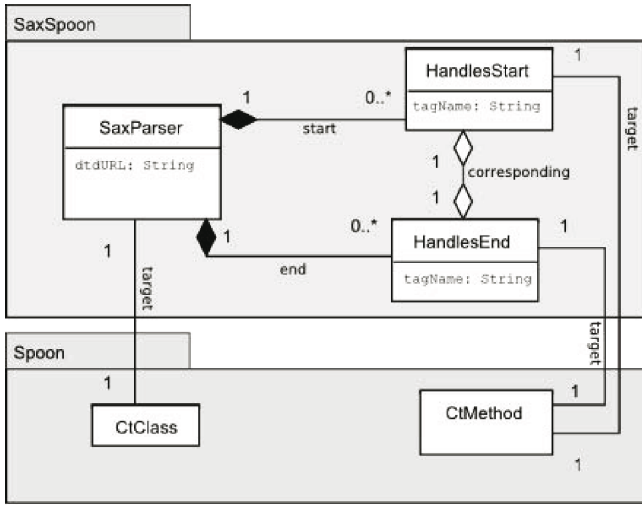


Fig. 2. SAXSpoon annotation model

This model representation of the target language, called *code model*, needs to be at the same level of abstraction as that of the annotation model to be able to mark the references from one to the other, i.e. to state that a given annotation is to be placed on a given code element. This code model is the AST of the language.

In this way, the annotation model corresponds to the model of the domain, defined by the annotation types and their corresponding constraints, while the code-wise validations define a *mapping* between the annotation model and the code model. Now, just as the annotations in the program conform to their annotation types, there is a model instance for both the annotation model and the code model. Code-model instances would represent the concrete syntax tree of a given program; while annotation-model instances would represent the annotations present in the target program.

3.2 Example

To better understand the nature of the annotation models and their interaction with the code model, let us suppose a simple annotation framework to define SAX-based XML parsers called SAXSpoon. SAX frameworks traverse the tags of an XML document up-calling a method each time a start or end tag is found. In our annotation framework, we will define three annotations: SAXParser, HandlesStart and HandlesEnd. SAXParser identifies a class as being a SAX parser and it defines a single property that points to the DTD document that defines the type of documents that the class will handle. HandlesStart and HandlesEnd respectively identify the methods that handle the start and end of a tag, given as parameter to each annotation. The corresponding annotation model for this framework is depicted in Figure 2. In it, The package SaxSpoon contains the annotation model, while the package Spoon contains the relevant parts of the code model.

Code-wise constraints for SAXSpoon can be encoded as OCL expressions on the relations between the *SaxSpoon* and *Spoon* packages; while structural constraints can be encoded in the relationships between the elements of the annotation model itself.

4 Validation Using Annotation Models

As discussed before, it is possible to embed constraints on the annotation and code model. These constraints are then checked against a model instance derived from an annotated program. In order to do this, the annotation developer must be able to declare the constraints on its annotation framework, and she must be able to direct the way in which the annotation model instance is generated. For this, we have extended *AVal* (as presented in Section 2.1) with annotations to specify the instantiation of the model (*Association* and *DefaultValue*) as well as the constraints on the model (*OCLConstraint*). The implementation for the aforementioned annotations and its corresponding tool chain is called *ModelAn*, and will be explained in the following Section.

4.1 ModelAn - Model Based Annotation Validation

ModelAn is a tool chain for the definition of annotation model constraints and their corresponding validation. It is driven by annotations (as opposed to models), and it uses *AVal* as an underlying layer. The workflow for the use of *ModelAn* is depicted in Figure 3. It starts from the annotation types defined as part of the framework by the annotation framework developer. The set of annotation types carry annotations that direct the *Model Extraction* engine in producing the annotation model and its corresponding *Model Instanciator*. The program written by the application developer is then fed to it, producing an annotation model instance that conforms to the annotation model extracted before. Using both the model and its instance, *ModelAn* uses a constraint checker to validate the program, and report back to the application developer any violations. The whole process is transparent to the application developer.

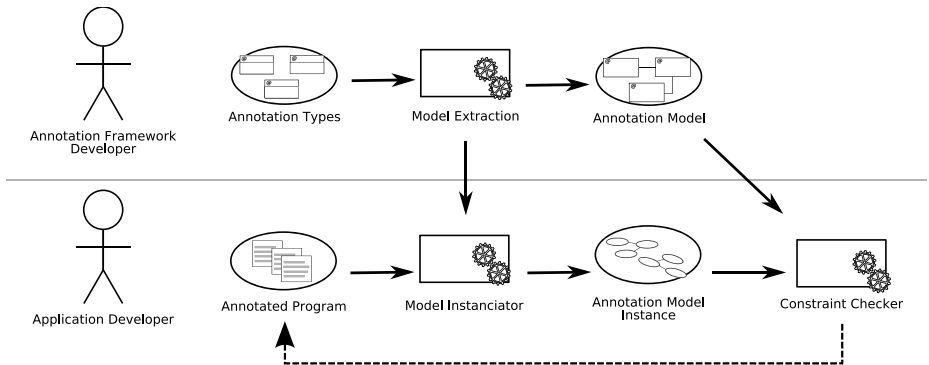


Fig. 3. ModelAn process flow

Model Extraction

The annotation model is extracted from the set of annotation types that compose the framework. As a starting point, each annotation type is represented as an element of the model with its corresponding attributes. In addition to this, each element in the model is associated with the code element which it is supposed to augment. This association is called the *target* of the annotation.

The model is then augmented by the annotation framework developer using two annotations on the annotation types: `Association` and `DefaultValue`

Association. Associations define the structural relations between annotations. An association must define a name, a type and a defining query. The OCL query is evaluated in the context of the annotation type on which it is placed, and can only reason on associations on the code model because it itself is defining the associations on the annotation model. For example, in the SAXSpoon annotation framework, there is a relation between a `SaxParser` and its start and end handlers. Therefore, the definition of the `SaxParser` annotation type would be as follows:

```
@Association(name = "Start",
             type = HandlesStart.class,
             query =
"HandlesStart.allInstances()->select(self.target.Methods->includes(target))")
public @interface SaxParser {
    String.dtdURL() default "";
}
```

In this example, the query traverses all the `HandlesStart` elements, looking for those which are placed on methods which belong to the class annotated with `SaxParser`. Hence, this query constructively defines the relation *start*. A similar construction is used to define the relation between `SaxParser` and `HandlesEnd`

DefaultValue. Attributes in annotations often have default values. In the general case, the default value is a static value (for example the empty string), but in some cases, the default value depends on the place in which an annotation is placed. For example, suppose that the name of the tag that a method handles is by default the name of the method. In this case, the default value cannot be known when the annotation type is defined, since it will change depending on the use of the annotation. The annotation framework developer can then state, using an OCL query, what the default value of the property should be. In the case of `SaxSpoon`, the definition of the `HandlesStart` would be:

```
public @interface HandlesStart{
    @DefaultValue("self.target.SimpleName")
    String.tagName();
}
```

Model Constraint Definition

Once the annotation model has been defined, the developer can define the constraints on it. In order to do this, `ModelAn` defines a single annotation, `OCLConstraint` that is to be placed on the annotation type. The constraint is represented by an OCL expression that is evaluated in the context of the annotation model element that corresponds to the current annotation type.

OCLEnstraint. OCL expressions placed on annotation types can use the associations defined by the `Association` annotation to express the constraints of the annotation framework. The `OCLEnstraint` annotation is an `AVal` annotation (see Section 2.2) that defines a single property that contains the expression itself. In `SaxSpoon`, the annotation framework developer may want to specify a constraint stating that a warning should be raised if a Sax parser handles the `Start`, but not the end of a given tag. For this, a constraint must be placed in the *corresponds* relation:

```
@Association(name = "corresponds",
             type = HandlesEnd.class,
             query =
"HandlesEnd.allInstances()->select(handler|handler.tagName = self.tagName)")
@OCLEnstraint("self.corresponds->size() = 1")
public @interface HandlesStart {
    String tagName();
}
```

In this example, a *corresponds* association is defined using the first `Association` annotation, and the second `OCLEnstraint` annotation places an OCL constraint that uses it to specify that there should be a single corresponding tag handler for the same tag.

Using the information defined by the `Association`, `DefaultValue` and `OCLEnstraint` annotations, the model extraction engine generates an `Ecore` file that contains the annotation model. The `Ecore` annotation model references the `SpoonEMF` [2] `Ecore` model that represents the Java programming language, i.e. the code model. The resulting annotation model for our running example is shown in Figure 4. The OCL queries that define the associations in the model are saved in this file by means of `ecore-annotations` on the references, while an annotation on the element states the annotation type that this element models. The OCL constraints are not included in the model itself.

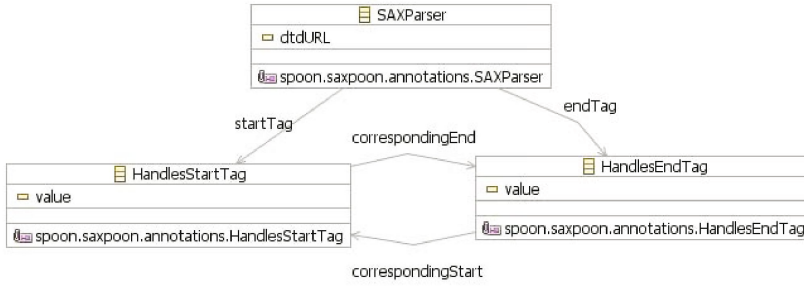
Finally, the model extraction engine will generate a set of `Spoon` source code processors that will instantiate the annotation model and the code model of a given program.

Model Instantiator

The annotation model is instantiated by a source code processor generated by the model extraction engine. The source code processor traverses the AST of the target program, and using the `Ecore` file that contains the annotation model, creates an instance of the corresponding element for each annotation it encounters. Once all the annotations in the program have their corresponding instance, the source code processor executes the OCL queries present in each association in order to populate them. At the end of the process, an in-memory instance of the annotation model that represents the annotated program is available. This instance can then be used to check the annotation framework constraints.

Constraint Checker

The constraints themselves are checked using `AVal`. The `AVal` source code processor passes over the program after the model instantiator. Each time an annotation with an



```

@Associations({
@Association(name="startTag",
  type = HandlesStartTag.class,
  query= "HandlesStartTag.allInstances()->"+
    "select(self.target.Methods->includes(target))"),
@Association(name="endTag",
  type = HandlesEndTag.class,
  query= "HandlesEndTag.allInstances()->"+
    "select(self.target.Methods->includes(target))"),
})
public @interface SAXParser {
    String dtdURL() default "";
}

@Associations({
@Association(name = "correspondingEnd",
  type = HandlesEndTag.class,
  query = "HandlesEndTag.allInstances()->"+
    "select(handler|handler.value = self.value)")
})
@OCLConstraint("self.correspondingEnd->size() = 1")
public @interface HandlesStartTag {
    String value();
}

@Associations({
@Association(name = "correspondingStart",
  type = HandlesStartTag.class,
  query = "HandlesStartTag.allInstances()->"+
    "select(handler|handler.value = self.value)")
})
@OCLConstraint("self.correspondingStart->size() = 1")
public @interface HandlesEndTag {
    String value();
}

```

Fig. 4. SAXspoon Ecore Model and Annotated types

OCL constraint is found, the OCL expression is evaluated on the model's instance. If the expression evaluates to `false`, an error is raised. Since the constraint checker uses `AVal`, the errors are presented to the programmer in the same format as Java compiler errors (see [13]).

Although the process to validate the constraints of a program starting from the annotation framework, all the way to obtaining the errors may seem long, it is important to note that each of the actors see only a one-step process. Indeed, for the annotation framework developer, only the model extraction step is necessary, while for the application developer, the model instantiation and constraint checking are a step of the compilation process, and therefore transparent. In terms of advantages, the use of models to define the constraints of an annotation framework allows the annotation framework developer to abstract away from the code model, and reason about the relations in the domain model itself. In the SaxSpoon example, this is evident in the constraint that establishes the correspondence between start and end handlers. In this constraint, the annotation framework developer does not refer to any code element in the constraint's expression, reasoning instead only on the actual domain of the annotation framework. It is also important to note that this additional abstraction level comes at no cost to the final application developer, since as we pointed out before, the constraint checking is hidden behind the compilation of the program. The framework developer, in contrast, is required to manipulate OCL expressions, which can be complex at times, to define the associations and constraints of the annotation framework. In order to reduce the use of OCL we expect to leverage UML's stereotypes as a way to graphically specify the annotation model. This is further discussed in Sections 6 and 7.

5 Case Study - Fraclet

Fraclet is an annotation framework for the Fractal component model [3]. The Fractal component model defines the notions of *component*, *component interface*, and *binding* between components. Each of these main notions is reflected in the annotation framework defined by Fraclet. There are two implementations of Fraclet [15], one using XDoclet2, and the other one using Java5 annotations and Spoon annotation processors. The annotations defined by Fraclet/Spoon are summarized in Table 2.

In Figure 5, Fraclet/Spoon is used to augment a Java class in order to represent a Fractal primitive component. The `Client` class uses a `Component` annotation to represent a component called `helloworld.Client` that provides a single interface named `r`. Fields of this class are marked as attributes, required ports or controller hooks. Finally, a method on the component is marked as a life-cycle handler.

In order to define the constraints of each of these annotations, we have applied the meta-annotations defined in Section 4.1 to extract an annotation model that represents Fraclet. The resulting model is shown in the Figure 6.

Once, the model was defined, we discussed with the Fraclet developers in order to learn the constraints that a correct Fraclet application must adhere to. Then we translated the constraints to their corresponding OCL expressions. These are the constraints and their corresponding translations:

Table 2. Overview of Fraclet annotations

Annotation	Location	Parameter	Description
Component	Class	<i>name</i>	Annotation to describe a Fractal component.
Interface	Interface	<i>name, signature</i>	Annotation to describe a Fractal business interface.
Attribute	Field	<i>argument, value</i>	Annotation to describe an attribute of a Fractal component.
Required	Field	<i>name, cardinality, contingency</i>	Annotation to describe a binding of a Fractal component.
Lifecycle	Method	<i>value</i>	Annotation that marks a method as a life-cycle callback. The parameter specifies the step of the life-cycle.
Controller	Field	<i>value</i>	Annotation that marks a field as an access point to the component's reflective services

```

@Component(name = "helloworld.Client",
           provides = @Interface(name = "r",
                                signature = Runnable.class))
public class Client implements Runnable {

    private final Logger log = getLogger("client");

    @Attribute(value="Hello world") private String message;
    @Requires(name="s")             private Service service;
    @Controller("name-controller") protected NameController nc;

    @Lifecycle(CREATE) protected void whenCreated() {
        log.info("helloworld.Client - created.");
    }

    public void run() {
        this.service.print(this.message);
    }
}

```

Fig. 5. Client Comoponent Fraclet Implementation

A Component's name must be unique in the application. By default, the name of a component is the simple name of the class on which the Component annotation is placed. This is expressed using a DefaultValue annotation. The definition of the component annotation is as follows:

```

public @interface Component {
    @Default("self.target.SimpleName")
    @OCLConstraint("Component.allInstances()->"+
        "select(c:Component| c <> self and c.name = self.name)->isEmpty()")
    String name() default "";
    //...
}

```

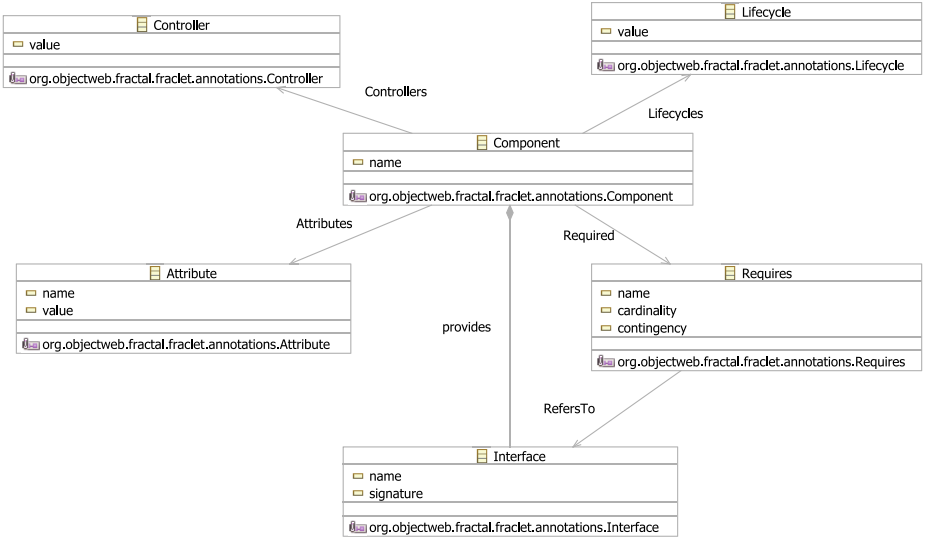


Fig. 6. Fraclet Annotation Model

A Field on a Component cannot be at the same time Attribute and Required

```

@OCLConstraint("Requires.allInstances()->"+
  "forAll(r:Requires|r.target <> self.target)")
public @interface Attribute {
  //...
}

```

A Required Interface must be defined. The name of the required interface is by default the name of the field on which it is placed.

```

@OCLConstraint("self.RefersTo->size() = 1")
@Association(type = Interface.class, name="RefersTo",
  query="Interface.allInstances()->select(i|i.name = self.name)")
public @interface Requires {
  @Default("self.target.SimpleName") String name() default "";
}

```

The previous version of Fraclet (studied in [13]) used Aval and needed six different annotations to perform the same tests we have implemented here with only OCLConstraint. In addition to this, we were able to elegantly specify the default values for the names of the Component and Attribute annotations, which was not addressed in the previous Aval-based version.

6 Related Work

Related work in annotation framework validation and development can be aligned along two axes: the development and validation of annotation frameworks and the relationship between annotations and models.

Annotation Validation. The need of validating annotations has been previously addressed by academia. First, in [5] Cepa et. al. propose a mechanism to validate the use of custom attributes (similar to annotations in the .NET platform) by placing custom attributes on the definition of other custom attributes. This is quite similar to the approach proposed in AVaI, however, their technique only allows for the definition of structural constraints, and no mechanism to extend the constraints is provided. Also, they provide no explicit code or annotation model. In [7], Eichberg et.al. propose to validate structural annotation constraints in Java programs by representing a program as an XML document, and representing the constraints as XPath queries. In their approach, the XML schema of the document acts as an implicit code model, but they do not provide an explicit annotation model. The lack of this model complicates the definition of constraints, since no relation exists between annotations.

Finally, annotations are extensively used for the validation of programs [8,9,11]. However, this use of annotations differs from our intention, since our focus is the validation of the use of annotations themselves, not of the program on which they are placed.

Annotations and Models. Annotations, seen as meta-data attached to a code entity, are semantically close to stereotypes as defined in UML 2.0 [1]. Indeed, it is common to represent annotations, during design, as stereotypes [4]. Nevertheless, it is difficult to establish a direct mapping between stereotypes and annotations given the particularities of annotations. For example, annotations do not allow for inheritance, an annotation can be placed on different code elements (stereotypes are restricted to one), and most importantly, annotations can refer to types that are defined in the program in which they are applied since for example, annotations can contain as a property enums defined in the program. This last characteristic is the most problematic, since it places annotation models somewhere in between levels M1 and M2. Nevertheless, it seems possible to construct a mapping between UML profiles and annotation models. This will be the subject of future work.

The use of models for the development of annotation-based programs is explored in [16] by Wada et. al. They propose a full MDA approach that starts from a model, and ends with an executable program. However, they start from the idea that the annotation framework already exists, and therefore, provide no support for the development of it. We believe their proposal and ours to be complementary.

7 Conclusion

We presented a way to specify and validate constraints on annotation frameworks based on models. As an implementation, we introduced ModelAn, an annotation framework that allows the annotation framework developer to define an annotation model and attach constraints to it. ModelAn also constructs a source code processor that generates instances of the annotation model in order to validate the constraints. The use of OCL constraints offers the developer a greater degree of expressiveness when defining new kinds of constraints with respect to the extensibility options of AVaI. In addition to this, the use of a code model and an annotation model provides an abstraction over

the direct manipulation of the AST of the program, which results in more concise constraints.

The use of models to define the constraints of an annotation model allows the annotation framework developer to abstract away from the code model, and reason about the relations in the domain model itself. Using OCL also provides a declarative way to express these constraints, and diminishes the prerequisite knowledge of the underlying AST API that the annotation developer must have. These two characteristics make ModelAn a more extensible annotation framework validation platform than Aval.

As future work, we believe that the relation between annotations and models can be further exploited. In particular, we are working on a model-directed approach that allows us to define an annotation framework from a set of UML stereotypes and their corresponding constraints. This will be done by implementing a model-to-model transformation that goes from a profile to an annotation model, and then to the actual implementation. Also, annotation models can prove to be of aid in the understanding of an annotated program, since they make explicit the relation between annotations on it. Furthermore, if a program uses different annotation frameworks to implement different concerns on different domains, then each associated annotation model will provide a domain specific view of the program.

References

1. OMG Unified Modelling Language Infrastructure (OMG UML) V2.1.2 (November 2004), <http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF>
2. Barais, O.: SpoonEMF, une brique logicielle pour l'utilisation de l'IDM dans le cadre de la réingénierie de programmes Java5. In: Journées sur l'Inénierie Dirigée par les Modèles (IDM) (June 2006) Poster
3. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.-B.: The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Softw. Pract. Exper.* 36(11-12), 1257–1284 (2006)
4. Cepa, V., Kloppenburg, S.: Representing Explicit Attributes in UML. In: 7th International Workshop on Aspect-Oriented Modeling (AOM) (2005)
5. Cepa, V., Mezini, M.: Declaring and enforcing dependencies between.NET custom attributes. In: Karsai, G., Visser, E. (eds.) GPCE 2004. LNCS, vol. 3286, pp. 283–297. Springer, Heidelberg (2004)
6. Cisterino, A., Cazzola, W., Colombo, D.: Metadata-driven library design. In: Proceedings of Library Centric Software Development Workshop (October 2005)
7. Eichberg, M., Schäfer, T., Mezini, M.: Using Annotations to Check Structural Properties of Classes. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 237–252. Springer, Heidelberg (2005)
8. Evans, D., Guttag, J., Horning, J., Tan, Y.M.: A tool for using specifications to check code. In: Proceedings of the ACM SIGSOFT 1994 Symposium on the Foundations of Software Engineering, pp. 87–96 (1994)
9. Evans, D., Larochelle, D.: Improving security using extensible lightweight static analysis. *IEEE Software* 19(1), 42–51 (2002)
10. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification, 3rd edn. Addison-Wesley, Reading (2005)
11. Hedin, G.: Attribute extensions - a technique for enforcing programming conventions. *Nord. J. Comput* 4(1), 93–122 (1997)

12. Michel, L.D., Keith, M.: Enterprise JavaBeans, Version 3.0. Sun Microsystems, JSR-220 (May 2006)
13. Noguera, C., Pawlak, R.: AVal: an extensible attribute-oriented programming validator for java. *Journal of Software Maintenance and Evolution* (July 2007)
14. Pawlak, R., Noguera, C., Petitprez, N.: Spoon: Program analysis and transformation in java. Technical Report 5901, INRIA (May 2006)
15. Rouvoy, R., Pessemier, N., Pawlak, R., Merle, P.: Using attribute-oriented programming to leverage fractal-based developments. In: *Proceedings of the 5th International ECOOP Workshop on Fractal Component Model (Fractal 2006)*, Nantes, France (July 2006)
16. Wada, H., Suzuki, J.: Modeling turnpike frontend system: A model-driven development framework leveraging UML metamodeling and attribute-oriented programming. In: Bruel, J.-M. (ed.) *MoDELS 2005*. LNCS, vol. 3844, pp. 584–600. Springer, Heidelberg (2006)

Prototyping Visual Interpreters and Debuggers for Domain-Specific Modelling Languages

Daniel A. Sadilek and Guido Wachsmuth

Humboldt-Universität zu Berlin

Unter den Linden 6

10099 Berlin, Germany

{sadilek, guwac}@informatik.hu-berlin.de

Abstract. This paper is about visual and executable domain-specific modelling languages (DSMLs) that are used at the platform independent level of the Model-Driven Architecture. We deal with DSMLs that are new or evolve rapidly and, as a consequence, have to be prototyped cheaply. We argue that for prototyping a DSML on the platform independent level, its semantics should not only be described in a transformational but also in an operational fashion. For this, we use standard modelling means, i.e. MOF and QVT Relations. We combine operational semantics descriptions with existing metamodel-based editor creation technology. This allows for cheap prototyping of visual interpreters and debuggers. We exemplify our approach with a language for Petri nets and assess the manual work necessary. Finally, we present *EProvide*, an implementation of our approach based on the Eclipse platform, and we identify missing features in the Eclipse tools we used.

1 Introduction

Developing a software system for a specific domain requires knowledge from the practitioners of that domain, the so-called *domain experts*. In the *Model Driven Architecture* (MDA), domain experts should be involved in creating the *computation independent model* (CIM) [1]. The CIM captures the requirements for the system—the “know-what”. Additionally, domain experts do also have knowledge about how the system can fulfil the requirements—the “know-how”. This knowledge is needed for the *platform independent model* (PIM). A PIM can be expressed in a general-purpose modelling language like the UML. But domain experts like seismologists or meteorologists are not used to the modelling concepts and notation used in the UML.¹ In contrast, a *domain-specific modelling language* (DSML), specific for the application domain, provides domain experts with concepts they know and with a special notation that matches their intuition. Thus, DSMLs allow domain experts to provide their know-how on the PIM level.

¹ UML can be customised with its profiling mechanism. However, this is not adequate in all cases because a UML profile can only introduce new concepts as specialisation of existing UML concepts.

A DSML can have a very narrow application domain. When building a new system, no existing DSML may be appropriate and a new one may be necessary. Usually, the concepts the new DSML should offer are not clear in the first place and multiple development iterations are necessary. Furthermore, when requirements for a system change, the DSML may need to be adapted. Therefore, a prototyping process for the DSML is needed. To evaluate prototypical versions of a DSML, the domain expert should be able to *use* the DSML: that is to create example models expressed in the DSML and to execute them.

To enable *model creation*, editors for DSMLs can already be generated from a declarative description [2]. The typical way in MDA to *execute PIMs* is to translate them (by model transformation or code generation). But such a translation does not have a proper level of abstraction for prototyping: language semantics is intermingled with platform-specific details. This inhibits understanding, troubleshooting, and adaptation of evolving language semantics—especially for domain experts.

Therefore, we argue that for prototyping a DSML, its semantics should be described on the platform independent level in an operational fashion. For this, we use standard model transformation techniques. By combining this approach with existing metamodel-based editor creation technology, we enable cheap prototyping of visual *interpreters* and *debuggers*.

In the following section, we introduce necessary vocabulary and technology. We present our approach and exemplify it with a language for Petri nets in Sect. 3. In Sect. 4, we present *EProvide*, an Eclipse-based implementation of our approach. In Sect. 5, we show the manual work necessary for developing a Petri net debugger and we discuss missing features we encountered in Eclipse EMF, GMF, and in the QVT implementation we use. We discuss related work in Sect. 6 and conclude in Sect. 7.

2 Preliminaries

2.1 Model-Driven Language Engineering

In the MDA, it is common practice to specify modelling languages by modelling means. A metamodel is an object-oriented model of the abstract syntax of a modelling language. With its *Meta Object Facility* (MOF) [3], the OMG provides standard description means for metamodels.

Example 1 (Petri net metamodel). Figure 1(a) provides a MOF compliant metamodel for Petri nets. A Petri net consists of an arbitrary number of places and transitions. Transitions have input (src) and output places (snk). Places are marked with a number of tokens. Places and transitions can have names.

Model transformations take a central place in MDA. They are used to translate models to other models, e.g. PIMs into PSMs (*platform-specific models*). Often they are used to translate a model into an executable form, e.g. to Java. From a

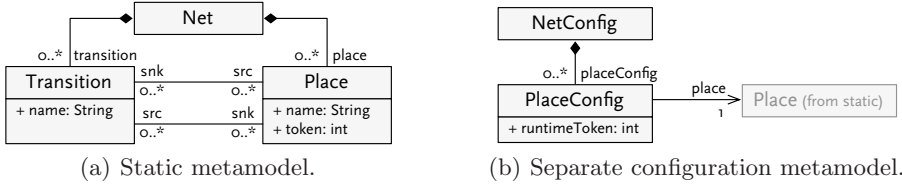


Fig. 1. Petri net metamodels

language perspective, model transformations define translational semantics for a modelling language.

In this paper, we use the high-level declarative language QVT Relations for transformations. It is part of the OMG standard Query/View/Transformation (QVT) [4] and heavily relies on OCL. In QVT Relations, a *transformation* consists of queries and relations. A *query* returns the result of an OCL expression. A *relation* relates model elements in the domains of the transformation by patterns. All relations of a transformation which are declared as *top* need to hold. If a relation does not hold, the transformation tries to satisfy it by changing model elements in domains declared as *enforce*. Relations can have two kinds of clauses: *Where clauses* must hold in order to make a relation hold. A *where* clause can contain OCL queries and invocations of other relation. *When clauses* act as preconditions. A relation must hold only if its *when* clause already holds.

Example 2 (Queries and relations in QVT Relations). Figure 2 shows a transformation in QVT Relations. It relates the two domains **input** and **output**, which both contain Petri net models. The transformation contains the queries **isActivated** and **getActivated**. The first query returns whether a given transition in a Petri net is activated. For this, it checks if all the transition's input places are marked with tokens. The second query returns an activated transition of a given Petri net. Here, the OCL predicate **any** ensures a non-deterministic choice.

Furthermore, the transformation contains several relations. The relation **run** relates a Petri net model from the **input** domain with a Petri net model from the **output** domain. The first pattern of **run** binds the variable **net** to a Petri net in the input domain. The second pattern *enforces* the same Petri net to occur in the output domain, i.e., it will be created if it not already exists.

The relation obtains an activated transition of the net by calling the query **getActivated**. To make the relation **run** hold, the unary relation **fire** needs to hold. **fire** simply checks if the transition can be found in the input domain.

The first relation **run** is declared as *top*. Therefore, it has to hold to process the transformation successfully. Since the relation **fire** is not declared as *top*, it only has to hold to fulfil other relations calling it from their *where* clauses. We will discuss the remaining relations **produce**, **consume**, and **preserve** later on.


```

transformation petri_sos(input:petri, output:petri) {

  top relation run {
    trans: Transition;
    checkonly domain input net:Net{};
    enforce domain domain output net:Net{};
    where { trans = getActivated(net); fire(trans); }
  }

  query isActivated(trans: Transition): Boolean {
    trans.src -> forAll(place | place.token > 0)
  }

  query getActivated(net: Net): Transition {
    net.transition -> any(trans | isActivated(trans))
  }

  relation fire {
    checkonly domain input trans:Transition{};
  }

  top relation produce {
    checkonly domain input place:Place{
      src = trans:Transition{}, token = n:Integer{}
    };
    enforce domain output place:Place{ token = n+1 };
    when { fire(trans); trans.src -> excludes(place); }
  }

  top relation consume {
    checkonly domain input place:Place{
      snk = trans:Transition{}, token = n:Integer{}
    };
    enforce domain output place:Place{ token = n-1 };
    when { fire(trans); trans.snk -> excludes(place); }
  }

  top relation preserve {
    checkonly domain input place:Place{ token = n:Integer{} };
    enforce domain output place:Place{ token = n-1 };
    when { not produce(place, place); not consume(place, place); }
  }
}

```

Fig. 2. Operational semantics for Petri nets

2.2 Structural Operational Semantics

The operational semantics of a language describes the meaning of a language instance as a sequence of computational steps. Generally, a transition system $\langle \Gamma, \rightarrow \rangle$ forms the mathematical foundation, where Γ is a set of *configurations* and $\rightarrow \subseteq \Gamma \times \Gamma$ is a *transition relation*.

Plotkin pioneered this approach. In his work on structural operational semantics [5], he proposed to describe transitions according to the abstract syntax of the language. This allows for reasoning about programs by structural induction and correctness proofs of compilers and debuggers [6]. The structural operational semantics of a language defines an abstract interpreter for this language working on its abstract syntax. It can be used as a reference to test implementations of compilers and interpreters.

3 Platform Independent Model Semantics

3.1 Abstract Interpretation

In this paper, we apply the idea of structural operational semantics to model-driven language engineering. For this, we rely on standard modelling techniques only: MOF and QVT Relations.

We represent the configurations in Γ as models, which we call *configuration models*. Hence, we define the space of all possible configurations with a metamodel, which we call *configuration metamodel*; and we define the transition relation \rightarrow with a model-to-model transformation, which we call *transition transformation*.

For some simple languages, computational steps can be expressed as mere syntactic manipulation of the program/model. A well-known example for this is the lambda calculus [7]. Another example are Petri nets.

Example 3 (Syntactic manipulation of Petri nets). A configuration of a Petri net is simply its current marking. Therefore, we can use the static DSML metamodel from Fig. 1(a) as configuration metamodel, as well. A computation step in a Petri net chooses an activated transition non-deterministically and fires it. The marking of a place connected to the fired transition is

- (i) increased by one token iff it is only an output place,
- (ii) decreased by one token iff it is only an input place,
- (iii) preserved otherwise.

The transformation given in Fig. 2, specifies these semantics in QVT Relations. It contains the relations **run** and **fire** as well as the queries **isActivated** and **getActivated**, which we already discussed in Ex. 2. The relations **produce** and **consume** adapt the token count of places that are connected to fired transitions:

- (i) The relation **produce** matches a place **place** in the input, an incoming transition **trans** of this place, and its number of tokens **n**. It enforces an increase of one in the number of tokens in the output. The relation needs to hold only if the matched transition is fired and if the matched place is not an input place of this transition.
- (ii) The relation **consume** works similarly. It matches a place in the input, an outgoing transition of this place, and its number of tokens. The number of tokens in the output is decreased by one. The relation has to hold only if the transition is fired and if the place is not an output place of the transition.
- (iii) If neither **produce** nor **consume** hold for a place, its token count is preserved by the relation **preserve**.

Describing operational semantics by mere syntactic manipulation works only for simple languages. In general, runtime states cannot be expressed with instances of the static language metamodel. Hence, a separate configuration metamodel is needed to represent runtime states. In addition, an *initialisation transformation* is needed that creates an initial configuration from a static model.

```

transformation init(net: petri, config: petriCfg) {
  top relation initNet {
    checkonly domain net n:Net{};
    enforce domain config nc:NetConfig{net = n};
  }

  top relation initPlace {
    checkonly domain net n:Net {place = p: Place{token = i:Integer{}}};
    enforce domain config nc:NetConfig{
      placeConfig = pc:PlaceConfig{place = p, runtimeToken = i}
    };
    when { initNet(n, nc); }
  }
}

```

Fig. 3. QVT transformation to initialise the runtime configuration of a Petri net

Example 4 (Advanced semantics of Petri nets). The Petri net semantics presented in Ex. 3 destroys the initial marking of a Petri net model by the execution. A separate configuration metamodel solves this problem (Fig. 1(b)). To preserve the initial marking of a Petri net, we distinguish **Net** and its configuration **NetConfig**. A net configuration contains a configuration **PlaceConfig** for each place in the net. The marking of a place is stored in the attribute **PlaceConfig.runtimeToken**.

With a separate configuration metamodel, we need an initial configuration to run the net. The QVT transformation given in Fig. 3 specifies this initialisation. It enforces a net configuration to contain a configuration for each place in the net. For a place configuration, the attribute **runtimeToken** is initialised with the initial marking from the attribute **token** of the configured place. Furthermore, we need to adapt the semantics description to act on the configuration instead of the net. We will see the adapted description in Ex. 6.

3.2 Visual Interpretation

Given a visualisation of the runtime state, a visual interpreter comes for free: PIMs are executed stepwise with the transition transformation and each step can be visualised. We give the user the option to control how many execution steps are performed before the visualisation gets updated. Thus, one can control how fast the interpretation process is animated.

Because we represent the runtime state of a PIM (its current configuration) as a model, it can easily be visualised reusing existing metamodel-based technology for creating model editors. There are two options for visualising runtime elements in a visual interpreter:

- (i) runtime elements are visualised as additional graphical entities,
- (ii) runtime elements affect the visualisation of static model elements, e.g. existing labels are extended with runtime values or the colour of existing graphical entities is controlled by runtime values.

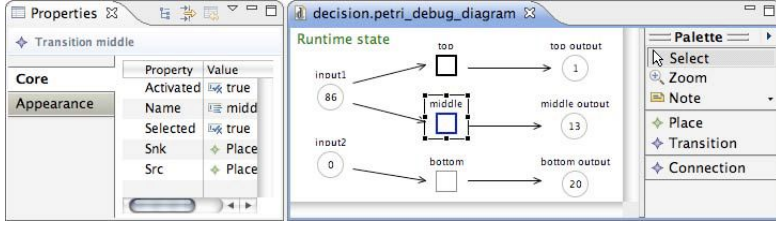


Fig. 4. Debugger for the Petri net DSML prototyped with *EProvide*

Example 5 (Petri net visualisation). Figure 4 shows an editor for the runtime state of a Petri net. Following Petri net conventions, circles and boxes represent places and transitions respectively. Incoming and outgoing arcs determine the input and output places of transitions. A number inside a circle shows the runtime marking of the place—not the initial (static) marking. This corresponds to option (ii).

3.3 Visual Debugging

Debugging means to control the execution process and to access and possibly modify the runtime state of a program. In usual debuggers, a program can be executed stepwise and breakpoints can be set to halt the execution at a specified point of execution. Whenever a program is halted, the user can access and modify variable values.

With our approach, a PIM can be executed stepwise by setting the number of execution steps to one. Further control on the execution, like support for breakpoints, can be achieved by extending the configuration metamodel with elements for controlling the interpretation process and adapting the transition transformation to use these elements. Accessing and modifying runtime state between interpretation steps is possible via the model editor that visualises the configuration model.

As for visual interpretation, visual debugging is achieved by applying existing editor creation technology.

Example 6 (Debugging Petri nets). When a user debugs Petri nets, he wants to know if a transition is currently activated or not. Furthermore, he wants to control which transition fires in the next execution step. We extend the configuration metamodel from Fig. 1(b) with a class `RuntimeTransition` that includes appropriate attributes. One is the derived attribute `activated`, which indicates the activation of a transition. We specify its value with an OCL query, similar to the body of `isActive` given in Fig. 2. Additionally, we introduce the attribute `selected`, which can be set by the user to select transitions for execution.

Breakpoints are another mandatory feature for debugging. For Petri nets, we introduce two kinds of breakpoints in our metamodel referring places and transitions respectively. A breakpoint for places defines a number of tokens.

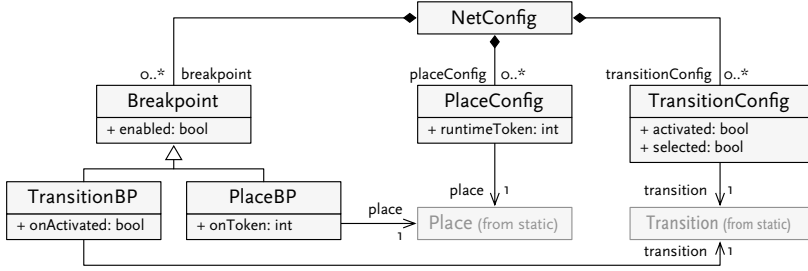


Fig. 5. Configuration metamodel extended with additional classes and attributes for debugging

If the place is marked with the defined number of tokens, the execution will be interrupted. A breakpoint for transitions refers to a transition and defines whether it should become activated or deactivated in order to interrupt the execution. Both kinds of breakpoints can be disabled or enabled.

Figure 5 shows the configuration metamodel extended for debugging. It includes the class **TransitionConfig** as well as the classes **Breakpoint**, **PlaceBP**, and **TransitionBP** for breakpoints. To achieve the expected behaviour of the debugger, we adapt the transition transformation. The adapted version is shown in Fig. 6. The new relations **breakPlace** and **breakTransition** check whether enabled breakpoints trigger an interruption of the execution. Furthermore, the relation **run** is only established if no breakpoint meets its breaking condition. To fire only selected transitions, the query **getActivated** is redefined to use **TransitionConfig**’s new attributes **activated** and **selected**. The query **isActivated** is obsolete. All other relations remain unchanged.

4 Implementation

4.1 Base Technologies

In this section, we describe the implementation of our approach. We based our implementation on Eclipse modelling technologies. As a metamodeling framework, we use the Eclipse Modeling Framework (EMF) [8]. In EMF, metamodels need to comply to the Ecore meta-metamodel which is very similar to Essential MOF. Based on the Ecore-based metamodel of a DSML, Java code for a *DSML plugin* can be generated with EMF. This plugin provides the infrastructure to create, access, modify, and store models that are instances of the DSML.

For the creation of graphical editors, we use Eclipse’s Graphical Modeling Framework² (GMF). In GMF, an editor is described with a set of declarative models, which we call *editor definition* in the following. Amongst other things, those models define which graphical elements are used to represent which elements of the DSML metamodel. The editor definition is used by the GMF code

² <http://www.eclipse.org/gmf/>

```

transformation petri_debug(input:petriCfg, output:petriCfg) {

  top relation breakPlace {
    checkonly domain input nc:NetConfig{
      breakpoint = bp:PlaceBP{
        enabled = true, onToken = n:Integer{}, place = p:Place{}
      },
      placeConfig = pc:PlaceConfig{place = p, runtimeToken = n}
    };
  }

  top relation breakTransition {
    checkonly domain input nc:NetConfig{
      breakpoint = bp:TransitionBP{
        enabled = true, onActivated = b:Boolean{}, transition = t:Transition{}
      },
      transitionConfig = tc:TransitionConfig{transition = t, activated = b}
    };
  }

  top relation run {
    trans: Transition;
    checkonly domain input nc:NetConfig{};
    enforce domain output nc:NetConfig{};
    when { not breakPlace(nc); not breakTransition(nc); }
    where { trans = getActivated(nc); fire(trans); }
  }

  query getActivated(netCfg: NetConfig): Transition {
    netCfg.transitionConfig -> any(selected & activated).trans;
  }

  top relation produce {
    checkonly domain input placeCfg:PlaceConfig{
      place = place:Place{ src = trans:Transition{} },
      runtimeToken = n:Integer{}
    };
    enforce domain output placeCfg:PlaceConfig{ runtimeToken = n+1 };
    when { fire(trans); trans.src -> excludes(place); }
  }
  ...
}

```

Fig. 6. Operational semantics for debugging Petri nets

generator to generate Java source code for a *graphical editor*. The generated code can be modified by hand afterwards. As we will see in Sect. 5, this was necessary for the Petri net example.

For model-to-model transformations, we use ikv's medini QVT³, an implementation of QVT Relations that can work with Ecore compliant metamodels.

4.2 EProvide

EProvide⁴ is an implementation of our approach. It is an Eclipse plugin that plugs into the Eclipse execution infrastructure to make domain-specific

³ <http://projects.ikv.de/qvt>

⁴ Eclipse plugin for PROtotyping Visual Interpreters and DEbuggers; available at <http://eprovide.sf.net>

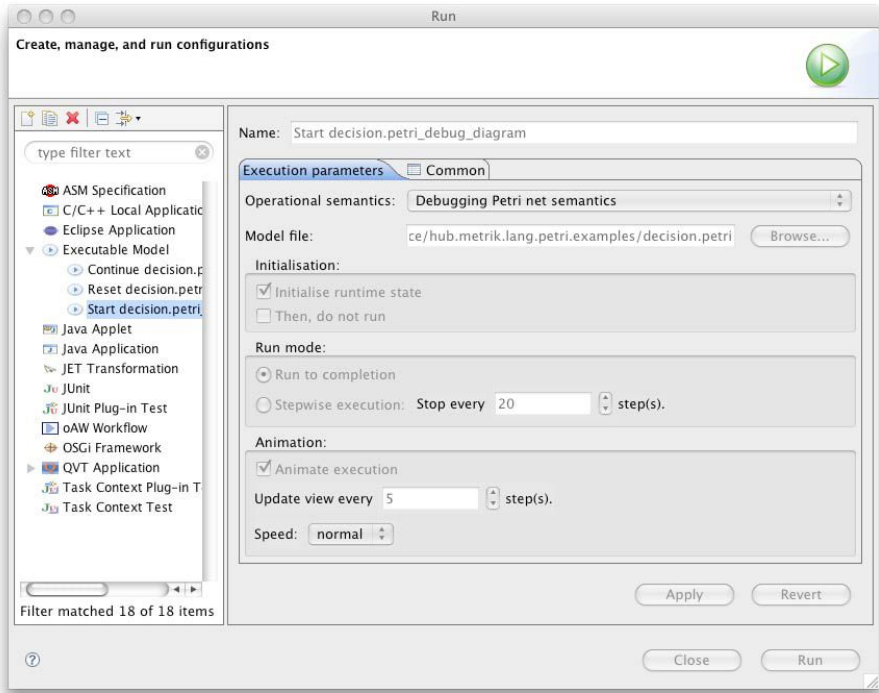


Fig. 7. *EProvide*’s run dialogue

PIMs—or domain-specific models in general—executable. It provides an extension point that DSML plugins can use to register themselves with *EProvide*.

The Language User Perspective. *EProvide* offers a run dialogue in which the user can configure model execution (Fig. 7): The user can choose whether the model should be initialised and whether it should be executed to completion or stepwise. If he selects “Run to completion”, the transition transformation gets repeatedly executed until the model reaches a fixed-point state in which it persists. If he selects “Stepwise execution”, the execution stops after the given number of steps (if no fixed-point state is reached before). Furthermore, *EProvide* allows to animate the execution process. The user can activate and configure this feature in the run dialogue, as well. He can specify how often the editor gets updated (e.g. every 5-th transformation step) and how fast the animation is (i.e. how long each visualised configuration is shown). If multiple semantics are available for a DSML, the run dialogue allows the user to select which semantics should be applied.

The Language Engineer Perspective. To achieve execution support for a new DSML, the DSML plugin has to provide an extension for *EProvide*’s extension point. In this extension, the following information is given: a name for the semantics that is shown in the run dialogue, the file extension of the model files that the semantics should be applied to, the path to the QVT files with the

transition and the initialisation transformation, and the full qualified Java name of the DSML's metamodel package.

5 Case Study: A Debugger for the Petri Net DSML

In this section, we show the manual work necessary for developing a Petri net debugger with *EProvide*. We develop the debugger in five steps, starting with a simple editor and ending with the full debugger. In some steps, manual modifications of Java code generated with EMF or GMF are necessary. These manual modifications could be avoided if EMF and GMF had some missing features that we will identify in the following. Table 1 summarises the work necessary for each step and also shows which work could be saved with which additional features.

Step 1: Editor. The first step is to create a new DSML plugin that contains an EMF compliant metamodel for the Petri net DSML (cf. Fig. 1(a) from Ex. 1) and a GMF-definition of the graphical editor. This step is independent of *EProvide*. The editor definition defines which graphical elements should be used (graphical definition), for which elements of the metamodel creation tools exist (tooling definition), and how metamodel elements are mapped to tooling elements and graphical elements. Figure 4 shows a screenshot of the final debugger; the editor actually does not yet highlight activated and selected transitions.

Step 2: Visual Interpreter without Separate Runtime Attributes. As second step, we create an interpreter that uses the static metamodel to save the runtime states as explained in Ex. 3. For this, we define a new extension for *EProvide*'s extension point in the DSML plugin and we implement the transition transformation as shown in Fig. 2 from Ex. 3. This transition transformation uses the OCL-function `any`, which should behave non-deterministically.

Missing Feature 1: Non-deterministic Choice in QVT Implementation. Unfortunately, the QVT implementation we use lacks non-determinism. Therefore, we have to extend the class `Net` with a method `choose()` implementing a non-deterministic choice. The signature of this method is specified in the metamodel while its implementation is given in Java. We adapt the transition transformation to use `Net.choose()` instead of the OCL-function `any`.

Step 3: Visual Interpreter with Separate Runtime Attributes. In this step, we extend the interpreter so that interpretation does not destroy the initial marking of the Petri net. As described in Ex. 4, this requires to store the configurations in separate metamodel elements and to provide an initialisation transformation. We suggested to define a separate runtime metamodel (Fig. 1(b)) that augments the static metamodel. This would result in the architecture shown in Fig. 8 and would have the advantage that the static metamodel is not polluted with runtime elements. In this architecture, the visual interpreter would be an extension of the model editor and would show the runtime marking instead

Table 1. Manual work required for the development of the Petri net debugger. The work that could be saved with additional EMF/GMF features is marked with the number of the corresponding feature in parentheses.

<i>1. Editor</i>		
Metamodel	3 classes (<code>Net</code> , <code>Place</code> , <code>Transition</code>), 5 associations, 3 attributes	
Editor definition	35 + 13 + 35 XMI nodes (graphical, tooling, mapping)	
<i>2. Visual interpreter without separate runtime attributes, destroys initial state</i>		
Metamodel	1 operation (<code>Net::choose</code>) for non-deterministic choice	(1)
DSML plugin	6 lines Java code for implementing <code>Net.choose()</code>	(1)
	8 lines in the <code>plugin.xml</code> for plugging into <i>EProvide</i>	
Semantics	31 lines QVT Relations code	
<i>3. Visual interpreter with separate runtime attributes</i>		
Metamodel	3 attributes (<code>Net::running</code> , <code>Place::initToken</code> , <code>Place::runtimeToken</code>)	
DSML plugin	23 lines Java code for <code>Net.running-switch</code>	(2)
Semantics	17 lines QVT Relations code for initialising runtime model	
Graphical editor	12 lines Java code for showing a running label	(2)
<i>4. Debugger without breakpoints</i>		
Metamodel	2 attributes (<code>Transition::activated</code> , <code>Transition::selected</code>)	
Semantics	1 line QVT Relations code for using the new attributes	
Graphical editor	27 lines Java code for visualising transition states	(3)
	8 lines Java code for change notif. for <code>Transition.activated</code>	(4)
<i>5. Debugger with breakpoints</i>		
Metamodel	3 classes (<code>Breakpoint</code> , <code>PlaceBP</code> , <code>TransitionBP</code>),	
Semantics	17 lines QVT Relations code to take breakpoints into account	
Editor definition	— (breakpoints can only be defined in generic tree editor)	

of the initial marking of a Petri net. In GMF, the layout information (element positions, size, etc.) of a diagram is stored in a separate graphical model. In the shown architecture, the graphical runtime model would augment the graphical static model so that the layout information from the static model would be reused for visual interpretation.

Missing Feature 2: Editor extension in GMF. GMF does not support this architecture. An editor definition cannot extend another one and a graphical model cannot reference another one. Therefore, we cannot use a separate runtime model with a separate visual interpreter but have to extend the static model to an *integrated model* with runtime elements. The corresponding *integrated metamodel* has three additional attributes and one with changed semantics. `Place.token` now delegates to one of the two new attributes `Place.initToken` and `Place.runtimeToken`. Which it delegates to is controlled by the third new attribute, `Net.running`. In order to show the user whether the static model

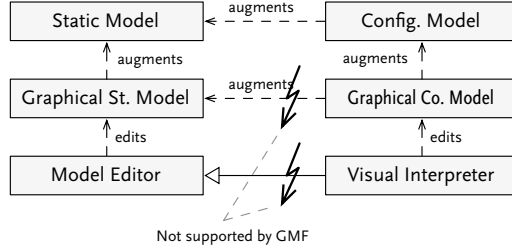


Fig. 8. The editor/interpreter architecture as it should be

or the runtime model is visualised, we manually add a “running” label to the generated GMF editor code that is shown only for the runtime model (cf. Fig. 4).

Step 4: Debugger without Breakpoints. In this step, we add features that allow the user to control the execution of a Petri net model. Stepwise execution of Petri net models is already provided by the *EProvide* infrastructure (cf. Sect. 4.2). In addition, we want to visualise activated transitions with a thick, black border (transition “top” in Fig. 4) and transitions selected by the user to fire with a thick, blue border (transition “middle” in Fig. 4). To implement this, we add two attributes (cf. Ex. 6) to the integrated metamodel: `Transition.activated` is a derived attribute and implemented in OCL, and `Transition.selected` is an ordinary attribute that can be set by the user. Furthermore, we adapt the transition transformation to use the new attributes.

Missing Feature 3: Visual attributes in GMF derived from model attributes. GMF’s editor definition does not support to derive the visualisation of a transition’s border from the transition’s model attributes. Therefore, we manually add the visualisation in the generated GMF editor code.

Missing Feature 4: Change notifications for derived attributes in EMF. EMF allows to define derived attributes in OCL. The Java code generated for the metamodel then contains code that interprets the OCL expression. If an attribute that the derived attribute depends on changes, the getter for the derived attribute returns an updated value when it is called the next time. But the generated code does not automatically send a change notification for the derived attribute. Therefore, we manually modify the generated code to send change notifications for `Transition.activated`.

Step 5: Debugger with Breakpoints. In the last step, we add support for breakpoints. For this, we add the three classes `Breakpoint`, `PlaceBP`, and `TransitionBP` to the integrated metamodel. Furthermore, we adapt the transition transformation to use the new classes (cf. Ex. 6). In this case study, we do not support graphical specification of breakpoints; they have to be specified in the generic EMF tree editor. (If we wanted to support graphical breakpoint specification, we had to extend the editor definition.)

6 Related Work

Ptolemy [9] allows animated interpretation of hierarchically composed domain-specific models with different execution semantics. Adding a new DSML to Ptolemy requires a lot of work as both its syntax and its semantics have to be coded manually in Java. GME [10] provides visualisation of the interpretation and support for creating a DSML editor without manual coding. But as with Ptolemy the interpreter semantics has to be implemented manually in Java or C++.

Several approaches address a metamodel-based formalisation of language semantics. Sunyé et al. recommend UML action semantics for executable UML models [11]. Furthermore, they suggest activities with action semantics for language modelling. Scheidgen and Fischer follow this suggestion and provide description means for the operational semantics of MOF compliant metamodels [12]. Muller et al. integrate OCL into an imperative action language [13] to provide semantics description means for the Kermet framework. In a similar way, OCL is extended with actions to provide semantics description means in the Mosaic framework [14]. The AMMA framework integrates Abstract State Machines for the specification of execution semantics [15]. Furthermore, the model transformation language ATL [16] can be applied to specify operational semantics. These approaches lack visualisation of the interpretation but those based on EMF can be integrated into *EProvide* easily.

Graph transformations are a well-known technology to describe the operational semantics of visual languages. Engels et al. describe the operational semantics of UML behaviour diagrams in terms of collaboration diagrams which represent graph transformations [17]. Similarly, Ermel et al. translate UML behaviour diagrams into graph transformations to simulate UML models [18]. Since they provide domain-specific animation views, the runtime state can not be changed by the user which inhibits debugging.

The Moses tool suite [19] provides a generic architecture to animate and debug visual models, which are represented as attributed graphs. The runtime state is visualised by so called animation decorators added to the attributed graph. The difference to our approach is that in Moses the execution semantics of models is given as an abstract state machine description and the runtime state of a model is encoded in ASM functions. In contrast, we store the runtime state itself as a model, which allows us to reuse the same editor creation technology for animation as for editing.

In AToM3, de Lara and Vangheluwe use graph grammars to define the operational semantics of a visual modelling language [20]. In contrast, we rely on OMG standard means to define metamodel-based operational semantics. Thus, operational semantics can be integrated in an MDA process more naturally.

7 Conclusion

Contribution. We showed how operational semantics of a DSML can be specified at the PIM level with standard modelling techniques. Furthermore, we

combined this approach with existing metamodel-based editor creation technology. This enables rapid prototyping of visual interpreters and debuggers. We illustrated our approach with a DSML for Petri nets. Thereby, we identified desirable features missing in base technologies we used. With *EProvide*, we offer an implementation of our approach as an Eclipse plugin.

Our approach minimises the effort for MDA practitioners to define prototypical language semantics. They can rely on standard means and tools they are used to. This facilitates short iteration circles for language engineering, early integration of domain experts, higher quality of DSMLs and the system under development, and thus minimises development costs.

Future Work. In the Petri net case study, we did not distinguish between static model and configuration model but used an integrated model. Although we did this because of the missing editor extension feature in GMF, it allows for another nice feature: a user can modify static model elements at runtime. As we argued, we would prefer not to pollute the static model with configuration elements. But without an integrated model, allowing the user to modify static model elements at runtime would require additional work. For example, if a user creates a new element in the static model, the corresponding element in the configuration model must be created. This adaptation needs additional specification. Here, the question is whether the adaptation must be programmed manually or if it is possible to describe it declaratively in a model.

Typically, a DSML metamodel evolves over time. Prototypical tool support as mentioned in this paper can help to reveal necessary changes. Automated adaptation of metamodels [21] helps to control metamodel evolution. Automated co-adaptation helps to keep related artefacts like instances or semantic descriptions in sync with the evolving metamodel. For the approach presented in this paper, automated co-adaptation of declarative editor models is needed.

Acknowledgement. We are thankful to Hajo Eichler, Omar Ekine, and Jörg Kiegeland from ikv for help with medini QVT. Also thanks to the anonymous reviewers for valuable comments. This work is supported by grants from the DFG (German Research Foundation, Graduiertenkolleg METRIK).

References

1. Miller, J., Mukerji, J.: MDA Guide Version 1.0.1. Technical report, Object Management Group (OMG) (2003)
2. The Eclipse Foundation: Eclipse Graphical Modeling Framework (GMF) (2007), <http://www.eclipse.org/gmf/>
3. Object Management Group: Meta Object Facility 2.0 Core Specification (2006)
4. Object Management Group: Meta Object Facility 2.0 Query/View/Transformation Specification (2007)
5. Plotkin, G.D.: A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus (1981)

6. da Silva, F.Q.B.: Correctness Proofs of Compilers and Debuggers: an Approach Based on Structural Operational Semantics. PhD thesis, University of Edinburgh (1992)
7. Barendregt, H.P.: The Lambda Calculus its Syntax and Semantics, 2nd edn. North Holland, Amsterdam (1987)
8. Budinsky, F., Merks, E., Steinberg, D.: Eclipse Modeling Framework, 2nd edn. Addison-Wesley, Reading (2006)
9. Brooks, C., Lee, E.A., Liu, X., Neuendorffer, S., Zhao, Y., Zheng, H.: Ptolemy II: Heterogeneous Concurrent Modeling and Design in Java. UC Berkeley (2007)
10. Lédeczi, Á., Bakay, Á., Maróti, M., Völgyesi, P., Nordstrom, G., Sprinkle, J., Karsai, G.: Composing Domain-Specific Design Environments. *Computer* 34(11), 44–51 (2001)
11. Sunyé, G., Pennaneach, F., Ho, W.M., Guennec, A.L., Jéquel, J.M.: Using uml action semantics for executable modeling and beyond. In: Dittrich, K.R., Gerpert, A., Norrie, M.C. (eds.) CAiSE 2001. LNCS, vol. 2068, pp. 433–447. Springer, Heidelberg (2001)
12. Scheidgen, M., Fischer, J.: Human comprehensible and machine processable specifications of operational semantics. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA. LNCS, vol. 4530, pp. 157–171. Springer, Heidelberg (2007)
13. Muller, P., Fleurey, F., Jézéquel, J.: 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)
14. Clark, T., Evans, A., Sammut, P., Willans, J.: Applied metamodeling: A foundation for language driven development (2004), <http://www.xactium.com>
15. Di Ruscio, D., Jouault, F., Kurtev, I., Bezivin, J., Pierantonio, A.: Extending amma for supporting dynamic semantics specifications of dsls. Technical Report HAL - CCSD - CNRS, Laboratoire D’Informatique de Nantes-Atlantique (2006)
16. Jouault, F., Kurtev, I.: Transforming models with atl. In: Bruehl, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
17. Engels, G., Hausmann, J.H., Heckel, R., Sauer, S.: Dynamic meta-modeling: A graphical approach to the operational semantics of behavioral diagrams in uml. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000. LNCS, vol. 1939, pp. 323–337. Springer, Heidelberg (2000)
18. Ermel, C., Hölscher, K., Kuske, S., Ziemann, P.: Animated simulation of integrated uml behavioral models based on graph transformation. In: VL/HCC 2005, pp. 125–133. IEEE Computer Society, Los Alamitos (2005)
19. Robert Esser, J.J.: Moses: A tool suite for visual modeling of discrete-event systems. In: HCC 2001, pp. 272–279. IEEE Computer Society, Los Alamitos (2001)
20. de Lara, J., Vangheluwe, H.: Defining visual notations and their manipulation through meta-modelling and graph transformation. *Journal of Visual Languages & Computing* 15(3-4), 309–330 (2004)
21. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609. Springer, Heidelberg (2007)

Data Flow Analysis of UML Action Semantics for Executable Models

Tabinda Waheed¹, Muhammad Zohaib Z. Iqbal², and Zafar I. Malik³

¹Department of Computer Science, Military College of Signals, National University of Science & Technology, Rawalpindi, Pakistan
tabindawaheed@yahoo.com

²Department of Computer Science, International Islamic University, Islamabad, Pakistan
zouhaib@hotmail.com

³Academy of Education and Planning, Ministry of Education, Islamabad, Pakistan
zafarimalik@hotmail.com

Abstract. Executable modeling allows the models to be executed and treated as prototype to determine the behavior of a system. These models use precise action languages to specify the algorithms and computational details required for execution. These action languages are developed on the basis of UML action semantics metamodel that provides the abstract syntax. The use of a concrete action language makes a traditional model work like an executable one. The actions specified by the action language might involve variables and their data values that are useful to be analyzed in terms of data flow. In this paper, we provide data flow analysis (DFA) of the standard UML action semantics that can be used with executable models. The analysis provides a generic data flow at the abstract syntax level and allows a mapping to any of the action languages providing the concrete syntax. Our approach does not focus on a particular action language; therefore it can easily be applied to any concrete syntax. We apply the proposed approach to a case study and identify the data flow from executable UML state machine.

Keywords: Executable modeling, Executable UML, Data Flow Analysis, Action Semantics.

1 Introduction

Executable modeling is the ability of models to be directly executable through the use of execution semantics [2]. The basic idea behind this concept is to model the systems using UML diagrams at a higher level of abstraction, which can be compiled, translated, and executed in order to validate system's correctness [3, 5]. Formal test cases can be executed against the models to verify their behavior and to check their conformance to the specifications [1, 4]. Earlier versions of UML were not executable due to semantic incompleteness and ambiguities. There are many variations in making UML executable. One of the directions in executable modeling is the concept of Executable UML [1, 4] that is one of the subsets of UML. This subset comprises of UML class diagram, UML state machines and the action language corresponding to

action semantics [1]. In OMG's UML Specifications [6], action semantics have been introduced by giving a set of fundamental actions and their precise semantics. Action semantics provides a standardized and platform independent way to specify the behavior of objects [6, 7]. The metamodel of action semantics provides the abstract syntax and does not have a uniform normative notation. Various vendors have defined the concrete syntax for the action languages based on UML's action specifications. UML 2.1 provides semantics for the basic actions which are concerned with sending signals, invoking operations, reading and writing attributes, etc. These actions also define behavioral constructs that can be implemented by the concrete action languages [6].

In the domain of Executable UML [1], actions are written inside the states in the form of Entry, Do, and Exit actions in a concrete action language. The executable UML assumes the state machines to be flattened [1], hence, the actions are only within the states [33]. These actions specify the activities to be carried out upon entry to a state, activities to be performed when in a particular state, and activities to be carried out while exiting the state, respectively. UML state machine represents the dynamic behavior and life cycle of an object in terms of various states that an object can assume. An object can change its state on an event trigger. This change in state might affect the values of the variables and attributes used within the state, therefore, affecting the overall behavior of the system. The state actions can also be involved in data flow within a state or among multiple states. In order to ensure the correct behavior represented by state machines, these actions should be considered in terms of data values and be analyzed in terms of data flow.

The data flow analysis (DFA) is used to measure program complexity and forms a basis of data flow based testing by employing the definitions and uses of the variables in a program [25, 28]. This information can help to identify the paths where data flow is evident and where data dependencies can affect the overall program. Traditional data flow analysis is employed on code by carrying out either inter-procedural or inter-class DFA [8]. This kind of DFA analyzes the data flow among code variables written in programming languages. In case of action languages used with executable models, the code-based DFA can not be applied due to abstract nature of action languages. Moreover, action languages fall short of addressing platform-specific and low-level details unlike programming languages [9]. In executable UML, the actions are written inside the states of a state-machine and can not be tested as an independent module like a procedure or a class. These actions can also be involved in data flow within a state or among multiple states. Hence, the existing model-based and code-based DFA fails to be applied to executable state-machines because none of these caters either inter-state data flow or testing of the actions.

This paper addresses the issue of analyzing actions for data flow and provides a generic solution. We analyze the action semantics of UML 2.1 in terms of data flow and reveal how data flow and dependencies participate in the actions. The action semantics are independent of a concrete syntax and UML does not provide any standard action language, rather it gives the semantics and abstract syntax, which can be used by the vendors to develop their own action languages. This has resulted in a variety of action languages, each having its own concrete syntax that is not similar to rest of the languages. Due to this variation of syntax, the data flow analysis for one particular action language can not be applied to another action language. Therefore, in

our approach, we eliminate the need of the specific syntax of an action language and base our data flow analysis on the Action Semantics specified by UML. We have not focused on a specific action language; rather we have focused on the actions at an abstract syntax level defined in UML Action Semantics. This makes our DFA generic and flexible to be used with any language compliant to the UML action semantics metamodel.

The rest of the paper is organized as follows: Section 2 discusses related work. Section 3 presents the proposed approach. Section 4 presents a case study to demonstrate the proposed approach. Section 5 concludes the paper and provides possible directions for future work.

2 Related Work

Data flow analysis (DFA) is a process that analyzes the programs on the basis of data and computes associations and relationships between data objects [25, 28]. Data flow analysis utilizes the definitions and uses of the variables in a program. The values of the variables might affect the control flow of the system, hence affecting the execution and behavior of a system. In other words, we can say that a system's functionality can be easily highlighted in terms of data; therefore, data flow information must be considered when a system is intended to be analyzed or tested.

Data flow analysis can be applied both on models as well as code [13, 14, 24]. In model-based testing, covering only the control information does not guarantee that the data is correctly flowing through the model. This requires the incorporation of data flow information to the control flow in order to thoroughly test the given system. There are a few techniques in literature that combine the control and data flow information for testing the models [11, 14, 30, 31]. In this context, Hutchins et al [14] examine the effectiveness of *all-edges* and *all DU-path* (Definition-Use path) coverage criteria and compares data flow and control flow coverage criteria. The authors have proposed a criterion known as *all-DU* and have compared it with the existing coverage metrics. [4].

Combining both control and data flow is also of importance in Embedded Systems. In this regard, Varea and Al-Hashimi [30] present a design representation called Data Flow Net (DFN) to model the interrelationship between control and data flow in Embedded Systems. This model is based on Petri net structure and addresses the control as well as data flow. The methodology takes a DFN model and a set of properties in temporal logic as input and the model checker translates it into source code. Farwer and Varea [31] have examined the effect of object-based approaches on data flow and control flow analysis of models. The focus is on the separation of control flow and data flow by using Object Net and System Net, where the former represents data and latter depicts control. The authors have used the approach of Petri-nets and provided a formal translation procedure for transforming a Dual Flow net into Object Petri net [31].

Data Flow Analysis has numerous applications such as code optimization [24], program slicing [25], data flow based testing, reaching definitions, constant propagation problem, dead code elimination, path profiling, forward and backward analysis [23]. DFA is also of importance in the compilation process of programs,

especially *Parallel Compilers* [26] and program slicing that is employed in interpreters, compilers, and other program manipulation tools [25, 27]. These tools normally use demand-driven techniques and slicing is incorporated for data flow analysis. Another application of DFA is to measure software complexity where control flow often seems insufficient. The data flow-based metric forms the basis for data flow based testing because this metric uses the definitions and uses of the variables in a program [29].

The most commonly used application of data flow analysis is data flow-based testing. Many researchers [12, 32] have discussed various forms of code-based testing including path testing and a variety of test coverage metrics and metric-based testing methods. Harrold & Rothermel [13] have presented an approach to test the methods that are accessible outside the class and can be called in an arbitrary order by using the data flow interactions between them. For this purpose, a class control flow graph has been developed which represents all the methods of a class connected via data flow. This graph helps in testing the class at three levels that correspond to 3 levels of Def/Use (DU) pairs. Tsai et al [10] address the issue of applying data flow testing to test the classes in object-oriented systems and to avoid infeasible test cases due to the existence of data flow anomalies. The authors discuss data flow testing in two phases: detection and removal of data flow anomalies, and intra-class test case generation from sequences of messages. Another major contribution towards data flow based testing is by Rapps and Weyuker [11]. They have addressed the issue that *all-path* coverage criterion may not be able to reveal all possible errors in a software system because it employs only control flow for testing purpose. The authors develop a program graph which is then converted into a def-use graph by identifying definitions and uses from each node.

The above discussion leads to the conclusion that DFA has numerous applications and is the most effective way to determine data flow. This information of data flow and data dependencies cannot be identified merely by control flow. Therefore, data flow information is required to provide the complete coverage of code as well as models and to ensure the correctness of the systems.

3 The Proposed Approach

Action language compliant to UML action semantics is the core of Executable UML, since it allows models to be executed. Various action languages are reported in literature, each having its own concrete syntax but having the same semantics. Based on the actions defined in action semantics metamodel, any language can be developed with a concrete syntax. Any language-specific technique developed for a particular syntax might not cover all the aspects available in some other language, hence can not provide reusability and generality. This can be avoided by focusing on the semantics at the abstract syntax level instead of concrete syntax so as to provide a generic solution. Keeping this in mind, we provide an approach that analyzes the UML action semantics in terms of data flow and can be used with Executable UML. We also provide a mapping from abstract syntax to concrete syntax. Our proposed approach is generic and provides the flexibility to be used with any action language by providing its Context-Free Grammar and mapping rules.

In this paper, we have used Action Specification Language (ASL) [15] as an example language to explain the idea. ASL is an abstract action language to define processing for executable UML models in an unambiguous and precise manner. The language defines its own syntax that includes all the basic syntactic constructs as required by a language. The syntactic information has been provided in terms of data types, sequential logic, classes and object manipulation, associations and generalizations, signals, arithmetic and logical operations, etc [15]. Some of these are typical language constructs that are purely syntax based, whereas, others are directly based on the semantics. Various ASL statements are based on the action semantics defined by UML. It is also used in a tool provided by Kennedy Carter for executable UML modeling known as iUML Model Simulator [16]. Our proposed approach is generic and is not restricted to a single action language. Several action languages exist, such as, BridgePoint's OAL [17], SMALL, TALL, Kennedy-Carter's ASL [15], JAL [18], Pathfinder Solutions' PAL [19], JUMBALA [20] and Kabira's Action Semantics [21] that can be used instead of ASL. We have used ASL, which is the most widely used language in the context of Executable UML.

The process flow of our proposed technique is presented in figure 1. The ellipses represent the activities while the boxes represent the input and output from each activity. The technique takes a state machine model as input. The state machine contains actions written in ASL as an example action language. These actions are written inside the states, while the transition actions are not considered since the state machine is flattened. The flattened state machine also assumes the guard conditions written in OCL to be transformed accordingly [34]. In our approach, the state machine can be taken in the form of XMI that is a widely used representation especially in model-based automated tools. This input state machine is transformed into a feasible path matrix to store the control flow information. In addition to control flow, our approach incorporates the data flow information by carrying out a detailed data flow analysis of actions which are written inside the states in ASL. These actions are first parsed based on the Context-Free Grammar (CFG) of the ASL and mapping rules have been provided for concrete syntax to abstract syntax. From this DFA, we find the def-use associations among the variables used within the states of input model. The steps of the technique are described in the following sub-sections.

3.1 Construct Feasible Path Matrix

UML state machine represents the life cycle of an object in the form of various states. The transitions between the states represent the flow of control from one state to another. This control flow information can be stored in a flow graph or matrix that can be used later on. In our approach, we construct a feasible path matrix from the input state machine. This matrix is constructed to identify the possible transitions among states, i.e., feasible paths. These feasible paths help to reduce the effort of searching the entire state-machine to examine data dependencies. The matrix consists of rows and columns equal to the number of states. Each entry of the matrix is either 0 or 1. An entry with 1 depicts that the two states are reachable and there is some path between the states, either direct or indirect. The entry with 0 represents an infeasible path. This feasible path matrix resembles the adjacency matrix, which shows only the

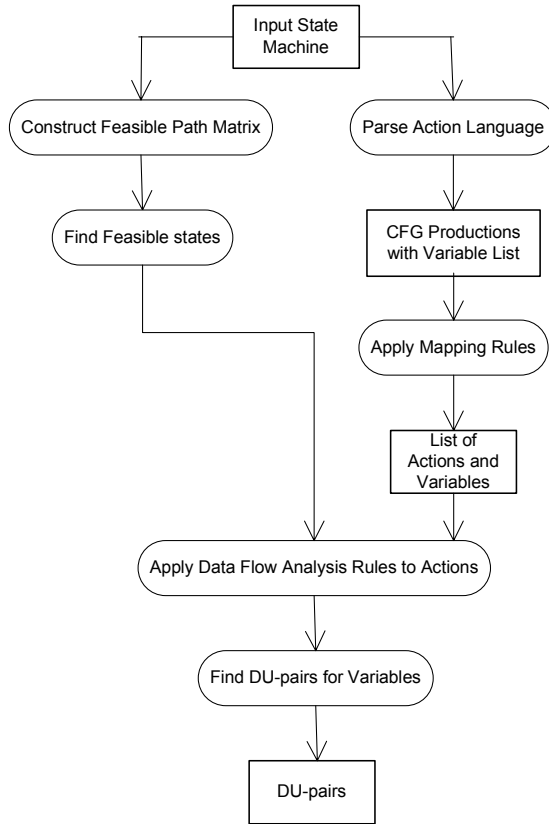


Fig. 1. Process Flow of Proposed Approach

direct paths between the nodes of a graph [22]. In addition of showing the direct paths for reachable states as in adjacency matrix, the feasible path matrix also shows the indirect paths between the states.

3.2 Find Feasible States

The feasible path matrix, constructed in previous step, depicts all those states that are reachable from a given state of the input state machine. The knowledge of reachable states helps to keep track of the control flow among various states. The feasible path matrix also plays a major role in identifying the candidate states for carrying out data flow analysis. This is because traversing all the states of the state machine to identify data flow is infeasible and exhaustive. Moreover, certain data dependencies may exist among states that cannot be identified by traditional data flow based testing techniques. The order of execution of the states is also important and should be considered while handling the control flow. To overcome the above issues, a feasible path matrix can be constructed to identify only the feasible paths in a state-machine.

From this matrix, we can find those states that are reachable from a given state, either directly or indirectly and, hence, can generate feasible paths.

3.3 Parse Action Language

In addition to the construction of feasible path matrix, the input state machine is sent to the action language parser that parses the actions written in the states of the executable state-machine. The parser can be easily developed that uses the context free grammar (CFG) of the action language. It parses the given action language statements into tokens based on the CFG productions. During the parsing process, the parser also generates the corresponding CFG productions for each line of ASL code in the form of a parse tree. We also store the variables and structural features used in each ASL statement along with the CFG productions. This information is required to compute data flow analysis for the variables and structural features. CFG represents the syntactic structure of the action language in the form of productions or rules. As an example, an excerpt of the CFG for ASL (Action Specification Language) is shown in figure 2.

```

<ASL_SEGMENT> -> <STATEMENTS> | <ASL_FUNCTION>
<STATEMENTS> -> <STATEMENT> <STATEMENTS> | ε
<STATEMENT> -> <SIMPLE_STMT> | <SEQ_LOG_STMT>
<SIMPLE_STMT> -> <CREATE_STMT> | <DELETE_STMT> |
                  <ASSIGNMENT_STMT> | <GENERATE_STMT>
<CREATE_STMT> -> <CREATE_OBJECT_STMT> | <CREATE_LINK_STMT>

```

Fig. 2. An excerpt of CFG for ASL

3.4 Apply Mapping Rules

After parsing the action language, the parser generates the CFG production rules and variable list for each language statement. The CFG productions are generated in the form of a parse tree. Because, our proposed approach is not language-specific; we need to have some generic information for applying the testing strategy. For this purpose, we have defined some rules that provide a mapping from action language to corresponding actions defined in UML 2.1 Action Semantics. We have provided a mechanism to map the abstract syntax of actions to the concrete syntax of action languages. This mechanism provides the flexibility to apply our approach, to any concrete action language by simply providing mapping rules for that particular language and giving its CFG to the generic parser.

The parse tree generated in the previous step is traversed in depth-first manner to find the CFG production with a corresponding mapping rule. While traversing the tree, each node and its child nodes are checked to see if a corresponding mapping rule representing a CFG production exists against this node. If found, the action on the right hand side of the mapping rule is returned as output.

The mapping rules are used to obtain the actions against each ASL statement of the input model. This is required because we have carried out the data flow analysis on actions, rather than a specific action language. We can write the mapping rules in an

XML file based on the XML schema to define the mapping rules for any language. The XML schema and an excerpt of the mapping rules is shown in figure 3 and 4 respectively. The rules basically map each CFG production to a corresponding action. Each rule consists of a CFG tag and an action. The CFG tag is further composed of a left and right tag to represent the CFG production. The right tag can be empty if the production consists of a single non-terminal symbol. In case of multiple actions against a single CFG production, we use the equality operator to separate the two actions. This equality operator helps to distinguish between the actions at the left of an expression from the one at the right. The output of this activity is a list of actions and variables against each CFG production obtained in previous step.

```

<xs:schema elementFormDefault="qualified"
  targetNamespace="http://www.w3schools.com">
  <xs:import namespace="http://www.w3.org/2001/XMLSchema-instance"
    schemaLocation="xsi.xsd"/>
  <xs:element name="mappingRules">
    <xs:complexType>
      <xs:sequence>
        <xs:element maxOccurs="unbounded" ref="rule"/>
      </xs:sequence>
      <xs:attribute ref="xsi:schemaLocation" use="required"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="rule">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="CFG"/>
        <xs:element ref="action"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="CFG">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="left"/>
        <xs:element ref="right"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="left" type="xs:string"/>
  <xs:element name="right" type="xs:string"/>
  <xs:element name="action" type="xs:string"/>
</xs:schema>

```

Fig. 3. XML Schema for Mapping Rules

3.5 Apply Data Flow Analysis Rules to Actions

After obtaining the list of actions and variables, we apply data flow analysis rules to these actions. These rules have been defined for the actions provided by UML 2.1 superstructure. UML 2.1 provides semantics for the basic actions that are concerned

```

-<mappingRules xsi:schemaLocation="ruleSchema.xsd">
  -<rule>
    -<CFG>
      <left> FUNCTION_INVOCATION1 </left>
      <right> # </right>
    </CFG>
    <action>CallAction</action>
  </rule>
  -<rule>
    -<CFG>
      <left> READ_STMT </left>
      <right> create CLASSNAME CREATE_OBJECT_STMT1 </right>
    </CFG>
    <action>WriteVariableAction=CreateObjectAction</action>
  </rule>
  .....
</mappingRules>

```

Fig. 4. An excerpt of mapping rules

with sending signals, invoking operations, reading and writing attributes and values, etc. The major categories of the actions defined by UML 2.1 are read/write actions and invocations actions [6]. We have carried out the data flow analysis of UML actions and have categorized them as defining or using some values. The data flow analysis rules have been defined in terms of definition and use of each variable, attribute, or object in the input model. This corresponds to the conventional def-use pairing of variables employed in data flow based testing techniques. The DFA of the actions is summarized in table 1; due to space limitation, we are not providing the details of each action. An entry of the table with 'N/A' means that the action is abstract and its def-use classification depends on its concrete subclasses.

3.6 Find DU-Pairs

After applying DFA rules on the actions, each action has been categorized as being defined or used. From this information, we generate DU-pairs to represent definition-use associations among the variables. We find DU-pairs for each variable and structural feature within a state by looking at its assignment as def or use. This process is repeated for each state in the input model. For inter-state data flow, we consult the feasible path matrix for only the reachable states instead of traversing the whole state-machine. For each feasible state, we look for each variable or structural feature present in one state and find if it is present in some other feasible state or not. If the variable or structural feature also exists in feasible states, then its inter-state DU-pairs are also generated depending on its assignment as def or use.

At the end of this process, we obtain the DU-pairs for all the variables or structural features that are involved in data flow within a state or among multiple states that are reachable from one state. The DU-pairs can later be used for several applications such as data flow testing, program slicing, forward and backward analysis, etc.

Table 1. Def-use classification of UML Actions

Action	Def/Use	Action	Def/Use
AcceptCallAction	Use	ReadLinkAction	Def/Use
AcceptEventAction	Use	ReadLinkObjectEndAction	Use
AddStructuralFeatureValueAction	Def	ReadLinkObjectEndQualifierAction	Use
AddVariableValueAction	Def/Use	ReadSelfAction	Use
BroadcastSignalAction	Use	ReadStructuralFeatureAction	Use
CallAction	Use	ReadVariableAction	Use
CallBehaviorAction	Use	ReclassifyObjectAction	Def
CallOperationAction	Use	ReduceAction	Def
ClearAssociationAction	Def	RemoveStructuralFeatureAction	Def
ClearStructuralFeatureAction	Def	RemoveVariableValueAction	Def
ClearVariableAction	Def	ReplyAction	Def/Use
CreateLinkAction	Def/Use	SendObjectAction	Use
CreateLinkObjectAction	Def/Use	SendSignalAction	Use
CreateObjectAction	Def	StartClassifierBehaviorAction	Use
DestroyLinkAction	Def/Use	StructuralFeatureAction	N/A
DestroyObjectAction	Def	TestIdentityAction	Use
InvocationAction	N/A	UnmarshallAction	Use
LinkAction	N/A	ValueSpecificationAction	Use
OpaqueAction	N/A	VariableAction	N/A
RaiseExceptionAction	Use	WriteLinkAction	N/A
ReadExtentAction	Use	WriteStructuralFeatureAction	N/A
ReadIsClassifiedAction	Use	WriteVariableAction	N/A

4 Case Study

In order to check the applicability of our proposed approach, we have applied it to a case study of Elevator Control System (ECS). ECS models the system of managing an elevator and its behavior. Figure 5 shows a state machine of Elevator object with 13 states and transitions among the states. Each of the states contains actions written in an action language. For our case study, we have used ASL as an example language.

Figure 6 shows the feasible path matrix constructed for the elevator state machine. Each of the states of input model is analyzed to find def-use associations among the variables. As an example, we only discuss here state 1 of the state machine, i.e., Idle | DoorClosed. This state has 5 actions written as ASL statements that are parsed by action language parser based on the CFG. This parsing is represented as parse trees. After parsing, the parse tree is traversed in depth-first and mapping rules are applied. The mapping rules return the action against each CFG production. Then each action is analyzed on the basis of data flow analysis rules discussed in section 3.6, depending on the variable or structural feature used. As a result, each variable or structural feature used in each statement is categorized as def or use as shown in table 2. Using this information, we compute def-use pairs. Some of the resulting DU-pairs for the variables in state 1 are depicted in table 3.

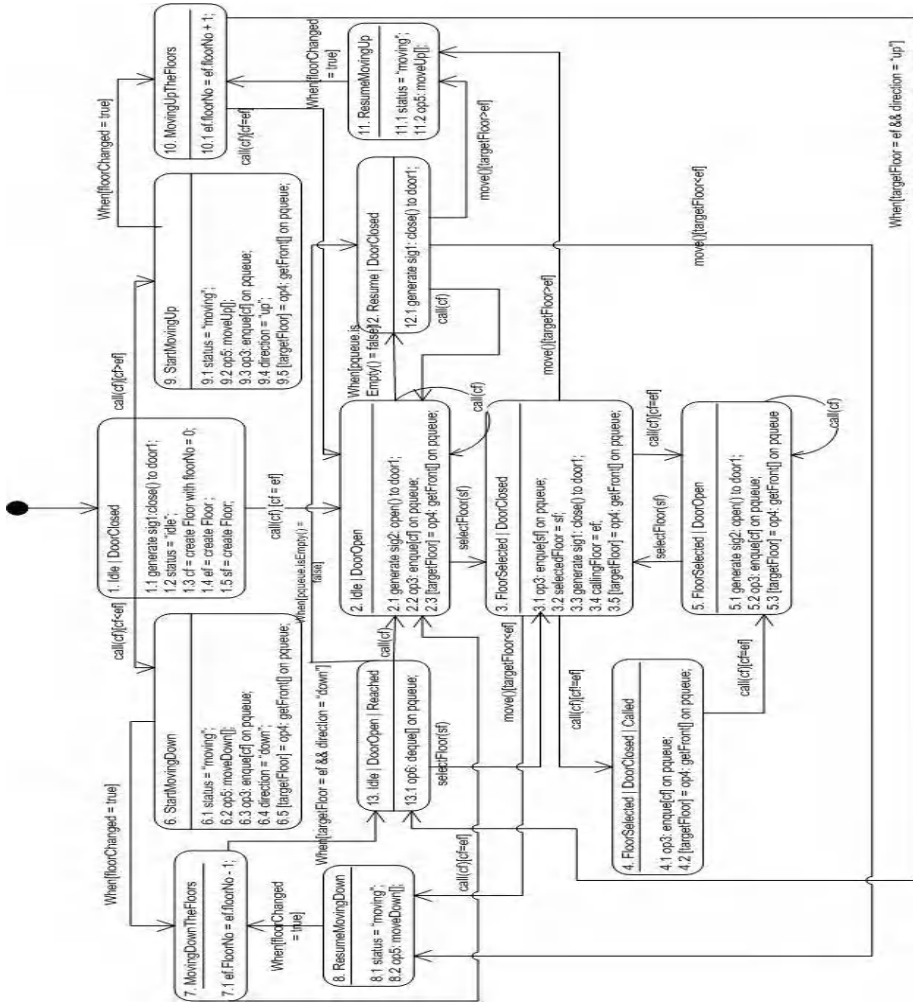


Fig. 5. State machine of Elevator

Table 2. Actions and the variable categorization for the input statements

ASL statement	Action Obtained	Variable List	Categorization
1.1	SendSignalAction	door1	Use
1.2	AddStructuralFeatureValueAction	status	Def
1.3	AddVariableValueAction = CreateObjectAction, AddStructuralFeatureValueAction	cf, floorNo	Def, Def
1.4	AddVariableValueAction = CreateObjectAction	ef	Def
1.5	AddVariableValueAction = CreateObjectAction	sf	Def

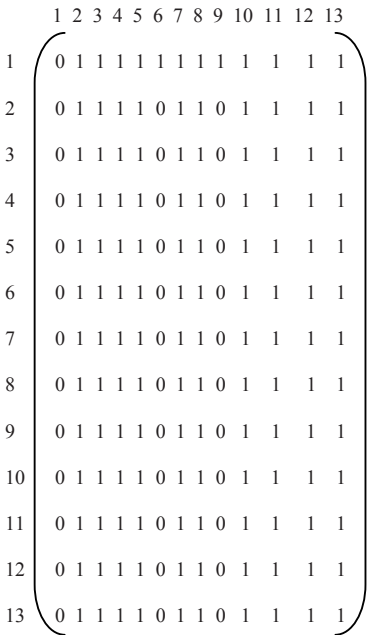


Fig. 6. Feasible Path Matrix for Elevator state machine

Table 3. DU-pairs for the variables/structural features in state machine

Variable / Structural Feature	DU-pair
cf	(1.3, 2.2)
cf	(1.3, 5.2)
cf	(1.3, 6.3)
cf	(1.3, 9.3)
floorNo	(1.3, 7.1)
floorNo	(1.3, 10.1)
floorNo	(7.1, 7.1)
floorNo	(7.1, 10.1)
ef	(1.4, 3.4)
ef	(1.4, 7.1)
ef	(1.4, 10.1)
ef	(7.1, 3.4)
ef	(7.1, 7.1)
ef	(7.1, 10.1)
sf	(1.5, 3.1)
sf	(1.5, 3.2)

5 Conclusion and Future Work

In this paper, we have presented the data flow analysis of actions defined by UML Action Semantics. This data flow analysis forms the basis of data flow-based testing and can also be used in other applications. In our approach, DFA is used to find def-use associations among the actions written in an action language. Our proposed technique is applicable to Executable models, in which the models can be directly compiled and executed prior to implementation. This is a major contribution because the Executable models use actions inside the states of the state-machine to enable a model to be executed and compiled. There has been very little work done in the testing of executable models, hence our technique has a major impact on the domain of Executable modeling. As a proof of concept, we have applied our technique on a real world case study of Elevator Control System (ECS).

We have carried out a detailed data flow analysis of UML actions that can be used in data flow testing, program slicing, code optimization and other applications of DFA. Our future work includes application of the proposed approach on a number of case studies of various sizes to have an empirical evaluation of the applicability of our approach. Mutation analysis can also be applied to ensure the fault detection effectiveness of our technique. We are developing a tool to prove the authenticity and applicability of our approach. Another possible direction can be to extend the technique to other UML artifacts, e.g., Activity Diagram in the context of executable modeling.

Acknowledgments. Our special dedication and acknowledgement to Dr. Jaffar-ur-Rehman, the founder of Center for Software Dependability (CSD), who was a source of inspiration and motivation for us. We would also like to acknowledge the efforts of Dr. Aamer Nadeem of CSD for his support, and all our fellow researchers for their cooperation.

References

1. Mellor, S.J., Balcer, M.J.: Executable UML; A Foundation for Model-Driven Architecture. Addison-Wesley, Reading (2002)
2. Eakman, G.T.: Verification of Platform Independent Models. In: UML Workshop: Model Driven Architecture in the Specification, Implementation and validation of Object-oriented Embedded Systems, San Francisco, California (October 2003)
3. Mellor, S.J., Agile, M.D.A.: MDA Journal (2004) [Accessed: July 16, 2007], http://www.omg.org/mda/mda_files/Agile_MDA.pdf
4. Mellor, S.J.: Introduction to Executable and Translatable UML, Application Development Toolkit, Whitepapers CNET Networks (2005)
5. Ambler, S.W.: Be Realistic About the UML: It's Simply Not Sufficient, AmbySoft Inc. (2005) [Accessed: September 12, 2006], <http://www.agilemodeling.com/essays/realisticUML.htm>
6. Unified Modeling Language: Superstructure. Version 2.1. Object Management Group (OMG) document ptc/06-04-02 (2004), <http://www.omg.org>

7. Sunyé, G., Pennaneac'h, F., Ho, W., Guennec, A.L., Jézéquel, J.M.: Using UML Action Semantics for Executable Modeling and Beyond. In: Proceedings of the 13th International Conference on Advanced Information Systems Engineering. ACM, New York (2001)
8. Harold, M.J., Rothermel, G.: Performing Data Flow Testing on Classes, SIGSOFT 1994-12/94 New Orleans, LA, USA. © ACM, New York (1994)
9. Mellor, S.J., Tockey, S., Arthaud, R., Leblane, P.: Software-Platform-independent, Precise Action Specifications for UML (2006), Available: Project Technology [Accessed December 5, 2006], <http://www.projtech.com>
10. Tsai, B.Y., Stobart, S., Parrington, N.: Employing Data Flow Testing on Object-oriented Classes. IEE Proceedings-Software 148(2) (April 2001)
11. Rapps, S., Weyuker, E.J.: Data Flow Analysis Techniques for Test Data Selection. In: Proceedings of the 6th International Conference on Software Engineering, Tokyo, Japan, pp. 0270–5257. IEEE, Los Alamitos (1982)
12. Gupta, N., Gupta, R.: Data Flow Testing, The Compiler Design Handbook: Optimization and Machine Code Generation, ch.7, 1st edn., pp. 247–267. CRC Press, Boca Raton
13. Harold, M.J., Rothermel, G.: Performing Data Flow Testing on Classes, SIGSOFT 1994, 12/94 New Orleans, LA, USA. © ACM, New York (1994)
14. Hutchins, M., Foster, H., Goradia, T., Ostrand, T.: Experiments on the Effectiveness of Dataflow and Controlflow-based Test Adequacy Criteria. In: Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, pp. 191–200. IEEE Computer Society Press, Los Alamitos (1994)
15. Carter, K.: <http://www.kc.com>
16. Wilkie, I., King, A., Clarke, M., Weaver, C., Raistrick, C., Francis, P.: UML ASL Reference Guide, ASL Language Level 2.5, Manual Revision D (2003)
17. Nucleus UML Suite, http://www.mentor.com/products/embedded_software/nucleus_modeling/index.cfm
18. Trong, T.D., Ghosh, S., France, D.: JAL: Java like Action Language, Specification 1.1 – Beta version. CS Technical Report 06-102. Department of Computer Science, Colorado State University, Fort Collins, USA
19. Pathfinder Solutions. Platform independent Action language, version 2.2 (December 2004), <http://PathfinderMDA.com>
20. Dubrovin, J.: JUMBALA – An Action Language for UML State Machines, HUT-TCS-A101. Research Reports 101. Laboratory for Theoretical Computer Science, Helsinki University of Technology, Espoo, Finland (March 2006)
21. Kabira Technologies, <http://www.kabira.com>
22. Skiena, S.: Adjacency Matrices. In: Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica, pp. 81–85. Addison-Wesley, Reading (1990); Cambridge University Press
23. Sagiv, M., Reps, T., Horwitz, S.: Precise interprocedural dataflow analysis with applications to constant propagation. In: Mosses, P.D., Schwartzbach, M.I., Nielsen, M. (eds.) CAAP 1995, FASE 1995, and TAPSOFT 1995. LNCS, vol. 915. Springer, Heidelberg (1995)
24. Ammons, G., Larus, J.R.: Improving Data-flow Analysis with Path Profiles. In: SIGPLAN 1998, Montreal, Canada. © ACM, New York (1998)
25. Moonen, L.: A Generic Architecture for Data Flow Analysis to Support Reverse Engineering. In: Proceedings of the Second International Workshop on the Theory and Practice of Algebraic Specifications (ASF+SDF 1997), Electronic Workshops in Computing, Amsterdam. Springer, Heidelberg (1997)

26. Kramer, R., Gupta, R., Soffa, M.L.: The Combining DAG: A technique for Parallel Data Flow Analysis. *IEEE Transactions on Parallel and Distributed Systems* 5(8) (August 1994)
27. Atkinson, D.C., Griswold, W.G.: Implementation Techniques for Efficient Data-Flow Analysis of Large Programs. In: *Proceedings of International Conference on Software Maintenance*, Florence, Italy. IEEE, Los Alamitos (2001); 0-7695-1189-9
28. Oviedo, E.I.: Control flow, data flow, and program complexity. In: *Mcgraw-Hill International Series In Software Engineering*. Mcgraw-Hill Inc., USA (1993)
29. Tai, K.C.: A Program Complexity Metric based on Data Flow Information in Control Graphs. In: *Proceedings of the 7th International Conference on Software Engineering*, Orlando, Florida, United States, pp. 239–248. IEEE, Los Alamitos (1984)
30. Varea, M., Al-Hashimi, B.M.: Dual Flow Nets: Modeling the Control/Data-Flow Relation in Embedded Systems. *ACM Transactions on Embedded Computing Systems (TECS)*. ACM Press, USA (2006)
31. Farwer, B., Varea, M.: Object-based Control/Data-flow Analysis, Technical Report DSSE-TR-2005-1, Declarative Systems and Software Engineering Group, University of Southampton, Southampton (2005)
32. Jorgensen, P.C.: *Software Testing: A Craftsman's Approach*, 2nd edn., pp. 137–174. CRC Press Inc., Boca Raton (2002)
33. Kim, Y.G., Hong, H.S., Cho, S.M., Bae, D.H., Cha, S.D.: Test Cases Generation from UML State Diagrams. In: *IEE Proceedings-Software*, vol. 146, pp. 187–192 (August 1999)
34. Briand, L.C., Labiche, Y., Lin, Q.: Improving State chart testing Criteria Using Data Flow Information. In: *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE 2005)*. IEEE, Los Alamitos (2005)

From UML Activities to TAAL - Towards Behaviour-Preserving Model Transformations

Gregor Engels¹, Anneke Kleppe², Arend Rensink², Maria Semenyak¹,
Christian Soltenborn¹, and Heike Wehrheim¹

¹ University of Paderborn, Department of Computer Science,
33098 Paderborn, Germany

{engels, semenyak, christian, wehrheim}@upb.de

² University of Twente, Department of Computer Science,
7500 AE Enschede, The Netherlands*
rensink@cs.utwente.nl

Abstract. Model transformations support a model-driven design by providing an automatic translation of abstract models into more concrete ones, and eventually program code. Crucial to a successful application of model transformations is their *correctness*, in the sense that the meaning (semantics) of the models is preserved. This is especially important if the models not only describe the structure but also the intended *behaviour* of the systems. Reasoning about and showing correctness is, however, often impossible as the source and target models typically lack a precise definition of their semantics.

In this paper, we take a first step towards provably correct behavioural model transformations. In particular, we develop transformations from UML Activities (which are visual models) to programs in TAAL, which is a textual Java-like programming language. Both languages come equipped with formal behavioural semantics, which, moreover, have the same semantic domain. This sets the stage for showing correctness, which in this case comes down to showing that the behaviour of every (well-formed) UML Activity coincides with that of the corresponding TAAL program, in a well-defined sense.

1 Introduction

The concept of model-driven development (MDD) crucially depends on the possibility of generating lower-level models (and finally code) from abstract models. Originally meant as a help for structuring complex programs, models today take on a different, and much more central, role: they not only act as the primary entity for discussions with customers, but also within the development trajectory, for fixing interfaces with other systems or analysing the system with respect to requirements. Thus, it is vital to ensure that the actual system really adheres to the models. The MDD way of ensuring this is by directly generating the code from the models, possibly through intermediate steps where abstract models are refined into more concrete ones. However, this process, called model transformation, is a real solution only by virtue of the *correctness* of the

* The research by this group was financed by the GRASLAND project, funded by the Dutch NWO (project number 612.063.408).

individual transformations, in the sense that they themselves do not change the meaning (usually called the *semantics*) of the models in unintended ways. Showing that transformations are semantics-preserving is the core problem addressed by this paper. In fact, we concentrate on *behavioural* semantics, which is concerned with what the system actually does (in contrast to, for instance, structural semantics, which is concerned with the system architecture).

The problem is aggravated by the fact that model transformations usually go between different meta-models. Quite often, the abstract source model is developed using a *visual* modelling language (e.g. the UML), whereas the target model is *textual* (e.g., a program).

A lot of research has been devoted to finding appropriate languages for describing model transformations in the first place [1,2], a quest that has recently resulted in the QVT language proposed by the OMG group (see [3]). In this context, various notions of “correctness” have already been studied. Correctness can for instance refer to the *syntactical* correctness of the generation algorithms (e.g. of transformation rules in a rule-based setting [4]), it can be *termination* of the generation [5] or the uniqueness of the generated model (*confluence* of rules) [6]. Behaviour preservation, addressed here, is different from all these — in fact it presupposes that the transformations are already correct in the above senses — and is particularly challenging. An area where behaviour preservation has received some interest is *refactoring*, a specific kind of model transformation improving the structure of models [7,8]. Contrary to our interest here, transformations during refactorings do not operate on different meta-models but stay within one language.

Showing behaviour preservation of model transformations between different meta-models first of all requires a *formal* definition of the behavioural semantics of source and target model. Moreover, the semantic domains should be the same, to avoid yet another transformation on semantic domains. Given a formal semantics, a comparison of the behaviour of source and target model is a matter of selecting an appropriate notion of equivalence over the semantic domain.

In this paper, we show that this ideal of showing behavioural correctness of model transformations is indeed attainable. As an example, we define a transformation from UML Activities to TAAL [9] programs. An overview of the approach is depicted in Fig. 1. UML Activities are used to model the orderings of actions within, for instance, business processes (see [10]). They are frequently employed in workflow modelling and constitute a very high level, visual description of workflows. They are defined as a subset of UML, which we will denote UMLA in the sequel. On the other hand, TAAL is a simple (Java-like) object-oriented programming language, featuring class definitions, object instantiation and concurrency. The transformation thus has to bridge the gap between a *visual* model on the one side and *program code* on the other side. We achieve this by defining the model transformation on the (MOF-compliant) abstract syntax meta-models of the two languages (*MT* in Fig. 1). The model transformation is thus a transformation of one graph into another, and consequently we employ graph transformation rules for their definition. This gives us a model transformation that is both formally defined and executable, employing the graph-transformation tool Groove [11] for rule execution.

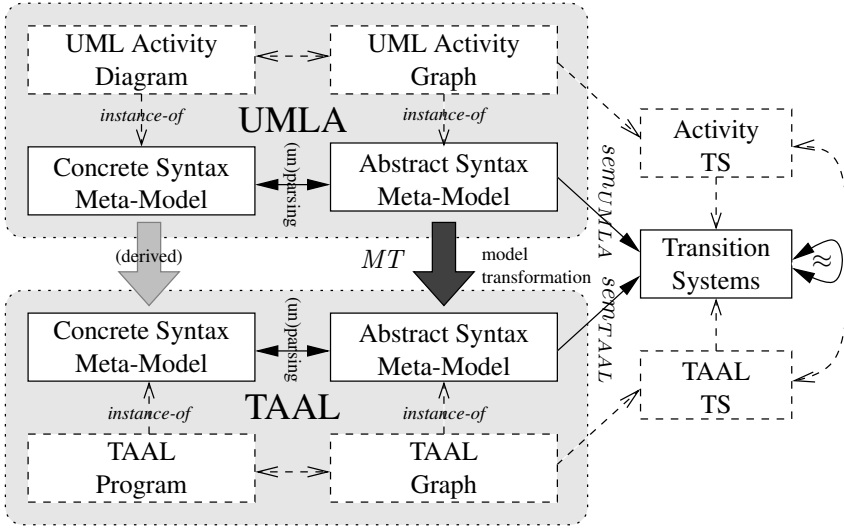


Fig. 1. Overall approach of behaviour-preserving model transformation

This choice of example model transformation has the great advantage that both languages (UMLA and TAAL) are already equipped with a formal semantics, which, moreover, is defined on the same basis, viz. (again) through graph transformation. On the UMLA side, we use a semantics defined with *Dynamic Meta Modelling* [12]; for TAAL, the semantics were developed together with the language [9]. These formal semantics (sem_{UMLA} and sem_{TAAL} in Fig. 1) first add run-time specific structure to the meta-models (e.g. a program counter representation on the TAAL side) and then define the behaviour as the possible changes in instances of that enhanced meta-model. Again, meta-models being graphs leads to a graph-rule based definition of the semantics, and we also use Groove to automatically derive the semantics of both UML Activities and TAAL programs. The underlying common semantic domain are *transition systems* (TS), in which transitions represent applications of graph rules, in particular also those corresponding to executions of *actions* (in the UML Activity) or *operations* (in the TAAL program). Our semantics thus generates a transition system out of a meta-model instance of a UML Activity (TS_{Act}) and TAAL program (TS_{TAAL}). On these transition systems we can compare the execution behaviour of UML Activity and generated TAAL program, and can show that the ordering of actions in the Activity coincides with the ordering of corresponding methods (with the same name) in the TAAL program.

Similar approaches to evaluating the correctness of model transformations have been presented in e.g. [13], where different variants of Statecharts are transformed into each other, and a bisimilarity check is carried out (on particular instances). In a sense, our technique also resembles certification techniques for compilers [14,15], where one particular instance of compilation is afterwards checked for correctness using a generated certificate. Nevertheless, our ultimate aim is a general proof of correctness of

the *transformation*. Once this task is done, we do not have to check the behaviour of transformation results any more. This paper presents the first steps in this direction, giving the model transformation itself, its tool-support, the two semantics and a comparison of the behaviour, viz. semantics, on examples.

In Section 2, we start with a short introduction to UML Activities and TAAL, and we define the graph-based transformations as a set of transformation rules over the meta-models of the two languages. In Section 3 we then argue that the semantics of Activity and generated TAAL program coincide with respect to *trace equivalence*, when comparing the execution traces of the UML model and the object-oriented program. Section 4 describes the tool support for transformation and semantics generation.

2 Transformation

This section presents the model transformation from UML Activities to TAAL. We start by discussing the general idea of the transformation and give a first example.

UML Activities are an expressive tool for expressing the order of execution of so-called *Actions*, i.e., atomic behavioural units. The ordering is specified by a directed graph with different sort of nodes: *Actions* themselves form nodes, the start and end of an execution is marked with a special *InitialNode* and *FinalNode*, and *MergeNodes* and *DecisionNodes* regulate the flow of control. Figure 2 shows an example Activity (plus its corresponding TAAL program). The semantics of the Activity is as follows: The first *Action* to be executed is *A*, indicated by the fact that the *InitialNode* (the filled circle) points to it. The *A* *Action* is followed by a *MergeNode* and a *DecisionNode*; if the guarding condition is true, *Actions B* and *C* will be executed, otherwise *D* is executed, and the Activity ends (indicated by the *FinalNode*).

Due to the *Merge-* and *DecisionNodes*, UML Activities allow for an unstructured flow of control which is hard to translate into a structured programming language without *GOTO* statements. Therefore, we restrict our model transformation to *well-formed* Activities which have a structured control flow.

Well-formedness is inductively defined (similar approaches to well-formedness can be found in [16]). For this, we introduced the concept of *building blocks*. Every building block has exactly one incoming and one outgoing edge connecting it to the rest of the Activity.

- An *Action* itself constitutes a building block (see Fig. 3a).
- A sequence of two building blocks, connected by an *ActivityEdge*, is a building block (see Fig. 3b).
- A *DecisionNode*, followed by two building blocks and a closing *MergeNode*, is a building block (see Fig. 3c). Note that one of the outgoing *ActivityEdges* of the *DecisionNode* must be equipped with a guard (i.e., a *ValueSpecification*).
- A *MergeNode* followed by a *DecisionNode* and a building block which is itself connected to the *MergeNode* is a building block (see Fig. 3d). Here, the *DecisionNode* has an additional outgoing *ActivityEdge* which is taken if the guarding condition is false.

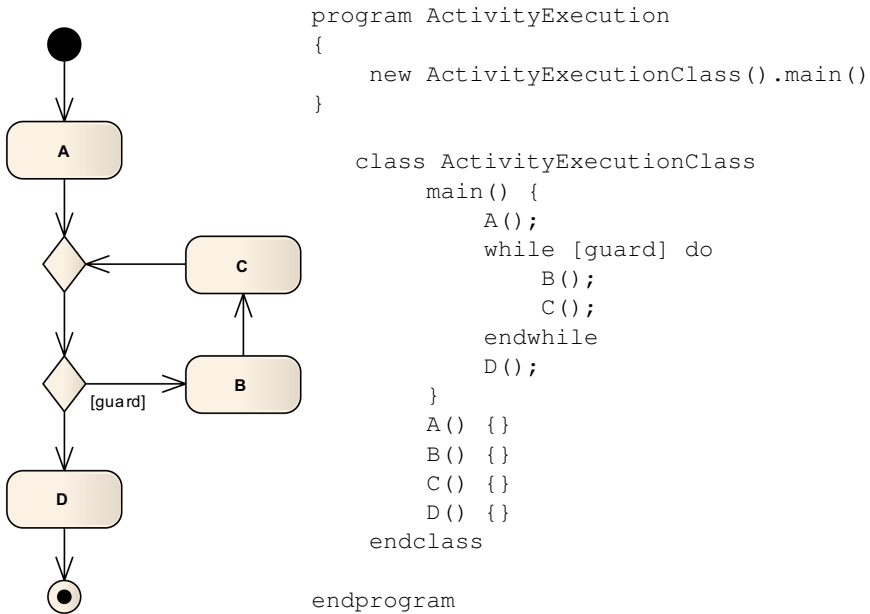


Fig. 2. A well-formed UML Activity and the corresponding TAAL program

- A ForkNode, followed by two building blocks and a closing JoinNode, is a building block (see Fig. 3e).
- Finally, an InitialNode followed by a building block followed by a FinalNode forms a well-formed Activity (see Fig. 3f).

Such well-formed Activities are the starting point for our transformation, which follows the inductive definition of well-formedness:

- Actions are mapped to TAAL operations.
- A sequence of two Actions is mapped to a sequential execution of the corresponding operations.
- A DecisionNode and its MergeNode are mapped to an if-then-else expression.
- A MergeNode followed by a DecisionNode is mapped to a while-do expression.
- A ForkNode followed by a JoinNode is mapped to a forking of methods (i.e., parallel execution of the parts in between the nodes).

The generated code is then embedded into a TAAL program skeleton, i.e., a `main()` method which is owned by a class `ActivityExecutionClass`. This class is instantiated, and the `main()` method is invoked. In the right of Fig. 2, we see the TAAL program corresponding to the UML Activity on the left. The MergeNode-DecisionNode structure of the Activity is translated into a TAAL while loop.

Now that we have given the general idea of our transformation, we want to look into the details of the transformation's realization. Before we can do so, we need to provide

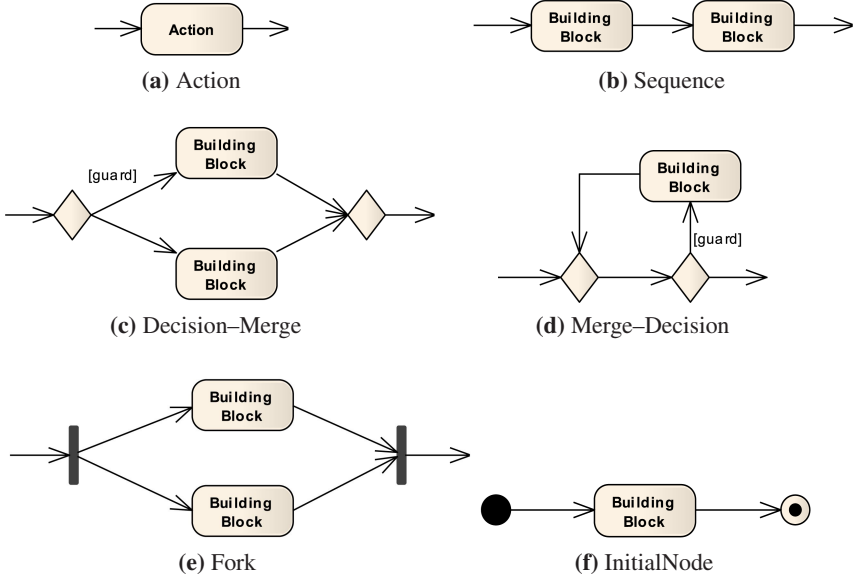


Fig. 3. Different building blocks (a-e) and one well-formed Activity (f)

a couple of prerequisites. As we will see below, the transformation is defined on the abstract syntax level, i.e. the meta-models.

Meta-models. The abstract syntax of the UML is defined by means of a *meta-model*, i.e., a set of class diagrams describing the structure of valid diagram instances. Figure 4a shows the Activity concepts relevant for this paper. The class diagram basically looks as expected: An Activity consists of a number of ActivityNodes which are connected by ActivityEdges. Actions are atomic units of behaviour, and ControlNodes are used to introduce decisions etc. into the modelled flow of execution.

The abstract syntax of the TAAL language is more complex. A TAAL Program consists of a number of Types, one of which is the ObjectType (representing the concept of a class). A Type owns a number of operations, which have a Signature and a Statement representing the body of the operation. There are a number of Statements, including a WhileStat, an ExprStat and a BlockStat used as a container for an arbitrary number of Statements. An operation call is represented by the OperCallExp expression. Figure 4b shows the most important concepts of the TAAL language. Note that for the sake of simplicity, we have omitted a huge number of concepts (including everything related to variables, literals etc.).

Transformation. As we have seen, the abstract syntax of both languages is described by means of meta-models, i.e., class diagrams. Therefore, a valid instance of a UML Activity or a TAAL program can be described as an object diagram which is consistent to the according class diagram. Since object diagrams can be treated as (labelled) graphs, we decided to use *graph transformation rules* (GTRs, [17]) for the specification

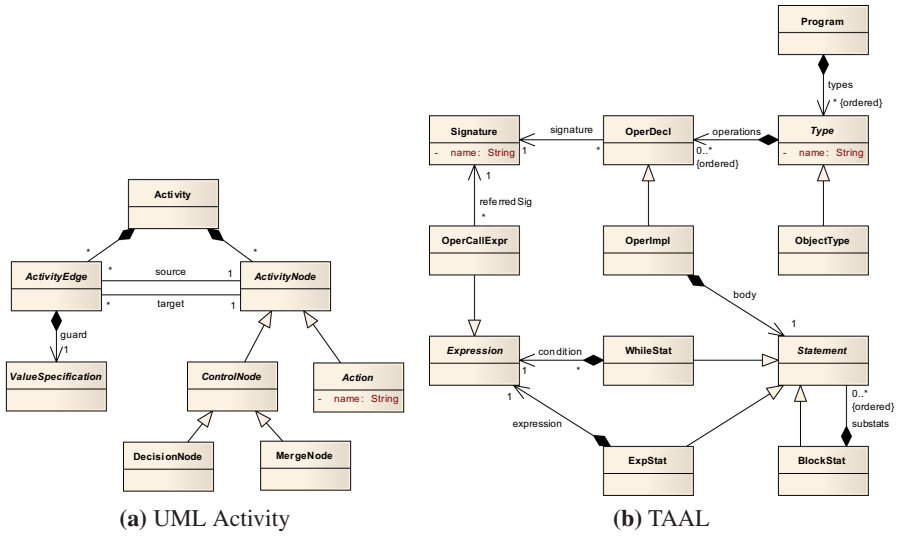


Fig. 4. Excerpt of the meta-models

of our transformation. This approach is a common one for defining model transformations [18], and has—in particular for our undertaking—a number of advantages: First of all, GTRs are specified completely formally; this is important for our final goal of *proving* that our transformation is behaviour preserving. Second, due to their visual appearance, GTRs are relatively easy to understand, and third, the semantics of Activities as well as TAAL programs is specified with GTRs, which will allow us to work with the same formalism for finally proving the correctness of our transformation. Moreover, due to the availability of GTR tools, our transformation is executable.

GTRs performs changes on a so-called *host graph*. They consist of a left-hand and a right-hand graph; if a subgraph similar to the left-hand graph can be found in the host graph, it is replaced by the right-hand graph. In our case, the start graph is the object diagram representing the Activity to be transformed. After a couple of applications of our GTRs, that object diagram is transformed into an object diagram representing the target TAAL program.

GTRs can be presented in two ways: by explicitly showing the left-hand and right-hand graph or by merging them into one graph. In this paper, we have chosen the latter, one-graph approach. This implies that nodes and edges have to be annotated according to their function within the rule. There are 4 types of elements:

- Nodes and edges which remain unchanged are depicted with black, solid, thin lines.
- Nodes and edges created by the rule are depicted with green, solid, fat lines.
- Nodes and edges deleted by the rule are depicted with blue, dashed, thin lines.
- Nodes and edges which must not exist in the host graph for the rule to match are depicted with red, dashed, fat lines.

Having said all that, let us now dive into the details. To specify our transformation, we had to implement 8 main transformation rules. Table 1 shows these rules and briefly

Table 1. Transformation rules and their tasks

Rule name	Task
Activity	Creates TAAL skeleton (Program, class <code>ExecutionClass</code> etc.)
Action_Implementation	Creates (empty) TAAL operation definitions for every Action, adds it to <code>ExecutionClass</code>
Action_Invocation	Creates TAAL operation call for every Action, “wraps” it in <code>BlockStat</code>
Sequence	Merges two <code>BlockStats</code> into one, according to two sequential building blocks
Decision	Creates TAAL if-then-else structure from according <code>DecisionNode</code> and <code>MergeNode</code>
While	Creates TAAL while-do structure from according <code>MergeNode</code> and <code>DecisionNode</code>
Fork	Creates TAAL fork structure from according <code>ForkNode</code> and <code>JoinNode</code>
Initial	Sets the remaining <code>BlockStat</code> as Statement of <code>main()</code> method

states their task within the transformation process. In the following, we will first explain the general idea of our transformation, and we will then by way of example show one of our transformation rules.

The transformation follows the inductive definition of well-formedness of UML Activities. A building block can be translated as soon as its included building blocks have been transformed. On the Activity side this is achieved by *reducing* translated structures to simple, structure-less building blocks. While the UML Activity thus gets simpler and simpler during the transformation, we at the same time build up the corresponding structures on the TAAL side which grows. To remember which building block belongs to which part of the TAAL constructs, we use *correspondence nodes*: Each correspondence node is connected to a UML Activity building block (depicted on the left side) and to the corresponding TAAL construct (depicted on the right side). Note that this approach is inspired by *Triple Graph Grammars* (TGGs, [19]), which explicitly consist of a left-hand graph, a right-hand graph and a correspondence graph associating constructs from the left side with their pendants on the right side. Note also that the concept of building blocks and correspondence nodes are only used within the GTRs; consequently, they do not appear in the meta-models of the two languages.

We want to illustrate this approach with an example transformation rule. Its task is to transform a certain Activity structure into a while loop, and it is depicted as Fig. 5. The rule can be applied if the graph to be transformed contains the structure which can be seen on the left side of the rule. Note the two `ActivityEdges` at the top and at the bottom of the Activity structure: They are the connection to the rest of the Activity and are therefore not deleted.

The part within the two `ActivityEdges` is the actual loop: A `MergeNode` is followed by a `DecisionNode` which has two outgoing `ActivityEdges`: the bottom one is the edge mentioned before, the left edge leads to the body of the loop. This body is in fact a building block – it represents some arbitrary (but well-formed) structure which

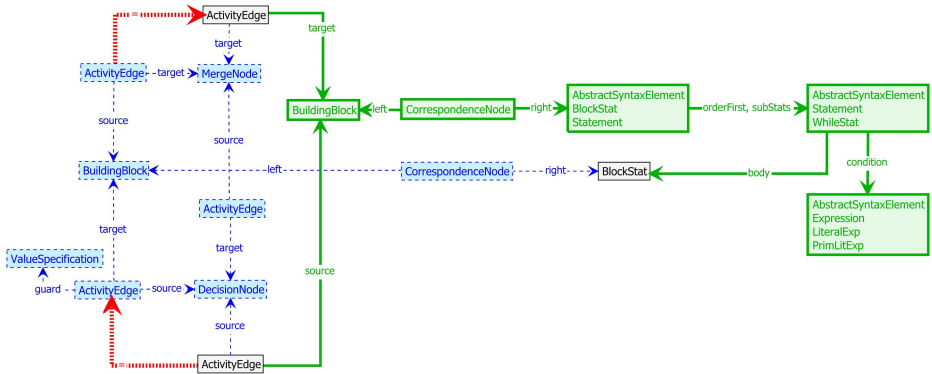


Fig. 5. Rule creating a While loop

has already been transformed. For instance, the building block could have been a single Action; in this case, it would have been a result of applying the **Action_Invocation** rule. It may also have been a more complex structure which has been reduced to one building block by applying a sequence of transformation rules.

Now, note that the building block has an association to a correspondence node, which has a BlockStat node as its right side. That BlockStat node is the result of the one or more transformation steps described above (the ones which finally resulted in the single building block on the correspondence node's left side).

The rule basically performs three changes on the host graph:

1. It creates the TAAL elements forming a while loop, i.e., the WhileStat, a wrapping BlockStat and some elements representing the loop's condition.
2. It sets the BlockStat corresponding to the building block as the body of the while loop.
3. It deletes the loop structure on the Activity side, replaces it with a single building block and creates a new correspondence node associating that building block with the wrapping BlockStat mentioned above.

Similar rules are used to treat simple Actions, sequences of building blocks and the Decision-Merge structure. In addition, we employ the **Action_Implementation** rule to create operation definitions on the TAAL side, the **Activity** rule to create the execution infrastructure like e.g. class definition, and the **Initial** rule to fill the main-method. Together, these rules perform a transformation of a meta-model instance of UML Activities into a meta-model instance of TAAL programs, from which we can then derive the concrete syntax TAAL program.

3 Behaviour Preservation

Recall from the introduction that our final goal is to prove that our transformation is *behaviour preserving*. In this section, we explain the notion of semantic equivalence we have in mind, and we argue on our example that our transformation fulfils this

requirement. To this end, we first of all need to explain the formal semantics of the two languages, and most importantly, fix the notion of *equivalence* used in the comparison (\approx in Fig. 1). As models of behaviour, we use the standard notion of *transition system*.

Definition 1. A transition system (Q, \rightarrow, q_0) over some alphabet A consists of a set of states Q , a transition relation $\rightarrow \subseteq Q \times A \times Q$, and an initial state $q_0 \in Q$. The set of transition systems with alphabet A is denoted $TS[A]$.

Some related notation:

$$q \xrightarrow{a} q' \iff (q, a, q') \in \rightarrow$$

$$q \xrightarrow{a_1 \cdots a_n} q' \iff \exists q_1, \dots, q_{n+1}. q = q_1 \xrightarrow{a_1} q_2 \xrightarrow{a_2} \cdots \xrightarrow{a_n} q_{n+1} = q'$$

A transition system captures the behaviour of a model (or program) if it comprises precisely the execution steps that the model specifies (or the program executes). A single execution step is captured by a transition. A run of the system is captured by a connected sequence of transitions, or in other words, a path through the transition system. Note that the individual transitions, or execution steps, are thought of as atomic; this imposes a limit on the size of the steps that can be captured by a single transition, since on too coarse a level of granularity, the atomicity assumption is not justified (large execution steps may overlap, interfere or be aborted). In consequence, as we will see, we end up with a rather “small-step” semantics.

The mechanism for extracting a transition system from a model is what we call the behavioural semantics of the model (or, more precisely, of the modelling language). In the case of UMLA and TAAL, this mechanism uses the same graphs as the model transformation, and again works by means of graph transformation systems: see [9,12], respectively. In a first step, the static graphs are enhanced to incorporate run-time specific aspects (e.g., a token in the case of UMLA and a program counter in the case of TAAL). A graph transformation system, combined with the start graph that is given by the abstract syntax graph of the model, gives rise to a transition system in the following way:

- Each state is a graph;
- Each transition is the application of a transformation rule, where the label of the transition is given by the name of the applied rule;
- The initial state is given by the start graph;
- Whenever a graph transformation rule is applicable to a state, the corresponding rule application is a transition and the resulting graph is a state.

Thus, every well-formed UML Activity gives rise to a transition system, as does every TAAL program. When, then, do two transition systems describe the *same* behaviour? This is a question that has received much attention, especially in the context of *process algebra* (see [20]). It has become clear that there is no single answer that is satisfactory in all cases; rather, “sameness” can be captured by one of a range of so-called *equivalence relations* over transition systems; see, e.g., [21]. The *weakest* (most liberal) notion of “sameness” is that of *trace equivalence*, which is defined as follows.

Definition 2 (trace equivalence). Assume that the alphabet A is partitioned into a set of internal and external actions, A_{in} and A_{ex} . A trace of a transition system $T \in \text{TS}[A]$ is a sequence $a_1 \cdots a_n$ with $a_i \in A_{\text{ex}}$ for all $1 \leq i \leq n$, for which there is a path

$$q_0 \xrightarrow{w_0 a_1 w_1 a_2 \cdots w_n a_n w_{n+1}} q'$$

such that $w_i \in A_{\text{in}}^*$ for all $0 \leq i \leq n+1$. The traces of T are collected in $\text{Traces}(T)$.

Two transition systems $T_1, T_2 \in \text{TS}$ are trace equivalent, denoted $T_1 \approx T_2$, if $\text{Traces}(T_1) = \text{Traces}(T_2)$.

In our case, the issue of equivalence is more complicated yet: despite the similarity of the semantic definitions of UMLA and TAAL, we cannot directly apply the existing theory, since the transition systems under comparison do not have the same alphabets. The reason for this is simple: as labels we have the rule names of the graph transformation system, and these are different for both languages. Furthermore, and more subtly, the granularity of the execution steps is not the same: in UMLA, executing an activity is based on the movements of tokens, whereas in TAAL it is based on a program counter; these mechanisms obey different rules, and hence moving a token from one activity to the next comprises different steps, in a different order, than moving a program counter from one method invocation to the next.

Our solution to this problem is to identify *one* rule in the graph transformation system for UMLA as well as the one for TAAL that we take to represent the “actual” execution of the action (on the one hand) or method (on the other).¹ For this rule, instead of using the rule name as label in the transition system, we use the name of the action. Thus, the functions sem_{UMLA} and sem_{TAAL} in Fig. 1 map each UMLA resp. TAAL abstract syntax graph to the transition system constructed as per the algorithm above, with the modified transition labelling. All other rule names are interpreted as *internal* in the sense of Def. 2.

The core challenge of our approach is then to prove that for all UMLA graphs G , the following holds:

$$\text{sem}_{\text{TAAL}}(\text{MT}(G)) \approx \text{sem}_{\text{UMLA}}(G) . \quad (1)$$

Our claim is that (1) indeed holds. As an example, Fig. 6 shows the transition systems derived from the example Activity and the resulting TAAL program of Fig. 2. We start with the TAAL transition system (2.b), which exactly looks as expected: The loop can immediately be identified. A closer investigation reveals that the set of traces is also as expected: First, the $A()$ operation is executed, followed by an arbitrary number of executions of the sequence $B() - C()$, and finally the $D()$ method is executed. Note that all this happens within the execution of the $\text{main}()$ operation, i.e., we do not take that operation into account. The set of traces of the TAAL program can thus be described by the regular expression $A(B C)^* D$.

The Activity’s transition system (2.a) looks different, though: It seems to contain two loops. These loops are due to the *traverse-to-completion* semantics of UML Activities [22]. Still, this does not affect the correctness in our chosen criterion: the UMLA transition system gives us exactly the same set of traces over Actions, namely $A(B C)^* D$.

¹For UMLA this is a rule called `action.start()`; for TAAL it is `OperVirtualCallExp`.

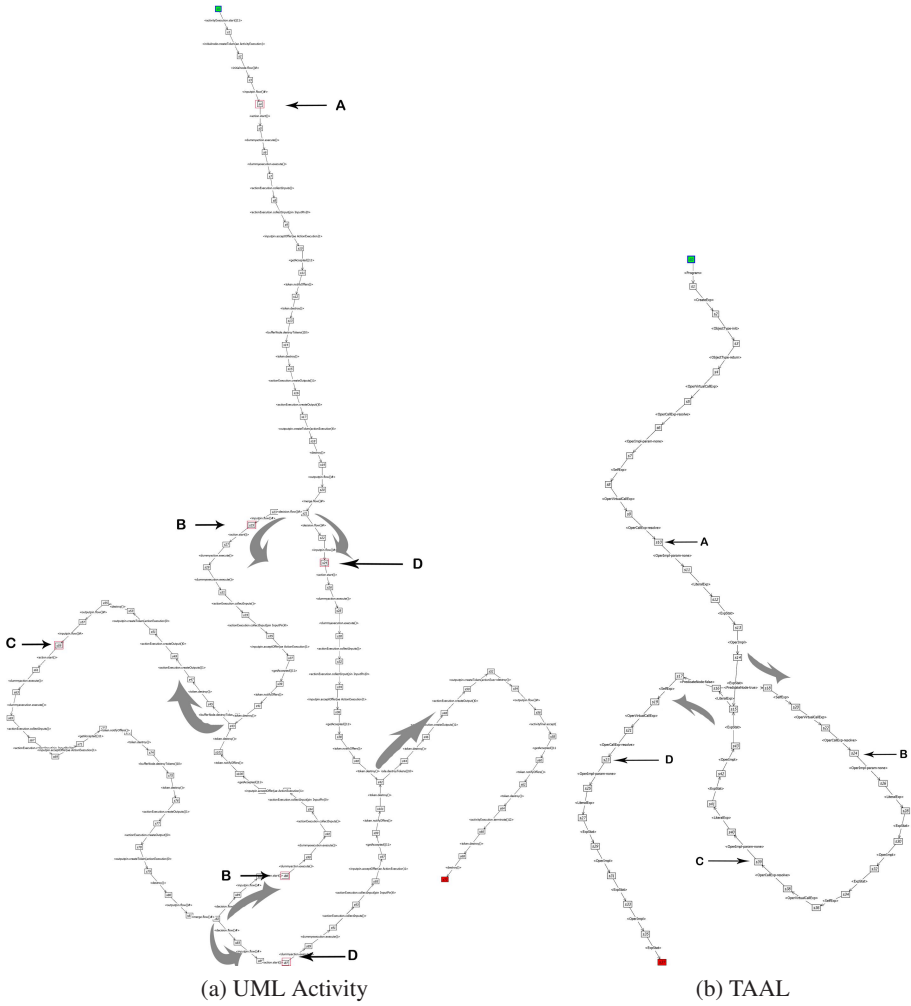


Fig. 6. The transition systems of the example

We have carried out this comparison on a large number of examples, involving different structures of the UML Activity, in particular also with more complex nestings of Decision and Merge nodes. In all of these examples, the resulting transition systems were trace equivalent. Nevertheless, we see this only as a first step towards showing behaviour preservation, and our ultimate aim is a general proof of correctness for the transformation, in the sense of 1.

4 Tool Support

The preceding sections have shown that in our setting, the semantics of UML Activities and TAAL programs as well as our transformation is specified by means of graph

transformation rules. Therefore, we rely on a tool which supports the creation and application of graph transformation rules. As we have mentioned before, we use the tool Groove [11] for this purpose.

For the transformation tool, our main requirement is that the transformation can be specified formally – otherwise, we would not be able to perform a formal proof of correctness. Since most of us already were experienced Groove users, it was an obvious choice to also use that tool for defining the transformation itself. This section will detail our reasons for that choice, and it will point out some particular strengths of Groove.

Despite the “standard” graph transformation features like creation and deletion of nodes and edges, Groove supports some more advanced concepts which allowed us to specify our transformation as desired. First, Groove supports *attributed graphs*, which e.g. allowed us to create TAAL operations having the same name as their corresponding Actions. Additionally, we were able to specify the **ActionImplementation** rule in such a way that one TAAL operation is created for every *name* of an Action; for an Activity that contains two Actions named *A*, this results in a TAAL program with one *A* operation, but two invocations of that operation.

Second, a powerful notion of *universal quantification* has been implemented in Groove. In a nutshell, this means that rules can be written which manipulate *all* occurrences of a node in a certain context. While implementing our transformation, this was of particular importance for the **Sequence** rule: Recall from Sect. 2 that this rule merges two BlockStats into one, and part of this is to add all sub statements of one BlockStat to the resulting BlockStat. Universal quantification allowed us to implement this behaviour within one rule.

Another reason for choosing Groove was that the transformation rules we defined basically relate parts of a UML Activity with their corresponding parts on the TAAL side, in contrast to an operational transformation specification (e.g. in Java). Since relating elements of source and target models will probably be an important part of our proof, we hope to reuse the transformation rules for this purpose.

Figure 7 shows a screenshot of Groove. On the left side, the names of the transformation rules can be seen. Note that at the bottom of the rule’s compartment, a couple of rules are shown whose names start with “Failure”. These rules match if certain structures exist in a state which would indicate that the transformation has failed. Note also that these rules have a priority of 0: This makes sure that the failure rules can only match if none of the transformation rules matches any more (i.e., after the complete transformation has been carried out).

The big compartment on the right shows the start state representing the Activity as introduced in Fig. 2. Note that Groove allows to hide parts of the displayed graph; we have hidden the Activity node and its edges to the Activity’s element to reduce the complexity of the graph’s visualisation. Note also the DMMSystem node to the left of the graph: This node and the associated Invocations are needed for the graph transformation rules describing the Activity’s semantics. They are deleted by our transformation.

In order to use Groove, we translated the UML Activity under consideration into a suitable format. For this, we have written an Eclipse [23] plugin which takes a UML Activity model as input and generates a Groove state graph out of it. The Activity is given in the XMI format which is then read and processed using the API of the Eclipse

frequently employed in a model-based development, and thus their correctness constitutes an important part of MDD. Our ultimate goal and future work is a formal proof of correctness.

The contribution of this work does, however, go beyond this specific transformation. Although the two modelling languages are conceptually very different, a comparison can be carried out. There are a number of important issues which helped towards this goal. First of all, it is the existence of meta-models (of the same language) which facilitated the definition of the transformation. Secondly, indispensable for a correctness proof is (a) a formal definition of the transformation (here given because of the use of graph transformation systems) and (b) a formal definition of the semantics of the languages. Crucial is also the (formal) definition of the employed notion of equivalence; for this, a *common* semantic domain of the languages is important. Last but not least, such a comparison would not have been possible without a tool for executing the model transformation.

The method proposed in this paper for the comparison of behavioural semantics is obviously only applicable if the modelling languages in question are indeed behavioural. Moreover, it should be possible to express their semantics by means of transition systems. Fortunately, the transition system formalism is itself quite general, so we do not expect this to be a limiting factor.

References

1. Akehurst, D.H., Kent, S.: A Relational Approach to Defining Transformations in a Meta-model. In: Jézéquel, J.-M., Hussmann, H., Cook, S. (eds.) UML 2002. LNCS, vol. 2460, pp. 243–258. Springer, Heidelberg (2002)
2. Czarnecki, K., Helsen, S.: Classification of Model Transformation Approaches. In: Workshop on Generative Techniques in the Context of Model-Driven Architecture (2003)
3. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification (2007), <http://www.omg.org/cgi-bin/doc?ptc/2007-07-07>
4. Küster, J.M., Abd-El-Razik, M.: Validation of Model Transformations - First Experiences Using a White Box Approach. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, pp. 193–204. Springer, Heidelberg (2007)
5. Ehrig, H., Ehrig, K., de Lara, J., Taentzer, G., Varró, D., Varró-Gyapay, S.: Termination Criteria for Model Transformation. In: Cerioli, M. (ed.) FASE 2005. LNCS, vol. 3442, pp. 49–63. Springer, Heidelberg (2005)
6. Lambers, L., Ehrig, H., Orejas, F.: Conflict Detection for Graph Transformation with Negative Application Conditions. In: Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G. (eds.) ICGT 2006. LNCS, vol. 4178, pp. 61–76. Springer, Heidelberg (2006)
7. Mens, T., Demeyer, S., Janssens, D.: Formalising Behaviour Preserving Program Transformations. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 286–301. Springer, Heidelberg (2002)
8. Ruhroth, T., Wehrheim, H.: Refactoring Object-Oriented Specifications with Data and Processes. In: Bonsangue, M.M., Johnsen, E.B. (eds.) FMOODS 2007. LNCS, vol. 4468, pp. 236–251. Springer, Heidelberg (2007)
9. Kastenberger, H., Kleppe, A., Rensink, A.: Defining Object-Oriented Execution Semantics Using Graph Transformations. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 186–201. Springer, Heidelberg (2006)

10. Object Management Group: Business Process Modeling Notation V1.1 (2008), <http://www.omg.org/spec/BPMN/1.1/PDF>
11. Rensink, A.: The GROOVE Simulator: A Tool for State Space Generation. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 479–485. Springer, Heidelberg (2004)
12. Hausmann, J.H.: Dynamic Meta Modeling. PhD thesis, University of Paderborn (2005)
13. Karsai, G., Narayanan, A.: On the Correctness of Model Transformations in the Development of Embedded Systems. In: Kordon, F., Sokolsky, O. (eds.) Monterey Workshop. LNCS, vol. 4888, pp. 1–18. Springer, Heidelberg (2006)
14. Glesner, S.: Using Program Checking to Ensure the Correctness of Compiler Implementations. J. UCS 9(3), 191–222 (2003)
15. Denney, E., Fischer, B.: Certifiable Program Generation. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 17–28. Springer, Heidelberg (2005)
16. Peterson, W.W., Kasami, T., Tokura, N.: On the Capabilities of While, Repeat, and Exit Statements. Commun. ACM 16(8), 503–512 (1973)
17. Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformations. Foundations, vol. 1. World Scientific, Singapore (1997)
18. Klar, F., Königs, A., Schürr, A.: Model Transformation in the Large. In: Crnkovic, I., Bertolino, A. (eds.) ESEC/SIGSOFT FSE, pp. 285–294. ACM, New York (2007)
19. Schürr, A.: Specification of Graph Translators with Triple Graph Grammars. In: Mayr, E.W., Schmidt, G., Tinhofer, G. (eds.) WG 1994. LNCS, vol. 903, pp. 151–163. Springer, Heidelberg (1995)
20. Bergstra, J.A., Ponse, A., Smolka, S.A.: Handbook of Process Algebra. Elsevier, Amsterdam (2001)
21. van Glabbeek, R.J.: The linear time – branching time spectrum I: The semantics of concrete, sequential processes. In: [20], pp. 3–100
22. Bock, C.: UML 2 Activity and Action Models, Part 4: Object Nodes. Journal of Object Technology 3(1), 27–41 (2004)
23. Eclipse Foundation, <http://www.eclipse.org>

A Practical MDA Approach for Autonomic Profiling and Performance Assessment

Fabio Perez Marzullo¹, Rodrigo Novo Porto¹, Divany Gomes Lima¹,
Jano Moreira de Souza¹, and José Roberto Blaschek²

¹Federal University of Rio de Janeiro – UFRJ, COPPE Database Laboratory,
Rio de Janeiro, RJ, Brazil

{fpm,rodrigo,dglima,jano}@cos.ufrj.br

²State University of Rio de Janeiro – UERJ, Rio de Janeiro, RJ, Brazil
blaschek@attglobal.net

Abstract. By releasing their Model Driven Architecture (MDA) as a new standard, the Object Management Group (OMG) [1] proposed a new development concept toward existing traditional paradigms. It set a new exciting research area in which it would be possible to develop truly independent and powerful programming environments capable of achieving new levels of productivity, performance and maintainability. With this goal in mind, this paper describes a research conducted with the purpose of improving database performance through the union of autonomic computing aspects and MDA. It is widely accepted that the model development approach is gaining importance in IT projects today; therefore the technique discussed here, presents a way of assessing performance, identifying flaws and improving software activities in order to create a self-managed environment. With new defined stereotypes and tagged values; in conjunction with profiling libraries, and relying on autonomic aspects, the proposed extension enables code generation in order to conduct a thorough set of performance analysis, indicating the best suitable database configuration for a given functionality. After setting the underlying problem, explaining tools configuration and concepts and describing the profiling technique, it presents a study based on a real project conducted by the Brazilian Ministry of Defense.

Keywords: MDA, Database, Profiling, benchmarking, performance testing, code generation.

1 Introduction

Research conducted in recent years has shown that model approaches are becoming essential tools in software development [1]. This new paradigm implies on new ways of analyzing overall software performance. Attempts to integrate performance analysis with MDA have been done, and are still in course [3]. However, such attempts present mostly ways to generate test code regarding general software engineering aspects.

The lack of an efficient representation for Database profiling has motivated us to extend the UML models (as implemented in the development environment) in order

to cope with undesirable performance situations along software execution. Therefore, the idea relies on an extension with the ability to mark business domain models with stereotypes and tagged values, seeking to understand database performance aspects.

Despite what is being done, there are still gaps involving proper database analysis. Still, different approaches focusing on benchmarking MDA techniques [5], or on the testing and verification of model transformation [6] are currently in activity, and our work comes to join all efforts to upgrade the MDA world.

Although MDA promises to implement a self-sufficient model driven development theory, supported now by a few important companies and tools, it still needs to address several specific development aspects, including an efficient and complete set of Meta Object Facility [7] models and objects to address database specifics.

Also, as stated by the IBM Autonomic Computing Manifest, we have come to a point at which the IT industry creates powerful computing systems on a daily basis. In order to make individuals and business more productive, by automating their practices and processes, we face paths ahead showing that the autonomic approach might prove to be valuable in the near future [21].

By pursuing the vision of creating intelligent solutions with self-management capabilities, the proposed extension, defines a set of general rules and techniques that creates a self-configuration and self-assessment environment, capable of identifying the best system-database interaction according to the functionality involved.

2 Problem Definition

Since the beginning of computational growth, companies are engaged in creating tools to aid software profiling; creating autonomic algorithms; and enabling developers to identify flaws and re-factor problematic systems [8, 9 and 10].

The promise of an efficient model driven development and the comparable effort of researching good autonomic practices have gained many adepts. The AdroMDA project [2] is one of few important model development tools, based on the OMG MDA specification [1]. From single CRUD (Create, Read, Update and Delete) application to complex enterprise applications, it uses a set of ready-made cartridges (which implements highly widespread of Java development API) to automatically generate up to 70% of software source code. It is expected that all efforts should point to an interesting future where the modeling and coding stages will be merged without significant losses.

The same horizon is faced by autonomic computing experts. Although it might be considered a young research area, the autonomic computing perspective has brought to discussion the necessity of creating new techniques to cope with Information Technology increasing complexity.

Given the foregoing, it is necessary to explain the problematic scenario that we have faced with in our project. Our projects use an MDA Framework composed by the AndroMDA environment, the Maven tool [16] and the Magic Draw Case Tool [24]. The AndroMDA environment comprises of a set of specialized cartridges that are used to generate the software code. One of its cartridges is the Hibernate Cartridge, which is responsible for generating all the database manipulation code. As for database code generation all seemed perfect, the development database was used

to test the system while as it was being constructed. Only three to five users were simultaneously accessing, the database and development seemed to go smoothly. When the system was deployed and put into production, a small set of functionalities presented longer response times than we had anticipated. The production environment needed to be accessed by 10 to 20 users simultaneously and had to deal with thousands, even millions of records to be queried or joined. It suffices to say that performance degradation perception was immediate, and as such, something should be done to locate and solve the root problem.

To solve it we began by stating three topics that needed to be immediately addressed:

1. How to isolate the functionalities and their problematic attributes?
2. How to solve these problems in an elegant and transparent way for the end user?
3. How to automate the solution extending the UML objects to address Database Testing and Analysis?

Using the current infra-structure to address the database performance analysis was not the best way, since we wanted to create a systematic and self-sustained analysis approach. It was necessary to adjust the Hibernate Cartridge, and implement a new Profiling Cartridge, embedding autonomic aspects, in order to create a self-configuring, self-healing and self-optimization infrastructure, as explained in the next section.

3 The Profiling Extension

The profiling extension process was straightforward. We needed to create new design elements to indicate when it would be necessary to embed monitoring code inside the software. For that purpose we followed a five-step design process, in order to extend the MDA framework:

1. Defining the profiling library. This stage was responsible for identifying the profiling libraries that would be used. Our objective was to create an infra-structure that could cope with any profiling library available, so we decided to use design patterns to enable the ability to plug the library as necessary. This technique is explained in the following sections. However, after analyzing several libraries two appeared to be the best choices: the JAMon Library [11] and the InfraRED Library [15].

2. Defining Stereotypes. This stage was responsible for creating all the stereotypes necessary to model configuration. For each feature available a stereotype was created:

1. *API timing:* Average time taken by each API. APIs might be analyzed through threshold assessment and first, last, min, max execution times:

→ <<APIView>>: which enables api monitoring;

2. *JDBC and SQL statistics:* Being the objective of our research, we created the following stereotype for accessing JDBC and SQL statistics:

→ <<SQLQueryView>>: which enabled the displaying of the created hibernate queries;

3. *Tracing and Call Information*: responsible for showing statistics for method calls. The following stereotype was created to mark/enable this option:

→ `<<TraceView>>`: which enabled a detailed call tracing for method calls;

3. *Defining Tagged Values*. This stage was responsible for creating all tagged values necessary to support and configure stereotypes:

1. `@profiling.active`: defines whether the profiling activities are going to be executed. Setting its value to “yes”, implies generating the profiling code, and consequently enabling the profiling activities; setting to “no”, the code will not be generated. Applies to `<<APIView>>`, `<<SQLQueryView>>` and `<<TraceView>>` stereotypes;
2. `@profiling.apiview.starttime`: Defines the starting time at which the profiling code should initiate monitoring. For example: it is not interesting to monitor every aspect of the system, therefore we indicate the minimum limit in which the profiling code should initiate the logging process. Applies to the `<<APIView>>` stereotype;
3. `@profiling.sqlqueryview.delaytime`: Defines the starting time in which the profiling code should initiate SQL monitoring. For example: it is not interesting monitoring all query execution in the system. It is only necessary to assess queries that surpass a certain delay threshold. Applies to the `<<SQLQueryView>>` stereotype;
4. `@profiling.traceview.delaytime`: Defines the starting time at which the profiling code should initiate Trace monitoring. For example: it is not interesting to monitor all method calls in the system. It is only necessary to assess the calls that surpass a certain delay threshold. Applies to the `<<TraceView>>` stereotype;

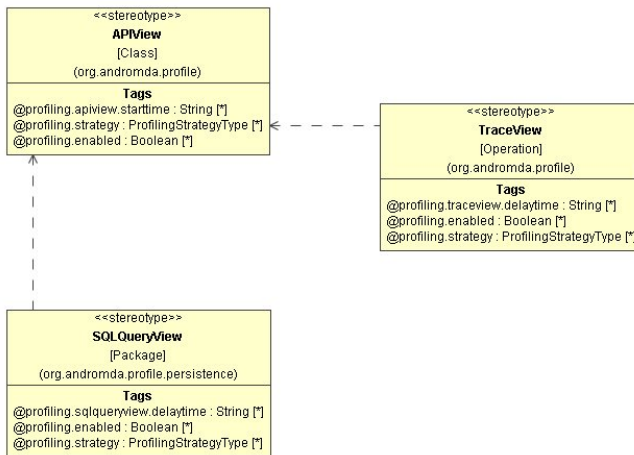


Fig. 1. Stereotypes and tagged values hierarchy

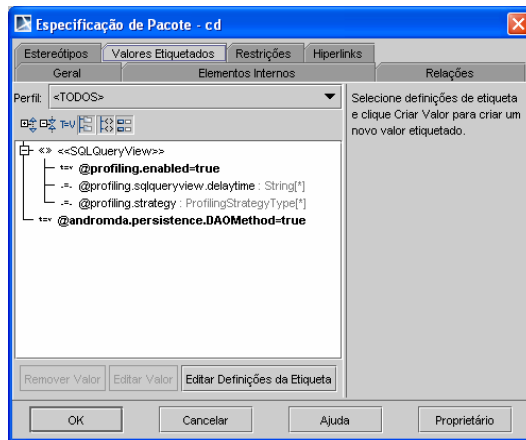


Fig. 2. Associating the stereotype to the persistence package

4. Adjusting the Hibernate Cartridge. This stage was responsible for extending the hibernate cartridge so it would be able to cope with the new stereotypes. The Hibernate API works through the concept of Sessions, where a set of instructions, referring to database interactions, creates the notion of an unique Session. Normally, when a database connection is opened, a Hibernate Session is opened to accommodate that database connection. It stays open as the hibernate continues to access the database. Given those rules, we realized that we would have, not only to analyze the software according to a class-based scope but we needed a broader and more complete approach in which the profiling would be able to gather information globally. The profiling scope should reach the entire persistence layer. This constraint motivated us to design the stereotype as having a package scope, not only class scope.

The solution involved the extension of the AndroMDA persistence base package (andromda-profile-persistence) in order to include support for our profiling stereotype and tagged value. That allowed us to embed the necessary profiling code in the hibernate generated code.

5. Creating new configurable profiling cartridge. This stage was responsible for creating the profiling cartridge responsible for generating the monitoring code. The cartridge is explained in the following section.

Next, an autonomic environment should be set in order to create the self-managed system-database relationship. This environment was based on the following aspects:

1. Self-Configuration Aspect. This aspect was responsible for identifying the best hibernate session configuration according to functionality's database access needs. Our objective was to create an infra-structure that could cope with the system demands, such as table joins and cache, and were also able to remember this configuration after resources have been released and/or system has been shutdown.

2. Self-Optimization Aspect. This aspect was responsible for identifying the best configuration values for previously detected attributes. It should also persist long after system shutdown.

3. *Self-Healing and Self-Protection Aspects.* These aspects are not specifically attended in this implementation. It is still under development. There is enough literature addressing connection breakdown, web server crashes recovery and network disconnection. IBM Tivoli [22] is one that covers such elementary hardware issues.

4 The Autonomic Profiling Strategy

The autonomic profiling approach was built on top of the AndroMDA framework [2]. According to the background presented previously, we had two profiling libraries candidates to use. Both presented efficient and complementary functionalities and we decided to design a way to interchange both libraries as necessary. The configuration strategy, as the term describes, was to use the well known *strategy* pattern. With this approach, we managed to set the profiling library by means of a tagged value:

- @profiling.strategy: defines which library should be used during profiling analysis. Assigned values are: {JAMON, INFRARED}.

The specification of the profiling configuration details the functional relation between the MDA, as the source model, and the profiling technique, as the target implementation. This configuration method enabled us to interchange any profiling library of interest.

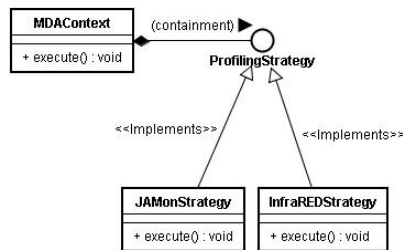


Fig. 3. illustrates the strategy pattern of interchanging profiling libraries and profiling configuration

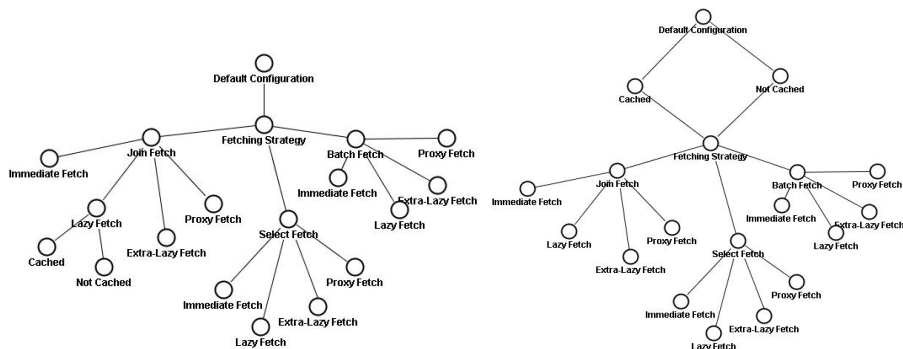


Fig. 4. Decision Tree for configuration definition. Configuration takes into consideration a fetching strategy first and a cache strategy first.

Also, the configuration process was responsible for identifying the best system-database relationship for the specified functionality. The configuration and optimization processes are best explained in figures 5 and 6:

The overall idea is this: (1) the hibernate configuration initiates in its default values; (2) when the source code is generated, it embeds all the autonomic code that will measure any configuration attribute and evaluate it to find the best performance value; (3) the profiling code creates a decision tree and identifies the best configuration for that functionality; it first evaluates a fetching strategy as first perspective and after a cache strategy as first perspective. After acquiring the assessment measures, it identifies the best perspective solution; and (4) it then stores the configuration for future access, whenever that functionality is called.

Additionally, we needed to guarantee that the configuration kept optimized through software execution. Meaning that, when in production, the system database would probably have the number of records increased, and join strategies could have its execution time affected. Therefore, the autonomic configuration policy should be re-executed and reconfigured in order to guarantee a best perspective configuration.

For example, whenever a new software release was put into production the configuration process was executed and used as base configuration for accessing functionality's data. This value is then used for assessing whether the actual execution time is well adjusted with the stored configuration. So, whenever the functionality was called, a comparison was made between the actual time and the configured time. If the actual time falls below or above 20% of the configured value the system reruns the configuration process searching for a new configuration more suitable for that functionality.

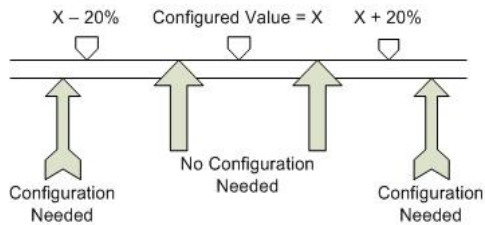


Fig. 5. This figure presents the range of values used to guarantee that the best accessing configuration was kept along system use

5 Profiling Analysis and Results

For the definition of the analysis process, we must understand the execution scenario. The development was focused on creating a Web-based application, using Struts [19], running on the JBoss Application Server (JAS) [20], and accessing a Relational Database through Hibernate API [13, 14]. System modeling was done using an UML tool, the AndroMDA, and the Maven tool [16], and for Java development we used the Eclipse IDE [17]. The data load, was irrelevant at development time, but it became crucial by the time the system was put into production. The generated Hibernate code and configuration did not comprise with the system's response time non-functional requirements.

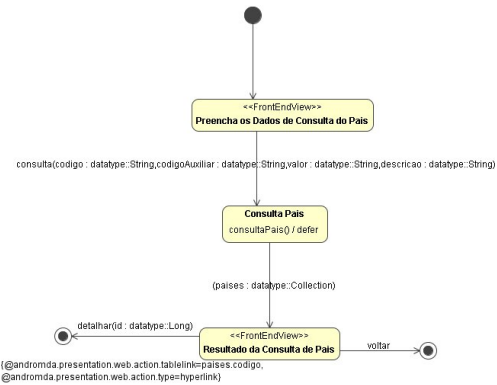


Fig. 6. Use Case used as Test Case

As soon as the system performance started to interfere with overall system usability, we found urgent to locate the bottleneck source and recreate all involved models. On the other hand, we needed to ensure that the new modeling would not create new bottleneck sources.

The profiling technique enabled us to embed monitoring code that gathered all necessary information and pointed the features that needed to be remodeled. The analysis below shows what the profiling technique is capable of.

Statistics were acquired along two different scenarios: (1) less than 10,000 registers in non-optimized and optimized environments; (2) more than 2,000,000 registers with non-optimized and optimized environments. After performing measurements in both scenarios, we came up with the averages presented bellow. Tables 1 and 2 present three functionalities that were assessed to test the MDA-Profiling strategy:

Table 1. Non-Optimized Environment: Identifying problematic functionalities. Functionalities F2 and F3 show that the real execution time were alarmingly out of range. Functionality names were omitted for confidentiality purposes. Time values averages were rounded up.

Functionalities	Expected Execution Time	Actual Execution Time	Problematic Functionality
Single Table (F1)	10 to 50 ms	30 ms	No
Join with 2 Tables (F2)	100 to 1000 ms	50,000 ms	Yes
Join with 3 Tables (F3)	500 to 1000 ms	10,000,000 ms	Yes

Table 2. Optimized Environment after reconfiguration: Solving problematic functionalities. After database access optimization, functionality F3 still presented an execution time 100% higher the estimate one. We assumed it as estimation error and corrected the non-functional requirement specification. Functionality names were omitted for confidentiality purposes. Time values averaged were rounded up.

Functionalities	Expected Execution Time	Actual Execution Time
Single Table (F1)	10 to 50 ms	23 ms
Join with 2 Tables (F2)	100 to 1000 ms	1000 ms
Join with 3 Tables (F3)	500 to 1000 ms	3000 ms

Table 3. Configuration process for functionality F3. The best path ended with the select fetch strategy and the extra-lazy fetch type, noticing that it was configured on a cache first perspective.

Cache Strategy		ON		
Fetch Strategy	Fetch Type	Time (in seconds)		Selection
Join Fetch		Average (Total)	(Hits)	
	Immediate	50 (250)	5	
	Lazy	16 (81)	5	
	Extra-Lazy	14 (70)	5	
	Proxy	12 (60)	5	
	Batch Fetch			
	Immediate	6 (30)	5	
	Lazy	20 (100)	5	
	Extra-Lazy	20 (100)	5	
	Proxy	8 (40)	5	
	Select Fetch			
	Immediate	6 (110)	21	X
	Lazy	3 (30)	11	
	Extra-Lazy	3 (30)	12	
	Proxy	5 (50)	12	

Each system functionality was assigned an estimated execution time, based on system’s non-functional requirements, as shown in Table 2. The system was deployed in production environment and measured without the optimized code. Those functionalities that had overcome the estimated value by 20% were identified as problematic.

After generating the code and deploying the system, the optimization process started searching for the best configuration possible. At this time, we were able to guarantee that the performance observed that functionalities F2 and F3 were the best that the configuration aspects could offer.

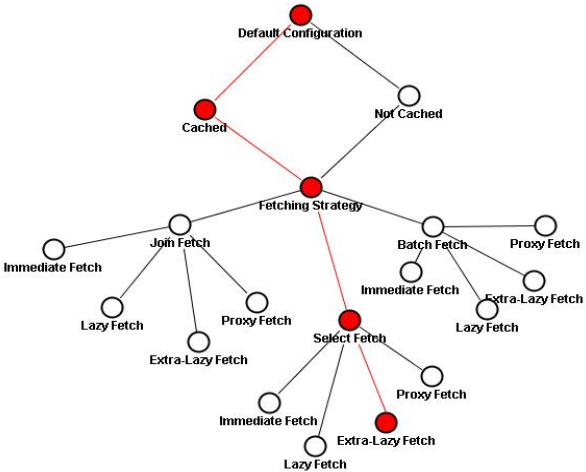


Fig. 7. The best path for functionality F3 in the decision tree. Both Lazy and Extra-Lazy were candidates, but the number of hits (which means the number of database access) where lower for Lazy, therefore the overall average was higher. Values were rounded up by the profiling library.

Values presented in table 2 show that the new optimized code reduced significantly the execution time. It showed a large reduction in database resource consuming, taking into consideration that in a web-based environment such optimization might prove to be extremely relevant.

6 Conclusions

This paper presented a new approach for using profiling techniques in MDA development. Our contribution aimed at creating a MDA extension to help identifying and solving performance problems regarding information system and database (data warehouse) communications. In addition, we defined an extensible, easy-to-use profiling infra-structure that can be configured to execute different profiling libraries and techniques, obtaining a more complete set of results.

The analysis results have validated the autonomic profiling approach and proved that the MDA extension might be used to analyze the code as soon as it is deployed. The initial effort to create the infra-structure proved laborious although following developments shall not suffer the same problems as it has already been implemented and added to the AndroMDA features.

Finally, the intention was to obtain development information in order to allow developers and analysts to make proper decisions regarding software design. According to the analysis results, the extension was able to expose flaws and delays during system execution, and, consequently promote the necessary corrections to ensure that the generated code was reliable and optimized in both scenarios.

References

1. OMG, Model Driven Architecture (2007), <http://www.omg.org/mda>
2. AndroMDA, v3.0M3 (2007), <http://www.andromda.org/>
3. Zhu, L., Liu, Y., Gorton, I., Bui, N.B.: MDAbench, A Tool for Customized Benchmark Generation Using MDA. In: OOPSLA 2005, San Diego, California, October 16-20 (2005)
4. OMG, UML 2.0 Testing Profile Specification (2007), <http://www.omg.org/cgi-bin/doc?formal/05-07-07>
5. Rodrigues, G.N.: A Model Driven Approach for Software System Reliability. In: Proceedings of the 26th International Conference on Software Engineering (ICSE 2004). IEEE, Los Alamitos (2004)
6. Lamari, M.: Towards an Automated Test Generation for the Verification of Model Transformations. In: SAC 2007. ACM, New York (2007)
7. OMG, Meta Object Facility (2007), <http://www.omg.org/mof>
8. Eclipse Test & Performance Tools Platform Project (2007), <http://www.eclipse.org/tptp/>
9. EJ-Technologies, JProfiler (2007), <http://www.ej-technologies.com/products/jprofiler/>
10. JAMon (Java Application Monitor) (2007), <http://jamonapi.sourceforge.net/>
11. NetBeans Profiler (2007), <http://profiler.netbeans.org/>

12. Frankel, D.S.: Model Driven Architecture – Applying MDA to Enterprise Computing. OMG Press, Wiley Publications (2003)
13. Hibernate (2007), <http://www.hibernate.org>
14. Bouer, C., King, G.: Hibernate in Action. Manning Publications Co. (2004)
15. InfraRED – Performance and Monitoring Tool for Java (2007),
<http://sourceforge.net/projects/infrared/>
16. Maven project management and comprehension tool (2007),
<http://maven.apache.org>
17. Eclipse Project (2007), <http://www.eclipse.org>
18. Velocity Project (2007), <http://velocity.apache.org/>
19. Struts Project (2007), <http://struts.apache.org/>
20. JBoss Application Server (2007), <http://www.jboss.org/>
21. Autonomic Computing – IBM’s Perspective on the State of Information Technology. IBM (2007), <http://www.ibm.com/research/autonomic/>
22. Tivoli Software, IBM (2007),
<http://www-306.ibm.com/software/br/tivoli/>
23. Centro de Catalogação das Forças Armadas - CECAFA (2007),
<http://www.defesa.gov.br/cecafa/>
24. No Magic Inc., Magic Draw Case Tool (2007), <http://www.magicdraw.com>

Ladder Metamodeling and PLC Program Validation through Time Petri Nets^{*}

Darlam Fabio Bender^{1,2}, Benoît Combemale¹, Xavier Crégut¹, Jean Marie Farines²,
Bernard Berthomieu³, and François Vernadat³

¹ Institut de Recherche en Informatique de Toulouse (CNRS UMR 5505)
Université de Toulouse. France

`first_name.last_name@enseeiht.fr`

² Departamento de Automação e Sistemas

Federal University of Santa Catarina. Florianopolis, Brazil

`last_name@das.ufsc.br`

³ Laboratoire d'Analyse et d'Architecture des Systemes (CNRS)

Université de Toulouse. France

`last_name@laas.fr`

Abstract. Ladder Diagram (LD) is the most used programming language for Programmable Logical Controllers (PLCs). A PLC is a special purpose industrial computer used to automate industrial processes. Bugs in LD programs are very costly and sometimes are even a threat to human safety. We propose a model driven approach for formal verification of LD programs through model-checking. We provide a metamodel for a subset of the LD language. We define a time Petri net (TPN) semantics for LD programs through an ATL model transformation. Finally, we automatically generate behavioral properties over the LD models as LTL formulae which are then checked over the generated TPN using the model-checkers available in the Tina toolkit. We focus on race condition detection.

1 Introduction

Actually, verifications of Programmable Logical Controllers (PLCs) programs are made through exhaustive testing. This method takes a long time to be executed and some errors may pass unnoticed in complex systems.

Ladder Diagram (LD) programs are difficult to debug and modify because its graphical representation of switching logic obscures the sequential, state-dependent logic inherent in the program design [1]. Not found bugs in PLC programs are often very costly and sometimes are even a threat to human safety. This work aims to provide a framework for automatic formal verification of PLC programs written in LD language.

To perform the formal verification of LD programs we introduce a model driven approach. LD models are designed according to an LD metamodel. These LD models are then translated into a formal language. The generic LTL properties to be verified are

^{*} This work is supported by the TOPCASED project, part of the french cluster *Aerospace Valley* (granted by the french DGE), cf. <http://www.topcased.org>

generated automatically, and finally we rely on model-checking to verify the temporal properties. For this work we have chosen the time Petri nets (*TPN*) as formal language, and the *Tina*¹ toolkit for simulation and model-checking. We focus on the verification of generic, i.e. model independent, properties, especially the absence of race conditions in LD programs.

This paper is organized as follows. Section 2 introduces the PLCs, the LD language and the validation issue. Section 3 presents the approach used to formally verify LD programs. Section 4 presents the related work and Section 5 concludes and presents some perspectives and future work.

2 PLCs, Ladder Diagrams and Validation Issue

2.1 Programmable Logical Controllers (PLCs)

A PLC is a special purpose industrial computer used to automate industrial processes. It can be connected to several inputs and outputs, and can be programmed to control the state of the outputs depending on the configuration of the inputs and its internal state.

The PLC execution follows a cycle. The state of all inputs is copied into memory. Then, the internal program runs and creates in memory a temporary table of all outputs. When this program completes, the table is written to the outputs and a new cycle starts. It repeats as long as the PLC is running.

A PLC can be programmed using any of the five languages [2] which are: Instruction List (*IL*), Structured Text (*ST*), Function Blocks Diagrams (*FBD*), Sequential Function Chart (*SFC*) and Ladder Diagram (*LD*). The semantics of these languages is not strictly defined, certain definitions are missing or contain ambiguities. Some research work has been made to solve these ambiguities, e.g. [3]. This article focuses on the LD language, but the same approach could be used for the other languages.

2.2 Ladder Diagram (LD)

Ladder Diagram is the most used language for programming PLCs. It is a graphical language where the basic elements are based on an analogy to physical relay diagrams [4], so it does not represent a big paradigm shift for technicians that are not used to the new computer techniques and programming languages. In Figure 1 is represented a simple LD program in the textual concrete syntax (ASCII art) normally used. We can look at it as a relay diagram where the power flows from the left to the right, passing through the inputs to activate the outputs. The program is delimited by a vertical line on the left representing the *hot* wire, and another one on the right representing the neutral wire. These vertical lines are called the left and right rails.

The horizontal lines (rungs) and the associated elements represent boolean equations. The dependent element of the equation (coil) is represented by the symbol "()", while the independent elements (contacts) are represented by "I I". A diagonal line is placed in the middle of symbols as in "I/I" to indicate that the negated value of the variable is used. Variables placed in series represent the *AND* boolean function while variables

¹ <http://www.laas.fr/tina/>

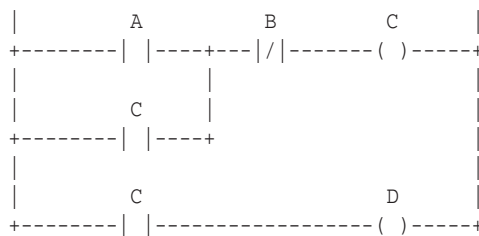


Fig. 1. Simple Ladder Diagram example

placed in parallel represent the *OR* boolean function. The rungs are executed in order from the top to the bottom. So the program in Figure 1 represents the boolean equations $C = (A \vee C) \wedge \neg B$ and $D = C$. A variable can be used as a coil and as a contact in the same rung, that is the case of variable C in the first equation.

The elements presented so far are the basic elements of LD. A program may also contain more complex elements, the functions and function blocks (*FB*). Functions always produce the same outputs when provided with the same inputs while *FBs* keep an internal state that also influences the outputs. They are represented by a rectangular box with their names inside, their inputs on the left side, and their outputs on the right side. These boxes are connected in the diagram's rungs through its inputs and outputs. The LD language also provides other imperative elements (procedures, goto) that are not treated in the scope of this work.

It can be noted that LD programs are hard to visualize and become unreadable even for small to medium size programs. The validation of these programs through exhaustive testing are very expensive and unsure. A method for their formal verification is then of great help for system engineers.

2.3 Ladder Diagram Validation

In this work we intend to create a framework for automatic verification of temporal properties in LD programs. The properties to be verified over an LD program could be generic (and apply to every model) or specific to one model. Model related properties must be formulated by the programmer, while generic properties only rely on the metamodel concepts and can be automatically generated from the model.

One of the important generic properties to be verified on an LD program is the absence of race conditions [5]. A race condition is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time but, because of the nature of the device or system, the operations must be done in the proper sequence in order to be done correctly. A race condition occurs in an LD program when under fixed inputs and function block states, one or more outputs keep changing their values. In the LD example of Figure 2, the variable A is an input, C and D are memory, B and E are outputs. It can be seen that even if A is kept stable, the variables C , D and E will not stabilize thus it is an example of race condition. This kind of problem is sometimes difficult to detect with traditional techniques, and bugs not detected during the test period can be very costly to correct later. In section 3.4 we present how to

express this property as a LTL formulae and how to automatically generate and validate the formulae from the model.

3 MDE-Based Approach for Formal LD Program Validation

The approach we chose to verify properties in LD programs consists in translating the semantics of LD programs to a language formally (mathematically) defined. The chosen formal language is the Time Petri Net (TPN) [6].

To define and execute the translation we used a model driven engineering (MDE) approach. LD Programs are models that must conform to the LD metamodel. These models are then translated to a TPN model that conforms to a TPN metamodel. This TPN model is then translated to the input format of the *Tina* tool. General properties over the Ladder model are also automatically generated as LTL formulae. Finally, we use model-checking to verify the properties represented in LTL formulae over the generated TPN. A general schema of the process can be seen in Figure 3. All translations between models cited above were written in ATL (ATLAS Transformation Language) [7]. This language can be used to write translations of type model-to-model (M2M) and model-to-text (M2T). In this case study we have developed one M2M (Ladder2PetriNet) and two M2T (PetriNet2Tina and Properties) transformations.

3.1 Ladder Diagram Metamodel

The first contribution of this study is the definition of a metamodel for a subset of the LD language. An incremental approach has been chosen to build the metamodel. As a first step it was built a metamodel able to represent a significant subset of the LD language. This metamodel can be seen in Figure 4. An LD program (*Program*) is composed of variables (*Variable*) and rungs (*Rung*). A variable may represent an input, output or memory location (*InOutKind*). A rung is composed of elements (*Element*) that denote basic (*BasicElement*) and complex (*ComplexElement*) elements.

A basic element can be "normal_open" or "normal_closed" (*PlacementKind*). It denotes either a coil (*Coil*) or a contact (*Contact*) and always references a *Variable*. Complex elements represent functions and FBs. The attribute *kind* contains the name of the represented function or FB (*Functions*). Complex elements are composed of inputs

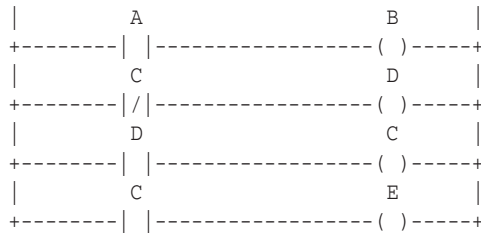


Fig. 2. Simple example of races in LD

(*Input*), outputs (*Output*) and internal variables (*InternalVariable*). The enumeration *Functions* can be easily extended to contain all the standard functions and FBs.

We introduce in this work the concept of path (*Path*). A rung is composed of paths. A path is a closed circuit by which the power can flow through the contacts to activate a coil. In the example of Figure 1 we have two paths in the first rung, one representing the boolean equation $C = A \wedge \neg B$ and the other representing the equation $C = C \wedge \neg B$. Note that this is a simple decomposition of the original equation. We have a single path in the second rung representing the equation $D = C$.

A path may have any number of contacts or outputs of complex elements as operators (*Operator*) but only one coil or input of complex elements as result (*Result*). The restriction of only one result per path was introduced to facilitate the translation later, but it does not introduce loose of generality because a path with n results can always be decomposed into n paths with one result. Note that *Result* is a generalization of the elements that may have their state changed by the power flow, while *Operator* is a generalization of the elements that are conditions for the power flow.

The data type of an internal variable of complex elements are restricted in this work to *Boolean* and *Integer* (*VariableType*) and the global variables of the program (*Variable*) are restricted to *Boolean*.

3.2 Time Petri Nets, SE-LTL and Tina Toolbox

In this study, we have chosen the technical space of time Petri nets as the target representation for formally expressing our LD semantics. We also have chosen to express our temporal properties as LTL formulae (*Linear Temporal Logic*) over the Petri net associated to a LD model. Then we manipulate Petri nets and LTL formulae within the *Tina* toolkit.

Time Petri Nets. Time Petri net (or TPN) [6] is one of the most widely used model for the specification and verification of real-time systems. TPNs are Petri nets in which a nonnegative real interval $I_s(t)$, with rational end-points, is associated with each transition t of the net [6].

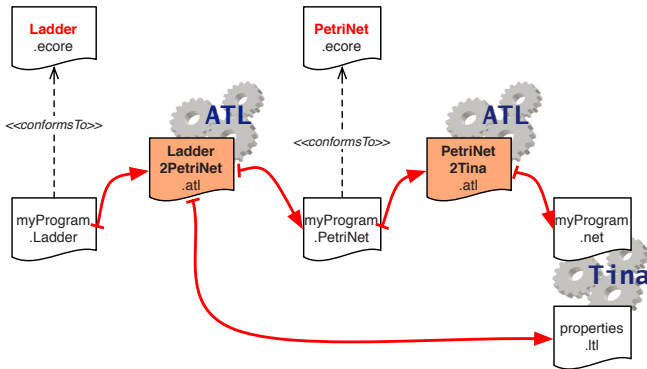


Fig. 3. General overview of the LD validation approach

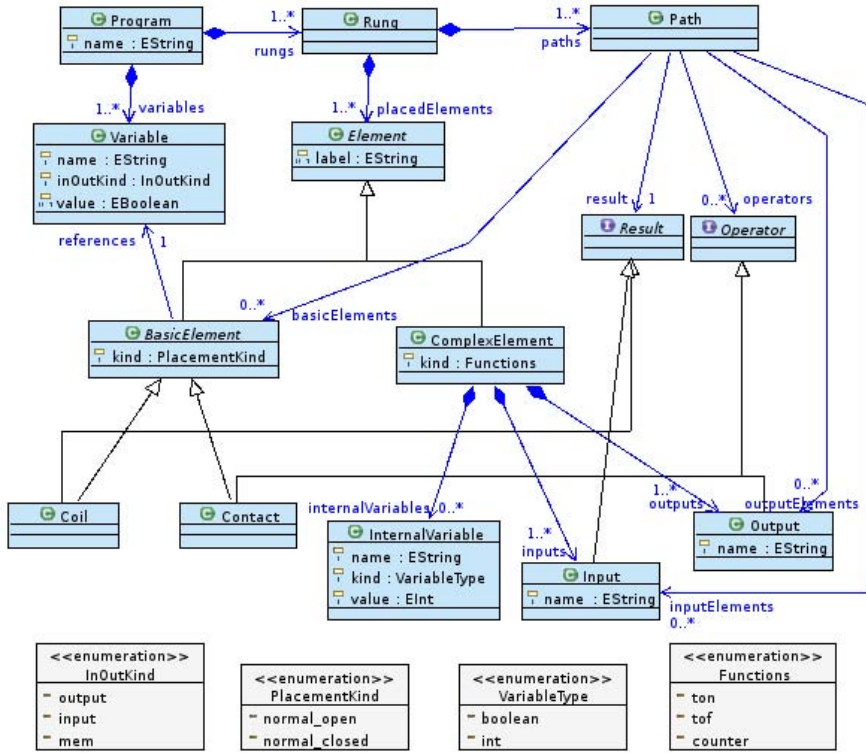


Fig. 4. A simplified LD Metamodel

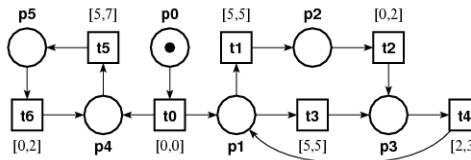


Fig. 5. A Time Petri net

Definition 1. A TPN is a tuple $\langle P, T, \text{Pre}, \text{Post}, m_0, I_s \rangle$, in which $\langle P, T, \text{Pre}, \text{Post}, m_0 \rangle$ is a Petri net, and $I_s : T \rightarrow \mathbf{I}^+$ is the Static Interval function.

P is the set of places, T is the set of transitions, $\text{Pre}, \text{Post} : T \rightarrow P \rightarrow \mathbf{N}^+$ are the precondition and postcondition functions, $m^0 : P \rightarrow \mathbf{N}^+$ is the initial marking. \mathbf{I}^+ is the set of nonempty real intervals with nonnegative rational end-points.

A Time Petri net is shown in Figure 5.

Let \mathbf{R}^+ be the set of nonnegative reals. For $i \in \mathbf{I}^+$, $\downarrow i$ denotes its left end-point, and $\uparrow i$ its right end-point (if i bounded) or ∞ . For any $\theta \in \mathbf{R}^+$, $i \dot{-} \theta$ denotes the interval $\{x - \theta | x \in i \wedge x \geq \theta\}$.

States and the temporal state transition relation $\xrightarrow{t @ \theta}$ are defined as follow:

Definition 2. A state of a TPN is a pair $s = (m, I)$ in which m is a marking and I is a function called the interval function. Function $I : T \rightarrow \mathbf{I}^+$ associates a temporal interval with every transition enabled at m .

We write $(m, I) \xrightarrow{t @ \theta} (m', I')$ if $\theta \in \mathbf{R}^+$ and:

1. $m \geq \mathbf{Pre}(t) \wedge \theta \geq \downarrow I(t) \wedge (\forall k \in T)(m \geq \mathbf{Pre}(k) \Rightarrow \theta \leq \uparrow I(k))$
2. $m' = m - \mathbf{Pre}(t) + \mathbf{Post}(t)$
3. $(\forall k \in T)(m' \geq \mathbf{Pre}(k) \Rightarrow I'(k) = \text{if } k \neq t \wedge m - \mathbf{Pre}(t) \geq \mathbf{Pre}(k) \text{ then } I(k) - \theta \text{ else } I_s(k))$

States evolve as follows: assume the current state is $s = (m, I)$, t is enabled at m , and became last enabled at time τ . Then t cannot fire before time $\tau + \text{Min}(I(t))$, and must fire no later than $\tau + \text{Max}(I(t))$, except if firing another transition before t made t not enabled anymore. Firing transitions takes no time.

TPN Metamodel. The time Petri nets metamodel is given in Figure 6. A Petri net (*Petri-Net*) is composed of nodes (*Node*) that denote places (*Place*) or transitions (*Transition*). Nodes are linked together by arcs (*Arc*). Arcs can be normal ones or read-arcs (*ArcKind*). The attribute *weight* specifies the number of tokens consumed in the source place or produced in the target one (in case of a read-arc, it is only used to check whether the source place contains at least the specified number of tokens). Petri nets marking is defined by the *tokens* attribute of *Place*. Finally, a time interval can be expressed on transitions.

Model-Checking. For this study, we use *State/Event – LTL* [8], a linear time temporal logic supporting both state and transition properties. The modeling framework consists of labeled Kripke structures (the state class graph in our case), which are directed graphs in which states are labeled with atomic propositions and transitions are labeled with actions.

Formulae Φ of *State/Event – LTL* are defined according to the following grammar:

$$\Phi ::= p \mid a \mid \neg\Phi \mid \Phi \vee \Phi \mid \bigcirc \Phi \mid \square \Phi \mid \diamond \Phi \mid \Phi U \Phi$$

Example of *State/Event – LTL* formulae :

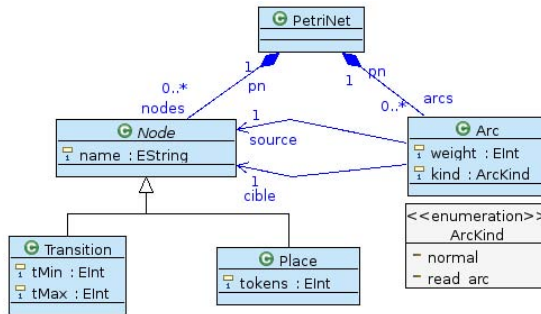


Fig. 6. Time Petri net Metamodel

	(For all paths)
P	P holds at the beginning of the path,
$\bigcirc P$	P holds at the next step,
$\square P$	P globally holds,
$\diamond P$	P holds in a future step,
$P U Q$	P holds until a step is reached where Q holds

Tina Toolbox for Time Petri Nets Verification

Tina (Time Petri Net Analyzer) is a software environment to edit and analyze Petri nets and time Petri nets [9]. The different tools constituting the environment can be used alone or together. Some of these tools will be used in this study:

- *nd* (*Net Draw*) : *nd* is an editing tool for automata and TPN, under a textual or graphical form. It integrates a “step by step” simulator (graphical or textual) for the TPN and allows to call other tools without leaving the editor.
- *tina* : this tool builds the state space of a Petri net, timed or not. It can perform classical constructs (marking graphs, covering trees) and also allows abstract state space construction, based on partial order techniques. It proposes, for TPN, all quotient graph constructions discussed in [10].
- *selt*: usually, it is necessary to check more specific properties than the ones dedicated to general accessibility alone, such as boundedness, deadlocks, pseudo liveness and liveness already checked by *tina*. The *selt* tool is a *model-checker* for an enriched version of *State/Event – LTL*. In case of non satisfiability, *selt* is able to build a readable counter-example sequence or in a more compressed form usable by the *Tina* simulator, in order to execute it step by step.

The *selt* logic is rich enough to encode marking invariants like $\square (p1 + p3 \geq 3)$.

Example of selt formulae :

$$\begin{aligned}
 & t1 \wedge p2 \geq 2 && \text{for every path } t1 \text{ and } m_0(p2) \geq 2, \\
 & \square (p2 + p4 + p5 = 2) && \text{a linear invariant marking,} \\
 & \square (p2 * p4 * p5 = 0) && \text{a non linear invariant marking,}
 \end{aligned}$$

selt also allows to define new operators :

$$\begin{aligned}
 \text{infix } q R p &= \square (p \Rightarrow \diamond q) && \text{definition of a “Responds to” operator, (denoted } R), \\
 t1 R t5 &&& t1 \text{ “Responds to” } t5.
 \end{aligned}$$

3.3 Translational Semantics of LD Programs through Time Petri Nets

In order to apply the model-checking techniques to verify an LD program using the available tools, it is necessary to represent the semantics of the program in a formal language. In this section we describe the translational semantics of an LD program in the form of a TPN.

The semantic transformation is based on the metamodel of the LD language (Figure 4) and the metamodel of the TPN (Figure 6). It was coded using ATL [7]. In this first work we do not translate the complex elements of the LD language, note that the

complex elements represent the functions and FB of the LD language, these elements have a unique semantics so a complete translation would have to take into account the specific semantics of each function or FB.

The ATL code is composed of eight matched rules. A matched rule enables to match some of the model elements of a source model, and to generate a number of distinct target model elements. Figure 7 shows what target elements are created for each rules. We use rule inheritance to filter different elements that are represented by a unique class in the metamodel. This inheritance could had been done at the metamodel level but our approach allows to keep the metamodel at a higher abstraction level and also allows to factorize the ATL code.

Each LD Program will be translated to a TPN. This TPN can be divided in three different groups of elements (Figure 8), each one with a different purpose during the simulation.

The first group is responsible for controlling the execution according to the PLC operation, a place is introduced for each execution step, that is, read the inputs, execute all the ordered rungs one by one, then update the outputs. The transitions are timed "[1, 1]" so as to guarantee that all the actions depending on one execution step (these actions are timed "[0, 0]") are executed before the state is changed. This mechanism is used to give a higher priority to transitions associated with the second and third group. We could have achieved the same effect by using prioritized Petri nets (PrTPNs) [11].

The second group represents the value of the variables. Each variable contained in the program will be translated into two places, one representing the False (*VariableName_0*) and the other the True (*VariableName_1*) state of the variable. Transitions are also created to set and reset these variables. Note that transitions of this group – describing the behavior of the environment – evolve in parallel and remain active between the capture of the environment (transition *input_reading*) and its update (transition *reset_variables*). This simplified modeling of the environment will result in a very strong combinatorial explosion.

The third group represents the execution semantics. This group is composed of two places for each variable as in the second group, but in this case the variables do not represent the real value of the variable, but are used to calculate the new values. This group represents the simulation variables. Input variables will have their simulation value copied from the real value - on the second group - during the input reading state. Output variables are calculated during the execution of the rungs and at the end, during the output update state, the real value - on the second group - will be updated according to the result of computation. At the end of each cycle all simulation variables are reset.

Figure 7 represents the elements generated for each of the eight rules. In some cases, elements generated by one rule are used by another rule. For example, the place *input_reading* is created by the rule *Program2PetriNet* and then is used by the rule *Input2PetriNet*.

The translation of paths generates the elements that calculate the value of the outputs according to the inputs. In Figure 7 (Path2PetriNet) one can see the elements generated by applying this rule to the first path of Figure 1. This path represents the boolean equation $C = A \wedge \neg B$.

The translation method generates a safe (1-bounded) TPN. This is guaranteed by the transformation. A structural analysis of the TPNs generated by the translation gives one

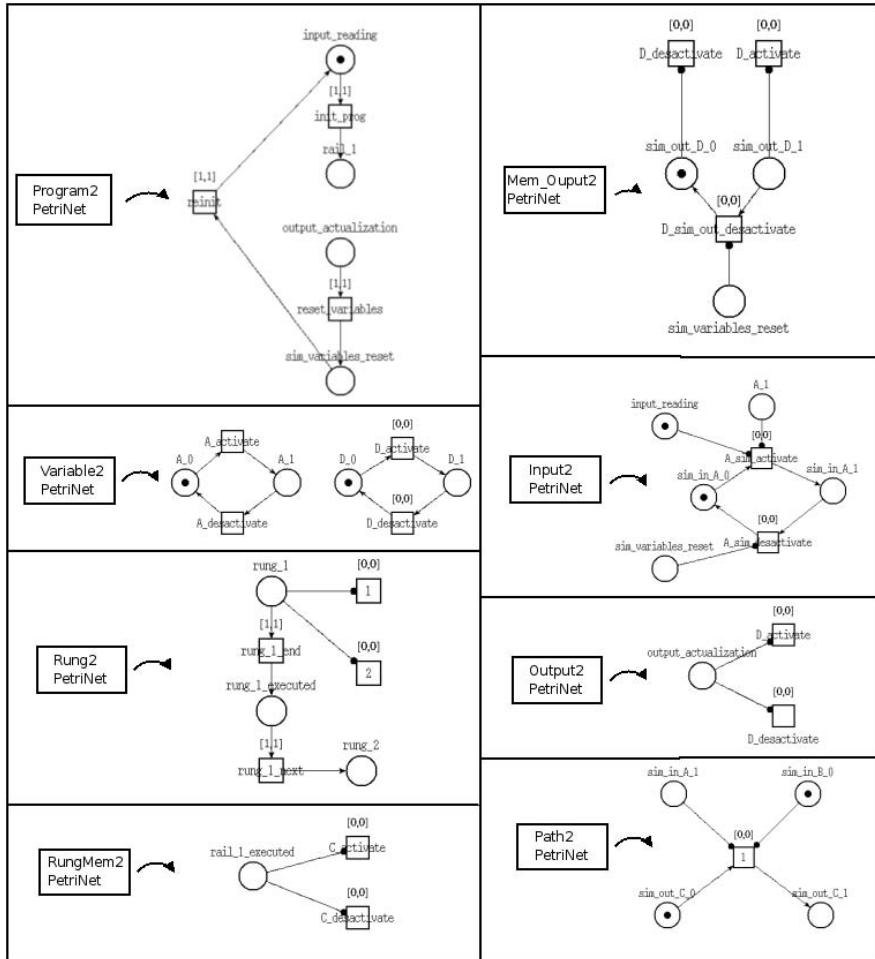


Fig. 7. Schema of the ATL transformation Ladder2PetriNet

invariant for all places in the first group, and one invariant for each couple of places representing the state True or False of variables.

3.4 Formal Verification of Race Conditions in LD Programs

In this section we demonstrate how to express the properties we wish to verify as LTL formulae, how to automatically generate these properties from the LD model and finally how to perform the model-checking with the *selt* tool.

Formalizing Race Conditions as LTL Formulae. To verify the absence of race conditions in an LD program we need to formally describe this property. We first describe the concept of stability of inputs and outputs. A boolean variable is said to be stable if it

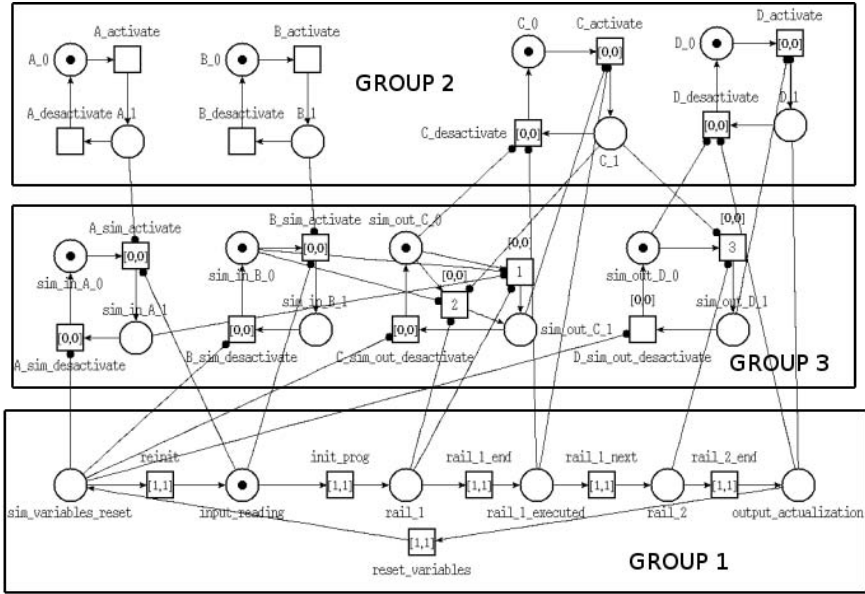


Fig. 8. TPN generated from the LD model on Figure 1

does not change its state, that is, it is always *True* or always *False*. When we translate an LD program, every boolean variable is represented by two complementary TPN places, $(VariableName)_0$ representing the *false* state and $(VariableName)_1$ representing the *true* state. Based on that we can formulate the Definition 3.

Definition 3. An LD variable called x is stable if $((\Box x_0) \vee (\Box x_1))$.

An LD program is free of race conditions if when the inputs are kept stable, all outputs and memory variables will stabilize. This property is represented in Definition 4.

Definition 4. An LD program is free of race condition if

$\Box (stable_inputs \Rightarrow \Diamond stable_outputs)$ where *stable_inputs* represents a logical AND between the stability condition for every input variable, while *stable_outputs* represents a logical AND between the stability condition for every output and memory variable.

To reduce the model-checking effort we can decompose the LTL formula into more simple ones. The decomposition is based on the following properties:

$$\begin{aligned} \Diamond((\Box P) \wedge (\Box Q)) &\equiv \Diamond \Box P \wedge \Diamond \Box Q \\ P \Rightarrow (Q \wedge R) &\equiv (P \Rightarrow Q) \wedge (P \Rightarrow R) \end{aligned}$$

Our general property can be decomposed in several different properties, one for each output or memory variable. Then we can check separately if every output will stabilize when the inputs are kept stable. When this property does not hold for a variable x , the LD program has a race on x .

The LTL formula can be simplified even more by applying the cone of influence (COI) reduction. The COI reduction is a technique that reduces the size of a model if

Table 1. Size of the TPN and state space generated

Program	TPN Generated	State Space
Figure 1	23 places, 24 transitions	462 places, 1432 transitions
Figure 2	31 places, 31 transitions	486 places, 1094 transitions

the propositional formula in the specification does not depend on all state variables in the structure [12]. The COI of a variable x is all the variables that influence the value of x . On the LD program represented in Figure 2 the COI of the variable B is $\{A\}$, while the COI of variable C is $\{C, D\}$. So we can redefine the Definition 4 as:

Definition 5. An LD program is free of race condition if for every output or memory variable V the following property holds: $\Box (COI(V)_stable \Rightarrow \Diamond V_stable)$, where $COI(V)_stable$ represents the stability condition for every variable in the COI of V .

Automatic Generation of LTL Properties. An ATL query was developed to automatically generate the LTL properties in a text format. This query is a M2T transformation that was built over the LD metamodel. It takes as input the LD model and generates the file *properties.ltl*. The COI of a variable is calculated through iterative searching in the model. We can use the generated file to verify temporal properties with the *selt* tool.

Model-checking. To perform the model-checking we first use the *tina* tool to build the state space of the TPN generated by the translation. The *selt* tool is then used to verify the LTL properties over this state space.

3.5 Results

This method was applied to the LD program in Figure 1 and as expected the property was evaluated to *TRUE*. The program is indeed free of race conditions. When applied to the program in Figure 2 the property was evaluated to *TRUE* for variable B , and to *FALSE* for variables C , D and E , proving that this program has a race on these three variables. In Table 1 is presented the size of the TPN and state space generated for both programs.

The results obtained for the two examples prove that the MDE approach can be used for verification of LD programs, transformation may be run on middle to complex LD diagrams. However, this first approach for translation of LD diagrams to Petri Nets can be largely optimised. The use of temporal constraints to simulate the priorities (instead of inhibitor arcs or directly of priorities²) results in an important extra cost. Likewise, the modeling of the environment is very simple (the environment continues to evolve in parallel between two captures) inducing a useless combinatorial explosion. With this translation, the Petri net associated with a LD program able to control a system containing 6 actuators (outputs) and 7 sensors (inputs) generates a state space of 7 million states. The “covering step graph” [13] construction of Tina, exploiting “partial order techniques” [14], allows to reduce drastically this explosion and leads to a state space of 40 states. However, a finer modeling of the environment will have to be carried out to take into account significant LD programs.

² Since March 2008, Tina 2.9.0 supports the priorities.

4 Related Works

Numerous authors have been working in formal verification of PLC programs. These works are normally based in two different approaches.

The first approach consists in implementing PLC programs directly in a formal language, and then automatically generate the PLC code. [15] introduces a program based on a set of rules to translate TIPN (*Time Interpreted Petri Nets*) control specifications into Ladder Diagrams. In [16], [17] and [18] a set of tools was presented for implementation of PLC programs using SIPN (*Signal Interpreted Petri Nets*), symbolic model-checking was used for validation and verification and the control specification was finally implemented through IL. [19] introduces a formalism based on timed state machines called PLC-automata and a translation of this formalism into ST code.

The second approach consists in translating the existing PLC programs into a formal language and automatically perform the verification. This approach permits the use of formal languages for the verification of PLC programs without changing the programming paradigm. Our work is based in this second approach. Some works using this approach have been developed. [20] and [21] provide a Petri Nets semantics for a subset of the IL language. [22] provides a framework for automatic verification of programs written in IL, a formal semantics (transition system) is presented for a subset of the IL language, which is directly coded into a model-checking tool (Cadence SMV³). It is then possible to automatically verify behavioral properties written in LTL. In [23] LD programs are formally represented through a transition system. In [5] a combination of probabilistic testing and program analysis has been used to detect race conditions in relay ladder programs. [24] generally discusses the transformations between NCES (*Net Condition Event Systems*) State Charts and PLC languages. In [25] is presented a method for formal verification of LD programs through a translation to timed automata. They check model related properties, while we focus on generic properties. The original program is abstracted according to these properties before the generation of the timed automata, in this way, for each property to be verified it may be generated a different timed automata. In [26] is discussed semantic issues and a verification approach for SFC and IL programs. It uses a model-checking framework to verify SFC programs and it suggests static analysis techniques, a combination of data flow analysis and abstract interpretation to verify IL programs.

Our work is based on this second approach and is the first to present an MDE approach for PLCs program validation.

5 Conclusion and Future Work

We have introduced in this article a model driven approach for formal verification of LD programs. We have defined a metamodel for a subset of the LD language. LD models are designed according to this metamodel. An LD model is then translated into a formal language (TPN) in the input format of *Tina* toolkit by means of two ATL transformations. The LTL properties to be verified are generated automatically from the model.

³ <http://www.kenmcml.com/smv.html>

Finally we utilize model-checking to verify the temporal properties over the generated TPN.

Some things still remain to be done, the metamodel must be completed to represent the complete LD language. The modelling of LD programs must be refined (specially the modeling of the environment) to reduce the combinatorial explosion.

In order to implement a complete translation of the LD concepts, we need a language directly supporting data processing like Fiacre [27]. Fiacre was designed in the framework of the TOPCASED project [28] dealing with model-driven engineering and gathering numerous partners, from both industry and academics. Therefore, Fiacre is designed both as the target language of model transformation engines from various models such as SDL, UML, AADL, and as the source language of compilers into the targeted verification toolboxes, namely CADP and *Tina*.

Model related properties must be formulated by the programmer, as LD programmers are not specialists in formal verification, a mechanism must be used to generate the LTL properties from the user specification. One of these mechanisms is presented in [29] where is presented an approach for generating LTL properties from TOCL ones, a temporal extension of OCL. In [30] is presented a framework for generating formal correctness properties from natural language requirements. The return of the model-checking counter-examples to the user must be implemented in order to hide to the user the complexities of the target language.

We are currently working on the recognition of the textual concrete syntax (ASCII art) specified in the IEC 61131-3 standard. We want to be able to generate LD models from its textual syntax. The metamodel presented in this article tries to represent the LD language in a format closer to its semantics. For the recognition of the textual syntax we should create an intermediate metamodel that is closer to this syntax and then apply a model transformation to obtain our original model. We are studying the possible solutions, one is the use of TCS [31], that relates metamodel concepts with the textual concrete syntax and allows to build a model from the textual syntax (injector) and also to generate concrete syntax from a model (extractor).

Our work demonstrates that it is possible to apply a model driven approach in formal verification of PLC programs. The combinatorial state explosion is a recurrent problem in the formal verification community and several works have been developed to overcome this problem.

References

1. Guasch, A., Quevedo, J., Milne, R.: Fault diagnosis for gas turbines based on the control system. *Engineering Applications of Artificial Intelligence* 13(4), 477–484 (2000)
2. International Electrotechnical Commission: IEC 61131-3 International Standard, Programmable Controllers, Part 3: Programming Languages (2003)
3. Tourlas, K.: An assessment of the IEC 1131 -3 standard on languages for programmable controllers. In: Daniel, P. (ed.) *SAFECOMP 1997: the 16th International Conference on Computer Safety, Reliability and Security* York, UK, September 7-10, 1997, pp. 210–219. Springer, Heidelberg (1997)
4. Schum, J.L.: *Locksmithing and Electronic Security Wiring Diagrams*. McGraw-Hill Professional, New York (2002)

5. Aiken, A., Fähndrich, M., Su, Z.: Detecting races in relay ladder logic programs. In: Steffen, B. (ed.) ETAPS 1998 and TACAS 1998. LNCS, vol. 1384, pp. 184–200. Springer, Heidelberg (1998)
6. Merlin, P., Farber, D.: Recoverability of communication protocols—implications of a theoretical study. *Communications, IEEE Transactions on* [legacy, pre - 1988] 24(9), 1036–1043 (1976)
7. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
8. Chaki, S., Clarke, E.M., Ouaknine, J., Sharygina, N., Sinha, N.: State/event-based software model checking. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 128–147. Springer, Heidelberg (2004)
9. Berthomieu, B., Ribet, P.O., Vernadat, F.: The tool TINA – construction of abstract state spaces for Petri nets and time Petri nets. *International Journal of Production Research* 42(14), 2741–2756 (2004)
10. Berthomieu, B., Vernadat, F.: Time petri nets analysis with tina. In: Third International Conference on Quantitative Evaluation of Systems, 2006. QEST 2006, pp. 123–124 (2006)
11. Berthomieu, B., Peres, F., Vernadat, F.: Model-checking bounded prioritized time petri nets. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 516–535. Springer, Heidelberg (2007)
12. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. *Formal Methods in System Design* 19(1), 7–34 (2001)
13. Vernadat, F., Azéma, P., Michel, F.: Covering step graph. In: Billington, J., Reisig, W. (eds.) ICATPN 1996. LNCS, vol. 1091, pp. 516–535. Springer, Heidelberg (1996)
14. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems. LNCS, vol. 1032. Springer, Heidelberg (1996)
15. Jimenez, I., Lopez, E., Ramirez, A.: Synthesis of ladder diagrams from petri nets controller models. In: Proceedings of the 2001 IEEE International Symposium on Intelligent Control, 2001 (ISIC 2001), pp. 225–230 (2001)
16. Minas, M., Frey, G.: Visual plc-programming using signal interpreted petri nets. In: American Control Conference, 2002. Proceedings of the 2002, vol. 6, pp. 5019–5024 (2002)
17. Klein, S., Frey, G., Litz, L.: A petri net based approach to the development of correct logic controllers. In: Proceedings of the 2nd International Workshop on Integration of Specification Techniques for Applications in Engineering (INT 2002), Grenoble (France), pp. 116–129 (2002)
18. Frey, G.: Design and formal Analysis of Petri Net based Logic Control Algorithms (Dissertation, University of Kaiserslautern). Shaker Verlag, Aachen (2002)
19. Dierks, H.: PLC-automata: a new class of implementable real-time automata. *Theoretical Computer Science* 253(1), 61–93 (2001)
20. Heiner, M., Menzel, T.: Instruction list verification using a petri net semantics (1998)
21. Heiner, M., Menzel, T.: A petri net semantics for the plc language instruction list. In: IEE workshop on discrete event systems (1998)
22. Canet, G., Couffin, S., Lesage, J.J., Petit, A., Schnobelen, P.: Towards the automatic verification of plc programs written in instruction list. In: 2000 IEEE International Conference on Systems, Man, and Cybernetics, vol. 4, pp. 2449–2454 (2000)
23. Moon, I.: Modeling programmable logic controllers for logic verification. *Control Systems Magazine, IEEE* 14(2), 53–59 (1994)
24. Rausch, M., Krogh, B.: Transformations between different model forms in discrete event systems. In: Computational Cybernetics and Simulation, 1997 IEEE International Conference on Systems, Man, and Cybernetics, 1997, October 12–15, 1997, vol. 3, pp. 2841–2846 (1997)

25. Bohumir Zoubek, J.M.R., Kwiatkowska, M.: Towards automatic verification of ladder logic programs. In: Proc. IMACS Multiconference on Computational Engineering in Systems Applications (CESA) (2003)
26. Huuck, R.: Software Verification for Programmable Logic Controllers. PhD thesis, Institute of Computer Science and Applied Mathematics, University of Kiel (2003)
27. Berthomieu, B., Farail, P., Gauffillet, P., Peres, F., Bodeveix, J.P., Filali, M., Saad, R., Vernadat, F., Garavel, H., Lang, F.: FIACRE: an intermediate language for model verification in the TOPCASED environment. In: European Congress on Embedded Real-Time Software (ERTS), Toulouse SEE (electronic medium) (2008), <http://www.see.asso.fr>
28. Vernadat, F., Percebois, C., Farail, P., Vingerhoeds, R., Rossignol, A., Talpin, J.P., Chemouil, D.: The TOPCASED Project - A Toolkit in OPen-source for Critical Applications and Sys-tEm Development. In: Data Systems In Aerospace (DASIA), Berlin, Germany, 22/05/2006-25/05/2006, European Space Agency (ESA Publications) (2006), <http://www.esa.int/publications> (electronic medium)
29. Combemale, B., Crégut, X., Garoche, P.L., Thirioux, X., Vernadat, F.: A Property-Driven Approach to Formal Verification of Process Models. In: Cardoso, J., Cordeiro, J., Filipe, J., Pedrosa, V. (eds.) Enterprise Information System IX. Springer, Heidelberg (2008)
30. Nikora, A.P.: Developing formal correctness properties from natural language requirements. NASA: Jet Propulsion Laboratory (2006)
31. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In: 5th international conference on Generative Programming and Component Engineering (GPCE 2006) (October 2006)

Array OL Descriptions of Repetitive Structures in VHDL

Stephen Wood¹, David Akehurst¹, Gareth Howells¹,
and Klaus McDonald-Maier²

¹ University of Kent, Canterbury, UK
S.K.Wood@dsl.pipex.com, dave@akehurst.net,
W.G.J.Howells@kent.ac.uk

² University of Essex, Colchester, UK
kdm@essex.ac.uk

Abstract. With the continuing advances in semiconductor technology driving the rise in complexity of electronic embedded systems, there is a demand for the development of high level modelling environments for computer-aided design. The modelling of highly repetitive structures in graphical form poses a particular challenge if a hierarchical approach is not adopted. This paper proposes a mechanism for describing such component structures in a compact form based upon extensions of the Array Oriented Language (Array-OL). An example is given as to how the structure described is subsequently mapped into VHDL code.

1 Introduction

To address the need for high level modelling environments, enabling designers of embedded systems to overcome the continuing rise in design complexity; the authors have previously developed a framework for deriving Very High Speed Integrated Circuits Hardware Description Language (VHDL) code from Unified Modelling Language (UML) state diagrams using Model Driven Development (MDD) techniques [1]. This paper focuses on extending that framework to incorporate the use of Array-OL (Array Oriented Language) [2] extensions as a mechanism for describing repetitive structural aspects of an electronic embedded system, including a description of how these descriptions may be mapped to VHDL.

The Array-OL specification language was developed by Thomson Marconi Sonar to fulfil the needs for specification, standardization and efficiency of multi-dimensional signal processing. The Authors in [3, 4, 5] have proposed a powerful Array-OL based mechanism for modelling complex topologies to specify connection links between model elements, defined as a UML profile which extends the UML 2.0 structural modelling mechanism. The proposed extensions enable a design-time deterministic specification of all the links of model elements that will exist at run time.

One of the proposed extensions for topology modelling is the abstract concept of *LinkTopology* [3], which comprises of an optional information set that can be associated to a relationship between potential instances. Two possible cases have been defined:

- *Reshape* (formerly known as *RepetitiveLinkTopology*), describing the links between different potential instances,
- *InterRepetitionLinkTopology* for links between potential instances connected to other potential instances of the same element.

A simplification of the former *RepetitiveLinkTopology* concept is also defined as the *RepetitiveStructuredClassifier*. It contains a single element with multi-dimensional multiplicity, which is connected to the ports of the *Repetitive-StructuredClassifier*, as depicted in [Figure](#).

The author of [6, 7] presents the formal semantics of the Array-OL language and discusses how to map and schedule an Array-OL application on a parallel and distributed architecture.

This paper illustrates some of the mechanisms, proposed in [3], for the description of regular connection patterns between model elements. It shows how these concepts can be mapped to VHDL; providing support for the structural description of a system. It further describes how it is decomposed into sub-systems and how these sub-systems are interconnected. The paper goes on to identify a limitation to the types of repetitive structures that this mechanism supports, proposing a solution in which an existing proposal for an Array-OL extension [3] could be further extended to define serial repetitions of elementary components, as a means of describing pipeline-like structures.

2 Array OL Description for Repetitive Component Structures

The Repetitive Component Structure describes the repetition of a single elementary component defined within a repetition space, as shown in Figure 1, and is modelled according to the repetitive concept of the Local Model in Array-OL. All components entities are characterised by an n^{th} dimensional array of ports, depicted as comma separated dimensions containing integer values corresponding to the number of array elements. Each array is initially indexed from its first element.

The elementary component **B** performs a simple task such as; Inversion, Logical operations, Multiplexing etc, on data supplied to its input ports, and presents the result at its output. For the purpose of this study, the nature of this behaviour is instantaneous and Asynchronous.

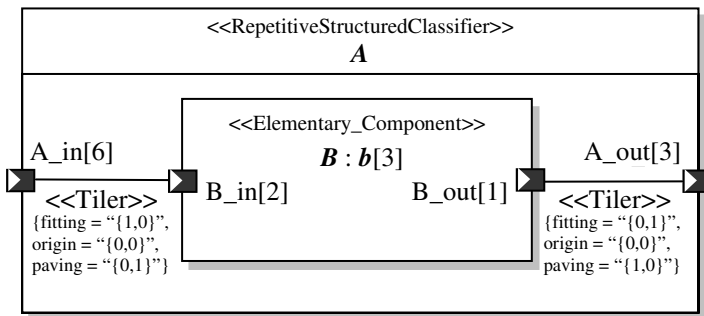


Fig. 1. Array OL description of a repetitive structure

Component **A** is subsequently defined as a repetitive structure formed from a quantity of elementary components. Each instance of the elementary component **B** must exhibit exactly the same behaviour, allowing the collective to be depicted as a single entity. The number of instances of the elementary component is declared by the lower-case **b** attribute; in this case it is 3. The total number of input ports of the repeated elementary components is derived from the product of input ports on a single elementary component and the number of instances of that component. Each port can be uniquely addressed by expressing these ports and component instances in the form of an array: *e.g.* for the example in figure; **[6]** represents a single dimensional array of the 6 input ports for **A**, these are connected to a 2-dimensional array representing the 2 ports of each of the 3 components **[2, 3]**. The same concept applies to the representation of output ports; however, for the purpose of this study, the following description refers to that of the input ports.

The configuration of the repetitive structure is defined by the interconnections between its ports and those of the elementary components. These interconnections are referred to as **Tilers**, depicted by a single line, and are characterised in part by the number of associated port connections made at each end. The Tilers require additional information to determine how the ports of each instance of an elementary component are connected to the ports of the repetitive structure or those of another elementary component if required.

As these ports are represented by arrays, the Tiler provides the means of associating an element of one array to the required element of another. To do so, the Tiler operates on sub-sets of the arrays known as **Patterns**. A pattern is the smallest representation of the elementary component as viewed by a Tiler, such that its repetition will generate the entire repetitive structure for that point. It is formed from the description of the ports for a single elementary component and is characterised by its size. The pattern for the description of the input ports of component **B** in *figure 1* would take the form of the vector (2, 0). This pattern is initially used to reference the first two ports of the repetitive structure **A**; starting from the first element of its array. This process is referred to as **Fitting**. The Tiler then provides a mapping of this pattern to the ports of the first instance of the elementary component, defined by the relative position from an index reference within the input array for **B**, in a process referred to as **Paving**. Tiling is an iterative process of Fitting and Paving, performed until all of the ports interconnections have been mapped.

Although the two arrays contain the same number of elements, they have very different ‘shapes’, the source being single dimensional **[6]** and the destination being 2-dimensional **[2, 3]**. The Tiler must therefore be instructed as to how the elements of one should be mapped to the elements of the other. A set of attributes are therefore defined as follows:

- **Origin** = (). The origin of the reference pattern (the first set of ports for the first component instance). This is usually expressed as (0) or (0, 0) in Cartesian form.
- **Fitting Matrix** = (). Describes how to fill the pattern with the source array elements

For each repetition of the Fitting process, a new reference in the source array is determined relative to that of the origin. The elements in the array related to the pattern are then identified from the Fitting Matrix.

For the example presented in *figure1*, the Fitting Matrix could be expressed as (1), signifying that adjacent elements in this single dimensional array should be accessed consecutively. A higher value would signify a defined distance between the elements used: e.g. A Fitting Matrix of (2) refers to alternate elements being used to fill the pattern.

Alternatively, the Cartesian representation of this Fitting Matrix would be (1, 0), depicting the reference of adjacent elements in the ‘X’ direction, and none in the ‘Y’ direction.

- **Fitting Limit** = (). Defines the size or shape of the pattern.

The Fitting Limit in this case refers to the number of ports represented by the pattern for a single elementary component, so Fitting Limit = (2). This is used in conjunction with the Fitting Matrix and the Origin to enumerate the pattern with the source array elements.

E.g. from the Origin at (0) of the input array, make a pattern of the size and shape (2, 0) from the Fitting Limit (2), and fill it from the elements whose relative positions are defined by the Fitting Matrix (1). This is depicted graphically in *figure2*.

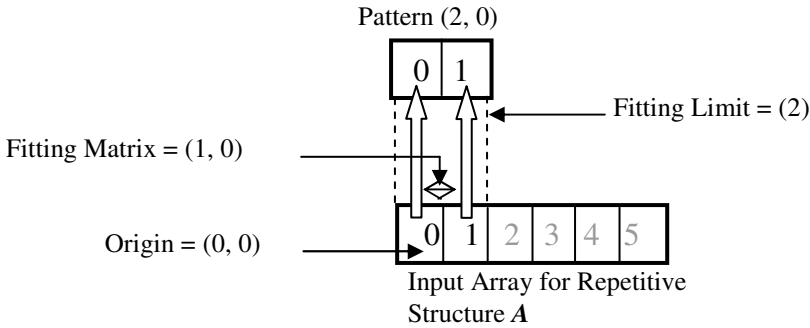


Fig. 2. First iteration of the Tiler’s ‘Fitting’ process– Filling a pattern of size = 2 from adjacent elements of the input array

Note: The values shown within the elements of both the array and the pattern signify the names of the elements by way of their relative position from the origin. This is not a representation of any data contained within.

- **Paving Matrix** = (). Describes how the pattern fills the destination array in each dimension.

The destination array in this case represents the repetition of the input ports, as seen by the Tiler, for each instance of the elementary component *B*. As the pattern size and shape was defined to match a single instance of the ports, the Paving Matrix provides the information necessary to map them to the correct instance of a component. In a similar scheme to that of fitting a pattern, Paving requires a destination reference point, from which the relative position for filling the array from the pattern can be determined. This can either be a reference to the Origin attribute as used by the Tiler for fitting the pattern, or defined as a separate attribute if the origin of the destination array

is not desired as the reference point. For each repetition of the Paving process, a new reference point must be determined relative to the first.

The Paving Matrix attribute for the considered example would take the form of (0, 1); signifying that for each repetition, the new reference point is to be on the same 'X' plane as the previous reference point position but one element away in the 'Y' direction.

- **Paving Limit** = (). Defines the number of patterns required to fill the array for all dimensions.

This provides the iteration limits for the iterative process of identifying the references points. For the considered example, the Paving Limit would be (0, 3).

Figure3 depicts the destination array and demonstrates how the first pattern is Paved into the array with respect to the initial reference point.

The second Tiler in this example performs exactly the same tasks as described here for the interconnection of the output ports.

Although different forms of repetitive structures are achievable using the Array OL description, the operation of the Tiler follows that outlined in this case study. A special case exists where ports are required to be mapped to different instances of the elementary component, an example of which is that of the 'Perfect Shuffle'. The Tiling operation required in fact implements two Tilers for an interconnection forming an operation referred to as '**Reshape**'. The description of the Perfect Shuffle and the operation of the Reshape interconnect is covered in a separate document.

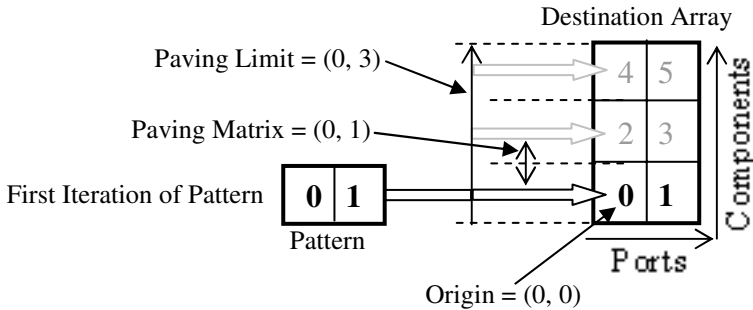


Fig. 3. Paving the destination array

3 Mathematical Description of the Tiling Process

This description is based upon the case study of the repetitive structure as depicted in *figure1*.

The first task is to determine a pattern capable of describing the elementary component's input ports for a single instance. This is then used to iteratively map all instances of the component to the ports of the repetitive structure.

The Pattern

A single component with 2 ports can be described as a pattern formed from a single dimensional array with 2 elements; represented as (2, 0). This pattern is a sub-set of

the source and destination arrays which describes the entire repetitive structure as viewed by the Tiler. The elements within the arrays use the zero-based indexing convention. Likewise, elements of the pattern itself use this convention; having the identity of **0** and **1** for the first and second elements respectively.

Fitting the Pattern to the Source Array

A formula to describe this process can be expressed as:

$$\forall \vec{x}_d, \vec{0} \leq \vec{x}_d \leq \vec{d}, (\vec{r} + F \times \vec{x}_d) \bmod \vec{m} \quad [Equation A]$$

For all elements in the pattern, in the range of the pattern size, (determine the sum and product) within the limit of the array

Where:

\vec{x}_d = an element in the reference pattern

$\vec{0}$ = the origin in the reference pattern

\vec{d} = the size and shape of the pattern

\vec{r} = the reference relative to the origin in the array

F = the Fitting Matrix

\vec{m} = the shape of the array

• For the First Iteration:

r = the origin (0, 0) and x_d is **0** and **1** (being in the limits of $0 \leq x_d < 2$) and $F = (1, 0)$

Hence: $\vec{r} + F \times \vec{x}_d$

$$\text{For } X_0 \quad \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot 0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \& \quad \text{For } X_1 \quad \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot 1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Having filled the pattern with the first two elements of the source array, (0, 0) and (0, 1), the next step is to '**pave**' the destination array for the first iteration.

Paving the Destination Array from the Pattern

The first task of the Paving process is to determine the relative reference points in the destination array. The formula that describes this process can be expressed as:

$$\forall \vec{x}_q, \vec{0} \leq \vec{x}_q \leq \vec{q}, (\vec{o} + P \times \vec{x}_q) \bmod \vec{m} \quad [Equation B]$$

Where:

\vec{x}_q = an element in the reference pattern

\vec{q} = the size and shape of the pattern

\vec{o} = the reference in the destination array

P = the Paving Matrix

For the first iteration: \vec{o} = the origin (0, 0). The Paving Matrix $P = (0, 1)$

The reference is determined from *equation B* as:

$$\vec{r} = \vec{o} + P \times \vec{x}_q$$

$$\vec{r} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \cdot 0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

The pattern can now be fitted into the destination array, relative to this reference, in the same way that the pattern was filled using the Fitting process.

Using: $\vec{r} + F \times \vec{x}_d$ from *equation A*

$$\text{For } X_0 \quad \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot 0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \quad \& \quad \text{For } X_1 \quad \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot 1 = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

An alternative expression for Paving can take the form of:

$$\forall \vec{x}_d, \vec{0} \leq \vec{x}_d \leq \vec{d}, (\vec{o} + P \times \vec{x}_q + F \times \vec{x}_d) \bmod \vec{m} \quad [\text{Equation C}]$$

where the two tasks of determining the reference point in the destination array and Fitting the contents of the Pattern into the array relative to that reference are contained within the same expression.

• The Second Iteration

Fitting the Pattern to the Source Array

$$\text{The next reference position becomes: } \vec{r} = \vec{r} + d = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 2 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \end{bmatrix}$$

Using: $\vec{r} + F \times \vec{x}_d$ from *equation A* to fit the new elements of the source array to the pattern:

$$\text{For } X_0 \quad \begin{bmatrix} 2 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot 0 = \begin{bmatrix} 2 \\ 0 \end{bmatrix} \quad \& \quad \text{For } X_1 \quad \begin{bmatrix} 2 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot 1 = \begin{bmatrix} 3 \\ 0 \end{bmatrix}$$

Paving the Destination Array from the Pattern

Using: $\vec{r} = \vec{o} + P \times \vec{x}_q$ from *equation B* to determine the new reference in the destination array:

$$\vec{r} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \cdot 1 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

And now: $\vec{r} + F \times \vec{x}_d$ from *equation A* to fit the pattern to the destination array elements:

$$\text{For } x_0 \quad \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot 0 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \quad \& \text{ For } x_1 \quad \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot 1 = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

• The Third and Final Iteration

Fitting the Pattern to the Source Array

$$\text{The next reference position becomes: } \vec{r} = \vec{r} + d = \begin{bmatrix} 2 \\ 0 \end{bmatrix} + \begin{bmatrix} 2 \\ 0 \end{bmatrix} = \begin{bmatrix} 4 \\ 0 \end{bmatrix}$$

Using: $\vec{r} + F \times \vec{x}_d$ from *equation A* to fit the new elements of the source array to the pattern:

$$\text{For } x_0 \quad \begin{bmatrix} 4 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot 0 = \begin{bmatrix} 4 \\ 0 \end{bmatrix} \quad \& \text{ For } x_1 \quad \begin{bmatrix} 4 \\ 0 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot 1 = \begin{bmatrix} 5 \\ 0 \end{bmatrix}$$

Paving the Destination Array from the Pattern

Using: $\vec{r} = \vec{o} + P \times \vec{x}_q$ from *equation B* to determine the new reference in the destination array:

$$\vec{r} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \cdot 1 = \begin{bmatrix} 0 \\ 2 \end{bmatrix}$$

And now: $\vec{r} + F \times \vec{x}_d$ from *equation A* to fit the pattern to the destination array elements:

$$\text{For } x_0 \quad \begin{bmatrix} 0 \\ 2 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot 0 = \begin{bmatrix} 0 \\ 2 \end{bmatrix} \quad \& \text{ For } x_1 \quad \begin{bmatrix} 0 \\ 2 \end{bmatrix} + \begin{bmatrix} 1 \\ 0 \end{bmatrix} \cdot 1 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

All elements of the source and destination arrays have now been mapped within the limits of the array sizes.

4 VHDL Implementation of the Repetitive Component Structure

It can be seen from this study that the operation performed by the Tiler can be considered as being equivalent to a ‘Port Map’ declaration used in a VHDL architecture description. This leads on to development of a VHDL implementation of the entire repetitive structure based upon its Array OL description.

To produce the required VHDL representation of this component structure, we first require a generic form of the component entities and their behaviour. This will provide a library of templates into which we can pass the specific details for the number of ports, elementary components and their interconnections.

5 VHDL Generic Template for the Repetitive Structure

Code Listing 1 shows the generic VHDL implementation of the repetitive structure case study referred to as **Y_Rep**, due to the nature of its repetition in the Y-Plane of

the repetition space. The architecture structure is completely generic, in that all values are referenced from the generic declaration in the entity.

The repetition is achieved by an iterative generation of port map declarations of the elementary *And_gate* component's ports to an array of interconnects.

The number of iterations is derived from the number of required instances of the elementary component. The behaviour of the *And_gate* component is defined separately in its own entity declaration. Its ports are defined here using the same names and types as those in its own entity.

The structure of this code and its derivation from the Array OL description will now be explained in stages, starting with the framework of a VHDL structure module.

```
entity Y_Rep is
generic (
    Array1Size: Positive:= 6;      -- * * Generic values for each of the Tilers * *
    Array2Size: Positive:= 3;      -- Total array size of source and destination for first Tiler
    Fit_Lim1: Positive:= 2;        -- Total array size of source and destination for second Tiler
    Fit_Lim2: Positive:= 1;        -- Fitting Limit of First Tiler
    Origin: Integer:= 0;          -- Fitting Limit of Second Tiler
    Fit_Matrix1: Positive:= 1;    -- Value of initial reference elements
    Fit_Matrix2: Positive:= 2;    -- Distance between array1 elements
    Pave_Lim: Positive:= 3;       -- Distance between array2 elements
    );                             -- Paving Limit for both Tilers

Port (
    Y_Rep_IN: in std_logic_vector (Origin to Array1Size -1); -- -1 due to the Zero-Based indexing
    Y_Rep_OUT: out std_logic_vector(Origin to Array2Size -1)
);

end Y_Rep;

architecture Structure of Y_Rep is
    component and_gate -- Port description of an elementary component defined in a separate entity
    Port (
        and_gate_IN: in std_logic_vector(Origin to Fit_Lim1 -1);
        and_gate_OUT: out std_logic_vector(Origin to Fit_Lim2 -1)
    );
    end component;

begin
    -- The Tiling Processes for each generated instance of the elementary component
    tilers: for i in Origin to Pave_Lim -1 generate
        new_and_gate: and_gate port map (and_gate_IN(Origin) => Y_Rep_IN(i*Fit_Lim1 + Fit_Matrix1*Origin),
                                         and_gate_IN(Origin+1) => Y_Rep_IN(i*Fit_Lim1 + Fit_Matrix1*Origin+1),
                                         and_gate_OUT(Origin) => Y_Rep_OUT(i*Fit_Lim2 + Fit_Matrix2*Origin)
                                         );
    end generate tilers;
end Structure;
```

Code Listing 1. Generic VHDL implementation of the repetitive structure case study

VHDL Module

Code Listing 2 shows the basic outline for a VHDL module of a structure. The line numbers depicted at the beginning of each line are included here for reference purposes, and are not representative of the actual line numbers of a completed VHDL module. Additional lines of code will exist outside of this structure, such as library references for the benefit of a VHDL code compiler. The bold type indicates reserved words in VHDL.

The VHDL module is essentially a description of a hardware component or *entity*. The first part of this description relates to its outward appearance, such that it may be

referred to by other components. At this point it is purely by a name declared in the first line, with no information as to what connections, or ports, are provided. Line 3 declares the end of the description of the entity. A description of the ports will be derived from the Array OL description of the component structure as depicted in *figure1*.

The entity description can also include declarations of *Generic* terms for parameter values.

The remaining portion of the code provides a description of the internal structure, or *architecture*, of the hardware entity, between lines 5 and 11. This again will be derived from the Array OL description of the component structure depicted in *figure1*.

The Entity Definition

• *Port definitions*

The generic form of port declarations is shown in *figure4*. For each named port, its direction with respect to data flow through the component entity is defined as *in* or *out*. The type of data to be handled by the port, such as: bit, std_logic, bit_vector, std_logic_vector, is defined with appropriated size or range.

```
port (  
  Port_name: direction type (range);  
  Port_name: direction type (range);  
);
```

Fig. 4. Generic form of Port declaration

Figure5 shows the Array OL description of the repetitive structure once again. This time however the components have been renamed to suit the application being described. For the purpose of this study, the use

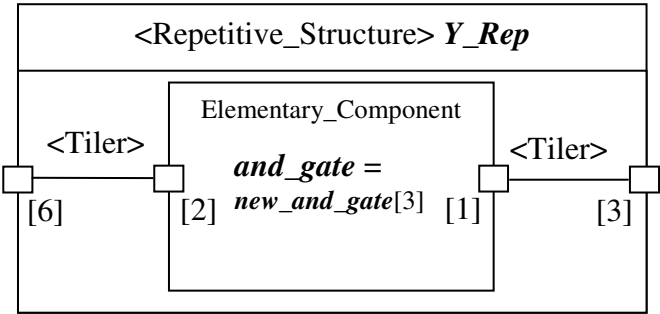


Fig. 5. Array OL description of a repetitive structure

of *Generic parameters* is overlooked, using instead the numeric values defined in the Array OL description.

The external ports of the entity are defined as vector quantities of the defined size with zero-based indexing. The names given to the ports use a naming convention whereby the entity name and direction of the port are used in combination.

The resulting ports are defined as: 6 inputs named *Y_Rep_IN* and 3 outputs named *Y_Rep_OUT*, and as such are represented in VHDL as:

```
entity Y_Rep is
Port (
    Y_Rep_IN: in std_logic_vector (0 to 6 -1); -- First
    Y_Rep_OUT: out std_logic_vector (0 to 3 -1) -- Second
);
end Y_Rep;
```

Code Listing 3. VHDL Entity declaration for the repetitive structure case study

The Architecture

The architecture describes the internal construction of the entity and its operation. For an elementary component such as an AND gate; this would comprise of a port assignment, whereby the logical AND of the input ports defined in the entity declaration, are assigned to the output port.

For example: `and_gate_OUT(0) <= and_gate_IN(0) and and_gate_IN(1)`

This would be sufficient to complete the description of a components architecture that performs an elementary task of this nature.

More complex architectures, such as that of the repetitive structure, define the use of other entities as components within its structure, together with the necessary signals required to provide the interconnections between each component.

- **Component**

A Component is declared as a reference to an existing entity of the same name by describing its ports in exactly the same way. For example; to use an elementary component such as the two input *and_gate* entity, a component called *and_gate* is defined as:

```
component and_gate
Port (
    and_gate_IN: in std_logic_vector (0 to 2 -1);
    and_gate_OUT: out std_logic_vector (0 to 1 -1)
);
end component;
```

Code Listing 4. VHDL Component declaration for the elementary component

where the input and output ports are of the same name, type and size as those declared in the *and_gate* entity. The size of the vectors representing the ports is derived from the patterns size for each tiler. The input ports are managed by the first tiler, having a pattern size, or fitting limit, of 2. The output ports are managed by the second tiler, having a fitting limit of 1.

• Architecture Structure

After the component declarations and any signal definitions have been made, the **begin** statement, shown on line 7 of *Code Listing 2*, signifies the start of the architecture structure definition. The definition describes how the repetitive structure is constructed by instantiating the elementary component at mapping its ports to the structure in an iterative process, until the maximum required components has been reached. This process represents that of the **Tiler** in the Array OL description, and therefore its values are taken directly from the description.

The following characteristics are a sub-set of those defined by the two Tilers, and as such are used in the VHDL implementation.

TILER 1(Input Ports)

Origin $o = 0$

Fitting Matrix $F = (0,1)$

Fitting Limit 1 $FL1 = (0,2)$

Paving Limit $PL = (0,3)$

TILER 2 (Output Ports)

Origin $o = 0$

Fitting Matrix $F = (0,1)$

Fitting Limit 2 $FL2 = (0,1)$

Paving Limit $PL = (0,3)$

```
tilers: for i in 0 to 3 -1 generate
new_and_gate: and_gate port map(
    and_gate_IN(0) => Y_Rep_IN(i*2+1*0),
    and_gate_IN(0+1) => Y_Rep_IN(i*2+1*0+1),
    and_gate_OUT(0) => Y_Rep_OUT(i*1+1*0)
);
end generate tilers;
```

Code Listing 5. Instantiation of the elementary components to form a repetitive structure

Each increment of the index i , within the limits of $Origin \leq i < Paving\ Limit$, instantiates the next repetition of the elementary component and provides a reference that is used to determine which of the structures ports are to be mapped to that instance of the component.

The input and output ports of the elementary component are mapped to the inputs of the repetitive structure using the expression: $\vec{r} + F \times \vec{x}_d$ from *equation A*.

In the VHDL equivalent of the Tiler, shown in *Code Listing 5*, the paving reference r is determined by multiplying the current index value i with that of the *Fitting Limit FL* ($FL1 = (2)$ and $FL2 = (1)$). The resultant value for r is then added to the product of the *Fitting Matrix F* (1 in both cases) and the pattern element values x_d ; where the first element of the input port pattern is (0) and the second and final element is (0+1), and the single element of the output port pattern is (0).

For example, the input ports of the elementary *AND gate* component are mapped as:

```
and_gate_IN(Origin) => Y_Rep_IN(i*Fit_Lim1 + Fit_Matrix1*Origin)
and
and_gate_IN(Origin +1) => Y_Rep_IN(i*Fit_Lim1 + Fit_Matrix1*Origin
+1)
```

and the output port is mapped as:

```
and_gate_OUT(Origin) => Y_Rep_OUT(i*Fit_Lim1 + Fit_Matrix1*Origin)
```

This concludes the description of the architecture structure and in turn the entire description of the VHDL implementation of the repetitive structure. Having demonstrated the relationship between the Array OL description and synthesisable VHDL code; a simplified version of the code shown in *Code Listing 1* can now be presented as *Code Listing 6*, where the use of generic values is dropped in favour of specific attribute values relative to this study.

```

entity Y_Rep is
Port(
    Y_Rep_IN: in std_logic_vector (0 to 6 -1); -- -1 due to the Zero-Based indexing
    Y_Rep_OUT: out std_logic_vector(0 to 3 -1)
);
end Y_Rep;

architecture Structure of Y_Rep is
    component and_gate -- Port description of an elementary component defined in a separate entity
    Port(
        and_gate_IN: in std_logic_vector(0 to 2 -1);
        and_gate_OUT: out std_logic_vector(0 to 1 -1)
    );
    end component;
begin
    -- The Tiling Processes for each generated instance of the elementary component
    tilers: for i in Origin to Pave_Lim -1 generate
        new_and_gate: and_gate port map (and_gate_IN(0) => Y_Rep_IN(i*2),
                                           and_gate_IN(1) => Y_Rep_IN(i*2+1),
                                           and_gate_OUT(0) => Y_Rep_OUT(i)
                                           );
    end generate tilers;
end Structure;

```

Code Listing 6. Specific VHDL implementation of the repetitive structure case study

The simplification made to the port references has been derived from the required behaviour and not the Array OL description. However, they are numerically the same and this can be considered as a mathematical simplification that may be carried out during the transformation between the Array OL description and the VHDL code generation.

The behaviour of the VHDL implementation presented in this study has been simulated, with results confirming the expected behaviour of the repetitive *AND Gate* structure.

6 Test Data and Results

The expected results derived from the required behaviour of the repetitive structure *Y_Rep* was mapped out in the table depicted in *figure 6*. This is a sub-set of the 64 possible input combinations, to demonstrate the operation of each instance of the elementary component. The timings used were arbitrary values.

In this example; the 3-bit output values (*Y_Rep_OUT*) represent the three outputs formed by the repetitive structure, with the right-hand (least significant bit) being the output of the first component instance, etc. The first two bits on the right-hand side of the 6-bit input values (*Y_Rep_IN*) form the respective component input pair.

TIME	Y_Rep_IN	Y_Rep_OUT
50 ns	000000	000
100 ns	000001	000
150 ns	000010	000
200 ns	000011	001
250 ns	000111	001
300 ns	001011	001
350 ns	001111	011
400 ns	011111	011
450 ns	101111	011
500 ns	111111	111

Fig. 6. Simulation test data

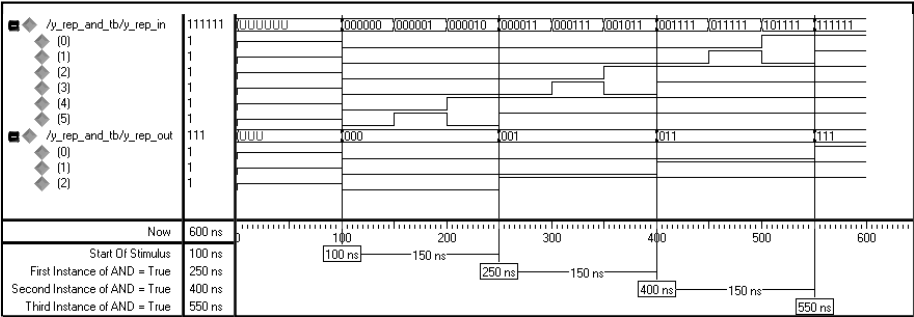


Fig. 7. Simulation test results

Each repeated instance of the *AND Gate* component should provide logic ‘1’ at its output (*and_gate_OUT*) when both of its input pairs (*and_gate_IN*) reach logic ‘1’ respectively.

A simulation was carried out using Mentor Graphics ModelSim® and the resultant simulation waveforms are presented here in *Figure7*.

Following the 100 ns initialisation period, the new input values are applied to the unit under test at 50 ns intervals. The first instance of the *AND Gate* component produces logic ‘1’ at its output at 250 ns when its input pair is asserted. Similarly for the second and third component instances produce logic ‘1’ at their outputs at 400 and 550 ns respectively. It can be seen that the results are the same as those predicted in *Figure6*.

7 Proposed Description for Serial Component Repetition

Having considered a method of describing potentially large parallel structures in a condensed form, and how that description may be mapped into VHDL architectures, it is proposed that a similar means of describing serialised structures such as serial buffers or pipe-lined stages should be determined.

At present, no such method of describing a serial repetition exists within the semantics of Array-OL. However, it is proposed here that instead of introducing a further extension to the language, the use of the *InterRepetitionLinkTopology* extension may be adapted as a possible solution to overcome this limitation. The *InterRepetitionLinkTopology* describes the links between potential instances connected to other potential instances of the same element, such as in a grid or cube topology, with the ability to specify the relative position of neighbouring potential instances of an element having multi-dimensional multiplicities.

Figure 8 shows the use of the *InterRepetitionLinkTopology* mechanism for modelling a two-dimensional 3x3 cyclic grid topology. The mechanism's *repetitionSpaceDependence* attribute is a translation vector on the space of the multi-dimensional array that identifies the position of a neighbour element. The *modulo* attribute (inherited from *LinkTopology*) indicates if the translation should be applied modulo the size of the multi-dimensional array. If it is not the case, the translation is not applied on the borders of the array, and the corresponding link will not be created.

It is envisaged that a serial repetitive structure can be modelled by declaring the elements $S_{array}[x, y]$ attribute such that repetitions are performed in a single dimension, such as $[3, 0]$ to provide a *west to east* repetition. Using the techniques previously presented by the authors in [1], the serial repetition can then be mapped to VHDL *generate* constructs to provide port mapping, similar to those presented here in section 4.

Further investigation of this mechanism is planned to verify the feasibility of this proposal for a means of modelling the serial repetition of components. This will primarily focus on how the existing concept of the *defaultLink* may be used to describe the input and output port connections of the repetitive structure to those of the system architecture.

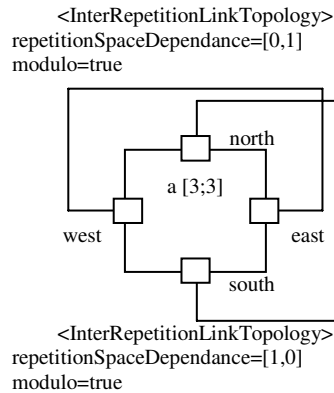


Fig. 8. A 3x3 cyclic grid topology model

8 Conclusion

This paper has presented a mechanism for describing repetitive structural aspects of an electronic embedded system as compact UML specifications to extend a MDD based framework for generating synthesizable VHDL code. Having identified semantic restrictions on the description of serial repetitions of components in this way, a proposed solution makes use of an existing proposal of an extension to the Array-OL language.

The intermediate process of transforming the Array OL description into the VHDL code is yet to be implemented. With the VHDL generic template providing the necessary replications of the design model architecture, the transformation tool needs

to extract the required attributes from the design characteristics and pass them into the generic declaration for the repetitive structure's entity.

Acknowledgment. This research is supported at the Department of Electronics at the University of Kent and at LIFL/INRIA Futurs at the University of Lille through the European Union ERDF Interreg IIIA initiative under the MODEASY grant.

References

1. Akehurst, D.H., Howells, W.G., McDonald-Maier, K.D., Bordbar, B.: Compiling UML State Diagrams into VHDL: An Experiment in Using Model Driven Development. In: Forum on Specification & Design Languages (FDL 2007), Barcelona, Spain (2007)
2. Demeure, A., Del Gallo, Y.: An array approach for signal processing design. In: Sophia-Antipolis conference on Micro-Electronics (SAME 1998), France (1998)
3. Cuccuru, A., Dekeyser, J.-L., Marquet, P., Boulet, P.: Towards UML 2 Extensions for Compact Modeling of Regular Complex Topologies. In: MoDELS/UML 2005, ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems, Montego Bay, Jamaica (2005)
4. Dumoulin, C., Dekeyser, J.-L.: UML Framework for Intensive Signal Processing Embedded Applications. Research Report 02-07. Laboratoire d'Informatique Fondamentale de Lille, Université de Lille I, France (2002)
5. Dumont, P., Boulet, P.: Another Multidimensional Synchronous Dataflow: Simulating Array-OL in Ptolemy II, Research Report RR-5516.LIFL, USTL (2005)
6. Boulet, P.: Array-OL revisited, multidimensional intensive signal processing specification. Technical Report 6113, INRIA (2007)
7. Boulet, P.: Formal Semantics of Array-OL, a Domain Specific Language for Intensive Multidimensional Signal Processing. SUBMITTED TO IEEE TRANSACTIONS ON COMPUTERS (2007)

Textual Modelling Embedded into Graphical Modelling

Markus Scheidgen

Department of Computer Science, Humboldt Universität zu Berlin
Unter den Linden 6, 10099 Berlin, Germany
`scheidge@informatik.hu-berlin.de`

Abstract. Although labelled graphical, many modelling languages represent important model parts as structured text. We benefit from sophisticated text editors when we use programming languages, but we neglect the same technology when we edit the textual parts of graphical models. Recent advances in generative engineering of textual model editors make the development of such sophisticated text editors practical, even for the smallest textual constructs of graphical languages. In this paper, we present techniques to embed textual model editors into graphical model editors and prove our approach for EMF-based textual editors and graphical editors created with GMF.

1 Introduction

In the past, the superiority of (purely) graphical representations was widely assumed at first and often challenged [1,2] later. Moher et al., for example, concluded in [3]: *"Not only is no single representation best for all kinds of programs, no single representation is [...] even best for all tasks involving the same program."* Today, graphical modelling languages and domain-specific languages (DSLs) often use a mixed form of representation: they use diagrams to represent structures visually, while other elements are represented textually. Examples for textual elements are signatures in UML class diagrams, mathematical expressions in many DSLs, OCL expressions used in other modelling languages, and many programming constructs of SDL [4].

Existing graphical editors address textual model parts poorly. The OCL editors of many UML tools, for example, barely provide syntactical checks and keyword highlighting. As a result, modellers produce errors in OCL constraints: errors that stay unnoticed until later processing, errors that when finally noticed are hard to relate to the OCL constraint parts that caused them. For other constructs, like operation signatures in UML class diagrams, editors often provide no textual editing capabilities at all. So editor users have to use big amounts of clicks to create signatures with graphical editing means. This process is slower and less intuitive than writing the signature down. As a general conclusion, editing the textual parts of models is less efficient than existing editor technology for programming languages would allow.

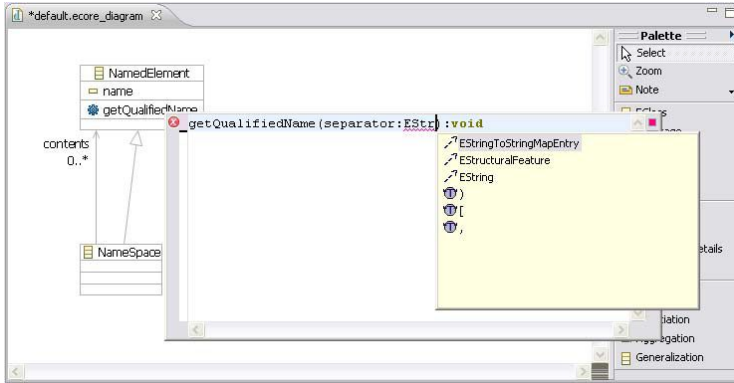


Fig. 1. The Ecore GMF editor with an embedded textual model editor

This programming technology includes modern programming environments with highly capable language-dependent text editors. These editors allow far more efficient programming than the plain text editors that were used before and that are still used for editing textual parts in graphical models. These modern program editors are complex and have to be built for each language independently. This renders the manual development of such editors too expensive for the textual parts of graphical languages, especially those of DSLs. However, recent research [5,6,7,8,9] utilizes generative engineering techniques to make text editor development for small applications practical. *Language engineers* only provide a high-level textual notation descriptions written in a corresponding meta-language, and meta-tools can automatically generate editors from those descriptions. The usual feature set of such generated editors comprises syntax highlighting, outline views, annotation of syntactical and semantic errors, occurrence markings, content-assist, and code formatting.

In this paper, we present techniques to combine textual modelling with graphical modelling, to embed textual model editors into graphical editors¹. Editor users open an *embedded text editor* by clicking on an element within the graphical *host editor*. The *embedded editor* is shown in a small overlay window, positioned at the selected element (see Fig. 1). Embedded editors have all the editing capabilities known from modern programming environments. To implement our approach, we use our textual editing framework TEF [11] and create embedded textual editors for graphical editors developed with the Eclipse Graphical Modelling Framework (GMF). TEF allows to create textual notations for arbitrary EMF meta-models. This includes meta-models with existing GMF editors.

As potential impact, our work can enhance the effectiveness of graphical modelling with languages that partially rely textual representations. Furthermore, it encourages the use of domain-specific modelling, because it allows the efficient

¹ The applicability of the presented techniques is not limited to graphical editors, but to editors based on the *Model View Controller (MVC)* pattern [10]. This also includes tree-based model editors, which are very popular in EMF-based projects.

development of DSL tools that combine the graphical and textual modelling paradigm. In general, this work could provide the tooling for a new breed of languages that combine the visualisation and editing advantages of both graphical notations and programming languages.

The remainder of this paper is structured as follows. The next section gives an introduction to textual modelling. After that, section 3 discusses the problems associated with embedding editors in general. In section 4, we realise our approach and implement it for several example languages. The paper is closed with related work and conclusions in sections 5 and 6.

2 Textual Modelling and Generative Engineering of Textual Model Editors

Textual modelling represents models with textual representations; modellers edit models by creating or changing a string of characters. We call the usage of languages with textual notations *textual modelling* and the process of using corresponding editors *textual model editing*. The necessary textual editing tools can be generatively engineered using *textual model editing frameworks*. These frameworks combine meta-languages for textual notations and corresponding meta-tools. Language engineers describe notations and meta-tools generate feature rich textual model editors automatically.

Before we explain how textual modelling can be embedded into graphical modelling, we use this section to introduce general concepts of textual modelling and the generative engineering of textual model editors. We accumulated this information from many existing but similar approaches [5,7,9,12]. Some technical details and the examples are specific to our own textual modelling framework TEF, which we also use to technically realise embedded textual modelling later in this paper.

2.1 Models and Their Representations, Corresponding Meta-models and Notations

We use the term *model* to refer to an abstract structure, where possible elements of this structure are predetermined by a *meta-model*. Thus, a model is always an instance of a meta-model. To read and modify a model, it needs to be represented, for example, graphically or textually. Possible *representations* are

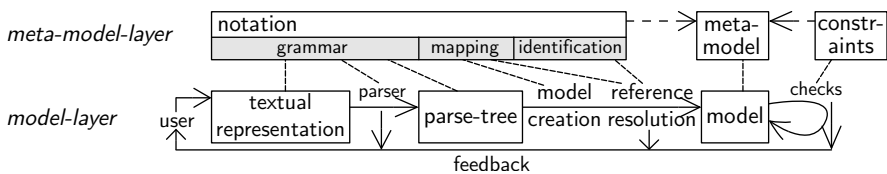


Fig. 2. The background parsing strategy and involved artefacts

defined by a *notation*. In the same way a model is an instance of a meta-model, a model representation is an instance of a notation.

A textual notation (used for the considered textual model editing technology) consists of a context-free grammar, and a mapping that relates this grammar to a meta-model. The grammar defines a set of syntactical correct representations. The mapping identifies the model corresponding to a representation. Fig. 3 shows an example notation. This notation for the Ecore language is mapped to the Ecore meta-model. A possible instance of this notation is shown in the top left of Fig. 4, the corresponding Ecore model is shown in its graphical representation on the top right of the same figure.

The notation in Fig. 3 is written for our TEF framework. It is a combination of a context-free grammar with EBNF elements, augmented with mappings to a meta-model. All string literals function as fixed terminals; all capitalized symbols are morphem classes for integers or identifiers; everything else are non-terminals. The bold printed elements relate grammar elements to meta-model classes and features. If such a meta-model relation is attached to the left-hand-side of a rule, the rule is related to a meta-model class; if attached to a right-hand-side symbol, the right-hand-side rule part is related to a feature. We distinguish between different kinds of meta-model relations: *element* relations relate to classes, *composite* relations relate to attributes or other compositional structural features, and *reference* relations relate to non-compositional structural features.²

2.2 The Background Parsing Strategy

Background parsing is a strategy to technically realise textual editing for context-free grammar based notations, used by existing textual model editors (see Fig. 2). Background parsing is a circular process, containing four steps. First, the user edits text as in a regular text editor. Second, the inserted text is parsed according to the notation's grammar. Third, a model is created from the resulting parse-tree based on a given grammar to meta-model mapping. Finally, language constraints are used to check the model. Errors in representation or resulting model can arise in all steps and are reported back to the user. Otherwise, the user is unaware of the parsing process, and continuous repetition gives the impression that the user edits the model directly. As opposed to other editing strategies, background parsing does not change the model, but creates a completely new model that replaces the current model, in each repetition.

² The terms *compositional* and *non-compositional*, hereby, refer to structural features that imply and not imply exclusive ownership. The corresponding structural features can carry other model elements as values. One model element used as a value in a compositional feature of another element becomes part of the other element. The part is called a *component* of the owning element; the owning element is called *container* respectively. Each model element can be the component of one container only, i.e. it can be a value in the structural features of one single model element at best. The component-container relationship is called *composition*. Cycles in composition are not allowed; composition forms trees within models. The roots of these trees are model elements without container; the leafs are elements without components. [13]

2.3 Creating Models from Parse-Trees

The result of parsing a textual representation is a syntax-tree or *parse-tree*. Each node in a parse-tree is an instance of the grammar rule that was used to produce the node. To create a model from a parse-tree, editors perform two depth-first traversals on the tree.

In the first run, editors use the element relations attached to a node's rule to create an instance of the corresponding meta-model class. This instance becomes the value represented by this node. It also serves as the context for traversing the children of this node. Morphems create corresponding primitive values, e.g. integers or strings. During the same traversal, we use the composite relations to add the values created by the respective child nodes to the referenced features in the actual context object. With this technique, the parse-tree implies composition between model elements. Furthermore, the compositional design of the meta-model must match the alignment of corresponding constructs in the textual notation.

```

syntax(Package) "models/Ecore.ecore" {
  Package:element(EPackage) ->
    "package" IDENTIFIER:composite(name)
    "{ " (PackageContents)* " }";

  PackageContents -> Package:composite(eSubpackages);
  PackageContents -> Class:composite(eClassifiers);
  PackageContents -> DataType:composite(eClassifiers);

  Class:element(EClass) ->
    "class" IDENTIFIER:composite(name) (SuperClasses)?
    "{ " (ClassContents)* " }";

  SuperClasses -> "extends" ClassRef:reference(eSuperTypes)
    ( " , " ClassRef:reference(eSuperTypes))*;
  ClassRef:element(EClass) -> IDENTIFIER:composite(name);

  ClassContents -> "attribute" Attribute:composite(eStructuralFeatures);
  ClassContents -> "reference" Reference:composite(eStructuralFeatures);
  ClassContents -> Operation:composite(eOperations);

  Attribute:element(EAttribute) -> IDENTIFIER:composite(name) ":"
    TypeRef:reference(eType) Multiplicity;

  TypeRef:element(EClassifier) -> IDENTIFIER:composite(name);

  Multiplicity -> ("/" INTEGER:composite(lowerBound) ":"
    UNLIMITED_INTEGER:composite(upperBound) "/" )?;

  Operation:element(EOperation) -> ...
}

```

Fig. 3. Excerpt from a notation description for the Ecore language

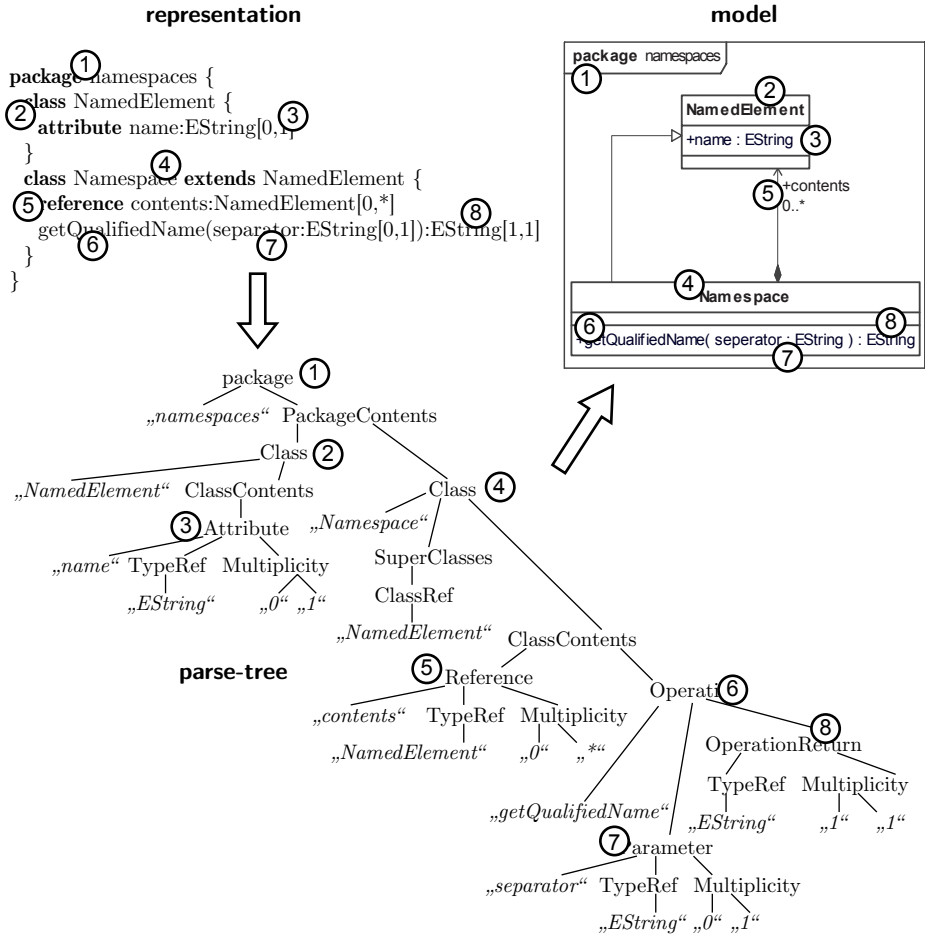


Fig. 4. A representation, parse-tree, and model based on the example Ecore notation

2.4 Identity and Reference Resolution

Before we can understand how editors add feature values for references in the second traversal, we have to understand identification of model elements. The *identity* of a model element is a value that uniquely identifies this element within a model. Each language defines a function that assigns each element of a language instance an identity. Example identities can be the model element itself, the name of the model element, or more complex constructs like full qualified names. An *identifier* is a value used to identify a model element based on the element's identity. In simple languages, identifiers can often be used directly to identify model elements: if an identifier and the identity of a model element are the same, this identifier identifies this model element. In many languages, however, identification is more complex, including name spaces, name hiding, imports, etc.

In those cases, an identifier depends on the context it is used in. The language must define a function that assigns a set of possible global identifiers to an identifier and its context. These global identifiers are then used to find a model element with an identity that matches one of those global identifiers.

Identity and identifiers are language-specific and might vary for model elements of different meta-classes. Textual model editing frameworks can only provide a simple default identification mechanism, i.e. based on a simple identity derived from a model element's meta-class and possible *name* attribute. TEF and other frameworks allow to customize this simple behaviour. Because no specific description mechanisms for identification could be found for existing textual editing frameworks yet, this part of a notation definition has usually to be programmed within the used textual editing framework.

In the second traversal (also called *reference resolution*), the editor goes through parse-tree and model simultaneously. Now, it uses all reference relations to add corresponding values to all non-compositional structural features. This time, it does not use the child nodes' values directly, but uses the child nodes' values as identifiers to resolve the corresponding referenced elements. Because all model elements were created in the first traversal, the referenced model elements must already exist in the model.

3 Embedded Textual Model Editing

Graphical model editors are based on the *Model View Controller* MVC pattern [10]. An MVC editor displays representations for model elements (model) through view objects (view). It offers actions, which allow the user to change model elements directly. Actions are realised in controller objects (controller). Examples for such actions are creating a new model element, modify the value set of a model element's feature, deleting a model element. The representing view objects react to these model changes and always show a representation of the current model. In MVC editors the user does not change the representation, only the model; the representation is just updated to the changed model. From now on, we assume that the host editor is an MVC editor.

We propose the embedded editing process illustrated in Fig. 5: The user selects a model element in the host editor and requests the embedded editor. The embedded editor is opened for the selected model element (1). We call this model element the *edited model element*. The edited model element includes the selected model element itself and all its components. The opened textual model editor creates an initial textual representation for the edited model element (2). The user can now change this representation, and background parsing creates new partial models, i.e. creates new edited model elements (3). The model in the host editor is not changed, until the user commits changes and closes the embedded textual model editor. At this point, the embedded editor replaces the original edited model element in the host editor's model, with the new edited model element created in the last background parsing iteration (4).

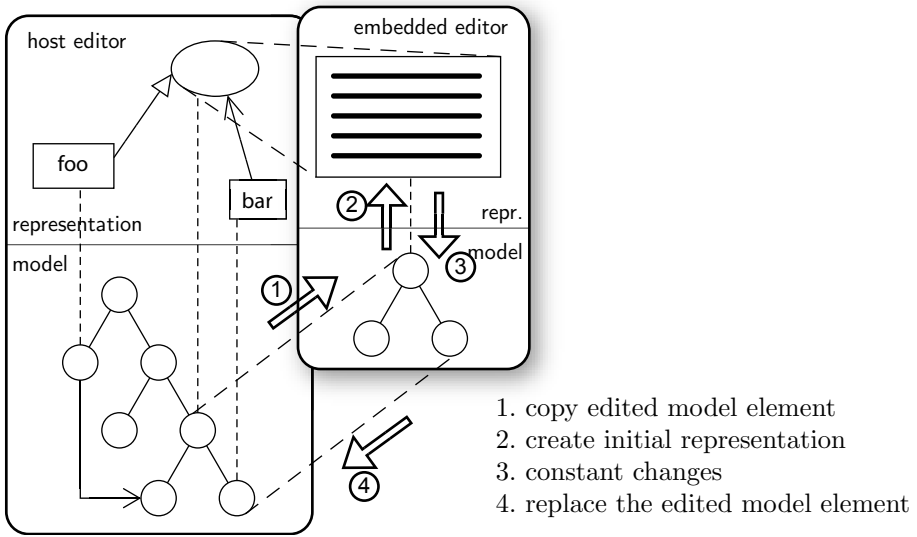


Fig. 5. Steps involved in the embedded textual editing process

There are three problems. First, we need textual model editors for partial models. Obviously, it is necessary to describe partial notations for corresponding partial representations. Second, when the editor is opened, it needs to create an initial textual representation for the selected model part. Finally, when the editor is closed, a new partial model has been created during background parsing. The newly created partial model needs to replace the original edited element. All references and other information associated with the original edited model element have to be preserved.

3.1 Creating Partial Notation Descriptions

Textual model editors rely on textual notations. Whether these notations cover a language's complete meta-model or just parts of it, is irrelevant, as long as the edited models only instantiate those meta-model parts that are covered by the textual notation.

Two solutions are possible: first, language engineers only provide a partial notation for the meta-model elements that they intend to provide embedded textual modelling for. In this case, the engineers have to be sure that they cover all related meta-model elements. Textual editing frameworks can automatically validate this. Second, language engineers provide a complete notation, and the editing framework automatically extracts partial notations for each meta-model element that embedded editing is activated for.

3.2 Initial Textual Representations

To create the initial textual representation, an editor has to reverse the model creating and parsing process: it creates a parse-tree from the edited model element and pretty prints this tree.

Creating a Parse-tree. To create a parse tree, the editor traverses the model along its composition. For each model element, the editor can determine a set of suitable grammar rules based on the element's meta-class, the values of its features, and the grammar to meta-model mapping. By using a back-tracking strategy, the editor can determine one or more possible parse-trees for a model. Notations that provide different possibilities to represent a single language construct in different ways, will lead to multiple possible parse-trees for the same model. Frameworks can give language engineers the possibility to prioritise grammar rules accordingly. If the editor cannot determine a parse-tree for a model, meta-model, grammar, and grammar to meta-model mapping are inconsistent. For example, a model element name might be optional as defined by the meta-model, but required by the notation: at some point in the model traversal, all possible grammar rules for the element would require the name of the element, but the name cannot be obtained.

Pretty Printing the Parse-tree. Pretty printing a parse-tree is basically straight forward. The only problem are the white-spaces between two tokens. A human readable textual representation needs reasonable white-spaces, i.e. *layout information*, between these tokens. This is a problem with two possible solutions. First, white-spaces originally created by editor users, can be stored within the model. Second, editors can create white-spaces automatically. Disadvantages for storing layout information are that the layout information has to be provided by editor users, and model and meta-model have to be extended with layout information elements. The advantage is that user layouts and all information that users express within layouts (also known as secondary notation [2]) are preserved. The second solution has complementary advantages and disadvantages.

For embedded editing, we propose the automatic generation of white-spaces. The edited text usually only comprises text pieces; white-spaces with hidden information, like empty lines, are not that important. Furthermore, the embedded text editors rely on the modelling facilities of the host editor; storing information beyond the model requires to change the host editor's implementation. And finally, with automatic layout, it is also possible to textually represent models that were not created via a textual representation. Models created with other means than a textual model editor (e.g. the host editor) can also be edited within such an editor.

Automatic layout of textual representations requires white-space clues as part of the textual notation definition. We propose the use of *white-space roles*. Language engineers add white-space roles as symbols to grammar rules. A white-space role determines the *role* that the separation between two tokens plays. White-spaces roles are dynamically instantiated with actual white-spaces, when text is created from a model. A component called *layout manager* defines possible white-space roles and is used to instantiate white-space roles. It creates white-spaces for white-space roles in the order they appear within the created textual model representation. The layout manager can instantiate the same white-space role differently, depending on the context the white-space role is used in.

An example: A layout manager for block-layouts as used in programming languages supports the roles *space*, *empty*, *statement*, *blockstart*, *blockend*, *indent*.

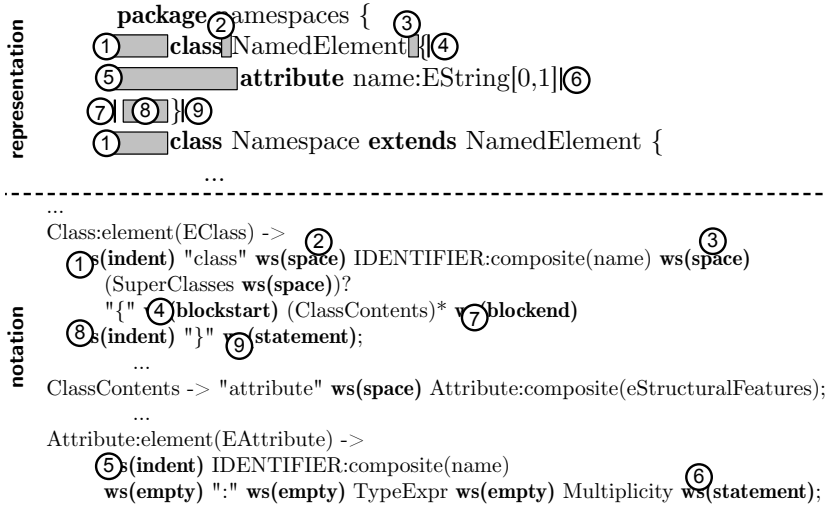


Fig. 6. An example text with white-spaces for an example notation with white-space roles

This manager instantiates each *space* with a space and each *empty* with an empty string. But, if the manager detects that a single line of code becomes too long, the layout manager can also instantiate both roles with a return and a followed proper indentation. The manager uses the *blockstart* and *blockend* roles to determine how to instantiate an *indent*. It increases the indentation size when a *blockstart* is instantiated, and decreases it, when a *blockend* is instantiated. Fig. 6 shows an example representation and corresponding notation with white-space roles for the Ecore language based on the block-layout manager.

3.3 Committing Model Changes

Problems caused by different editing paradigms. The host editor changes the model with small actions that only affect single or very few model elements. Opposite to the MVC host editor, the embedded textual model editor is based on background parsing and creates complete new model elements for each representation change. This causes two problems. First, other model elements that are not part of the edited model element might reference the edited model element or parts of it. These references break when the original edited model element is replaced by a new one. Second, the edited model element might contain information that is not represented; this information will be lost, because it is not part of the model element created through background parsing.

In today's modelling frameworks, e.g. EMF, we know all the references into the edited element and its parts, and we can simply reassign these references to the replacement model element and its parts. This would solve the first problem. We could also merge changes manifested in the newly created model element into the original model element. This would only update the original edited

model element and not replace it. This would solve both problems. Anyway, both solutions require identification: the editor has to access whether a model element in the original model element is meant to be the same as a corresponding model element part of the edited (newly created) model element. The editor can achieve this based on the elements' identity. This is obviously language-specific, and identification has to be defined for each language (see section 2).

With identification, the editor can tell whether two model elements have the same identity, and realising the first problem solution becomes very easy. The editor takes all references into the original edited model element, determines the identity of the referenced model elements within the original editor model element, searches for a model element with the same identity within the newly created model element, and reassigns the reference. The second problem solution requires some sort of algorithm that navigates both, the edited model element and the newly created model element, simultaneously along the model elements' composition. The merge algorithm has to compare the model elements feature by feature based on their identity, and transcribes all differences into the original edited model element. Deeper discussions about model merging is outside of this paper's scope; model merging algorithms and techniques are described in [14,15].

One problem remains: this problem occurs, if the user changes the text representation in a way that the identity of an element changes. Using the background parsing strategy, it is not clear what the user's intentions are. Did the user want to change, e.g., the name of a model element, or did he want to actually replace a model element, with a new element. The editor can only assume that the user wanted to create a new element. One way to solve this problems is to give the user the possibility to express his intention, e.g. provide a refactoring mechanism that allows to rename a model element.

Problems Caused by Different Undo/redo Paradigms. A convenient feature of model editors is the possibility to undo and redo model changes. This needs to be preserved for model changes in embedded text editors. In MVC editors, model changes are encapsulated in command objects, which allow to execute single model changes, and to reverse the execution of single model changes. Commands for executed model changes are stored in a command stack, which the editor uses for undo/redo. This is different in a textual model editor based on background parsing, where users change a string of characters. User actions are represented as replacements on that string. Undo/redo is based on a stack of string replacements. Embedded textual model editors and their graphical host editors obviously use incompatible representations for model changes.

We propose the following solution. Embedded text editors offer string replacement based undo/redo during textual editing. When the user closes the embedded editor and commits the textual changes, the necessary actions to replace the original edited model element are encapsulated into a single command, which is then stacked into the host editors undo/redo facility. This is a compromise: it allows to undo whole textual editing scenes, but does not allow to undo all the intermediate textual editing steps once the embedded editor is closed.

4 Realisation and Experiences

4.1 A Framework for Embedded Textual Model Editors

We created an EMF-based ([13]) textual editing framework for the Eclipse platform called Textual Editing Framework (TEF) [11]. This framework allows to describe textual model editors that use the background parsing strategy. Editors can be automatically derived from notation descriptions and support usual modern text editor features: syntax highlighting, error annotations, content assist [16], outline view, occurrences, and smart navigation. An example notation description is shown in Fig. 3.

We extended TEF for the development of embedded editors. Embedded editors can be created for EMF generated tree-based editors and editors created with the Graphical Modelling Framework (GMF) [17]. These embedded editors do not require to change the host editor. In theory, TEF should work for all EMF-based MVC host editors. To use TEF for embedded editors, language engineers provide a notation description for those meta-model elements that embedded textual editing is desired for. TEF automatically generates the embedded editors and provides so called object-contributions for corresponding EMF objects. These object-contributions manifest as context menu items in the host editor. With these menu items, users can open an embedded text editor for the selected model element. The embedded editor is a full fledged TEF editor providing all its features, except for the outline view, which is still showing the host editors outline. The embedded editor will only show the textual representation of the edited model element. The embedded text editor can be closed in two ways. One way indicates cancellation (by clicking somewhere into the host editor); the other way commits the changes made (pressing shift-enter).

We used the following problem solutions in TEF, which automatically creates partial notation descriptions by reducing a given notation for the specific edited model element dynamically, when the editor is opened. TEF editors create initial representations based on pretty printing with automatic white-space generation using layout managers. Embedded editors commit model changes by creating one single compound command that is added to the host editors command stack to preserve the host editors undo/redo capability. This command contains sub-commands that replace the original edited element and reassign all broken references based on either TEF's default identification or a language-specific identification mechanism. We plan to implement merging of newly created edited model element and original model element as future work.

4.2 Textual Editing of Ecore Models

We used TEF to develop a textual notation for Ecore and generated embedded textual editing for the graphical Ecore GMF editor and the standard tree-based Ecore editor. We wanted a more convenient editing of signatures for attributes, references, and operations. With the textual editing capabilities this becomes indeed more convenient and, e.g., renders the process of creating an operation with many parameters more efficient.

The work on the Ecore editors affirmed a few of the mentioned problems. First, many elements in the Ecore language carry information drawn from many side aspects of meta-modelling, such as parameters for code generation or XMI generation. Including all this information in the textual notation, would render it very cumbersome. Omitting this information in the textual notation, however, causes the loss of this information, when the corresponding model parts are edited textually. We hope to eliminate this problem by applying a model merging approach to update the original edited model element instead of replacing it. Second, a textual notation for Ecore needs complex identification constructs to realise textual references. Identification in Ecore has to rely on namespaces, imports, local and full qualified names. None of those constructs were defined in the language itself, and had to be invented on top of the actual Ecore language. We had to augment the automatic generated editing facilities, with manual implementations that describe such identification constructs.

4.3 An OCL Editor Integrated into Other Model Editors

The OCL constraint language is often used in conjunction with languages for object-oriented structures, or the behaviour of such structures. Hence, OCL expressions are often attached to the graphical notations of languages like UML, MOF, or Ecore. Therefore, editors for those languages should support OCL editing, but they usually only do by means of basic string based text editing.

We used TEF to develop an OCL editor based on the MDT OCL project. As an example, we integrated this editor with the tree-based Ecore editor: The EMF validation framework requires OCL constraints stored in Ecore annotations, because Ecore itself does not support the storage of OCL constraints. Because this only allows to store OCL constraints in their textual representation as strings, the embedded textual OCL editor is actually a normal text editor, which only uses background parsing to create internal OCL models to support advanced editor features, i.e. code-completion and error annotations. This makes committing the changes to OCL constraints particularly easy, because the embedded editor only has to replace a string annotation in the host-editors Ecore model. Another example application that we integrated the OCL editor into is the graphical editor for the UML activity-based action language in [18,19].

4.4 Editing for Mathematical Expressions in DSLs

Many of today's DSLs are developed based on EMF and instances of these languages are consequently edited using EMFs default generated tree-based model editors. This is fine for most parts of these languages, but can become tiresome, if models contain mathematical expressions. Because mathematics is commonly used to express computation, such expressions are part of many languages.

We used TEF to develop a simple straight forward notation for a simple straight forward expression meta-model. This expression meta-model and notations is a blueprint for integrating sophisticated editing capabilities for expressions into DSLs. We used this to realise expressions in a domain-specific language

for the definition of cellular automata, used to predict the spread of natural disasters like floods or fire.

5 Related Work

Work on textual notations based on meta-modelling with MOF includes Alanen et al. [20], Scheidgen et al. [21], Wimmer et al. [12]. This basic research was later utilised in frameworks for textual model editors. These are either based on existing meta-models (TCS [7], TCSSL [9], MontiCore [5]), or they generate meta-models or other parse-tree representations generated from the notations (xText [22], Safari [6]). Those frameworks however, only support the editing of text files. Models have to be created from those text files separately. Furthermore, pretty printing capabilities, if supported, are not directly integrated into the editors (because they are only text editors). This makes it hard to facilitate those frameworks for embedded textual editing, because these editors can not create an initial representation.

The GMF framework itself, provides some very simple means to describe structured text. It allows to create simple templates that assign different portions of a text to different object features. These simple templates allow less than regular complexity, and are therefore inadequate for many textual language constructs. First premature steps to describe the relations between graphical notations and textual notations have been made by Tveit et al. [23].

Background parsing, its notation meta-languages, and used algorithms are inspired by attributed grammars as described in [24]. Besides using context-free grammar and background parsing, textual modelling can also be conducted using the MVC pattern. MVC is used to realise Intentional Programming [25] and the Meta Programming System (MPS) [26]. Because the editing paradigm is the same than in graphical editors, it is thinkable that these frameworks can be integrated into graphical editors as well, maybe with more natural solutions to most of the presented problems. However, using MVC, only allows to create models based on simple actions. This way, you have to create, e.g., an expression as if you would create its parse-tree: from top to bottom. This is the same kind of limitation that is already imposed upon graphical editors, and this is exactly why we want to use textual modelling in the first place.

6 Conclusions and Future Work

The work presented in this paper showed that textual model editors can be embedded into graphical editors. It also showed that this can be done efficiently with generative development based on already existing technology. Furthermore, existing graphical editors do not necessarily have to be changed to embed textual editors; embedded editors can be provided as an add-on feature. Although, only formal studies can prove that embedded textual modelling raises productivity, we can assume increased productivity from the effect that modern text editors already had on programming.

There are a few issues that we suggest for further work. Problematic is that background parsing makes it hard to read the user's intention. A further problem are changes that alter a model element's identity and might result in unintended effects. In today's programming world, this is solved with refactoring. Textual editing frameworks need to allow the generative engineering of such facilities for textual editors (or model editors in general).

Furthermore, we need to explore the different grades of notation and editor integration. In this paper, we suggested to provide notation descriptions for both the graphical and the textual notation separately, which allows to use both editors (host and embedded) on arbitrary model elements. But, in many cases it would be more natural to integrate both notation description into a single notation description. This requires better integrated description languages and editing frameworks, but would allow for more concise, coherent notation descriptions and a more seamless overall modelling experience. On the technical site, we should explore other possibilities to integrate editors into each other. For example, it would be desirable to have the textual editor widget directly contained in the corresponding graphical widget. This makes additional overlay windows superfluous and would create a more fluid editing experience. Furthermore, this allows to see syntactical highlighting and error annotations directly in the graphical model view.

References

1. Green, T.R.G., Petre, M.: When Visual Programs are Harder to Read than Textual Programs. In: Human-Computer Interaction: Tasks and Organisation, Proceedings ECCE-6 (6th European Conference Cognitive Ergonomics) (1992)
2. Petre, M.: Why looking isn't always seeing: readership skills and graphical programming. *Commun. ACM* 38(6) (1995)
3. Moher, T.G., Mak, D.C., Blumenthal, B., Leventhal, L.M.: Comparing the Comprehensibility of Textual and Graphical Programs: The Case of Petri Nets. In: Empirical Studies of Programmers - Fifth Workshop (1993)
4. ITU-T: ITU-T Recommendation Z.100: Specification and Description Language (SDL). International Telecommunication Union (2002)
5. Krahn, H., Rumpe, B., Völkel, S.: Integrated Definition of Abstract and Concrete Syntax for Textual Languages. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) *MODELS 2007*. LNCS, vol. 4735. Springer, Heidelberg (2007)
6. Charles, P., Dolby, J., Fuhrer, R.M., Sutton, S.M., Sutton, J.S.M., Vaziri, M.: SAFARI: a meta-tooling framework for generating language-specific IDE's. In: *OOPSLA 2006: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications* (2006)
7. Jouault, F., Bézivin, J., Kurtev, I.: A DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In: *GPCE 2006: Proceedings of the 5th International Conference on Generative Programming and Component Engineering* (2006)
8. Kleppe, A.: Towards the Generation of a Text-Based IDE from a Language Meta-model. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) *ECMDA-FA*. LNCS, vol. 4530. Springer, Heidelberg (2007)

9. Muller, P.A., Fleurey, F., Fondement, F., Hassenforder, M., Schneckenburger, R., Gérard, S., Jézéquel, J.M.: Model-Driven Analysis and Synthesis of Concrete Syntax. In: 9th Intern. Conf. on Model Driven Engineering Languages and Systems (2006)
10. Holub, A.: Building user interfaces for object-oriented systems. JavaWorld (1999)
11. Homepage: Textual Editing Framework, <http://tef.berlios.de>
12. Wimmer, M., Kramler, G.: Bridging Grammarware and Modelware. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713. Springer, Heidelberg (2005)
13. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework (The Eclipse Series). Addison-Wesley Professional, Reading (2003)
14. Zito, A., Diskin, Z., Dingel, J.: Package Merge in UML 2: Practice vs. Theory? In: 9th Intern. Conf. on Model Driven Engineering Languages and Systems (2006)
15. Kolovos, D.S., Paige, R.F., Polack, F.: Merging Models with the Epsilon Merging Language (EML). In: 9th Intern. Conf. on Model Driven Engineering Languages and Systems (2006)
16. Scheidgen, M.: Integrating Content-Assist into Textual Model Editors. In: Modellierung 2008. LNI (2008)
17. Homepage: Graphical Modelling Framework, <http://www.eclipse.org/gmf/>
18. Scheidgen, M., Fischer, J.: Human Comprehensible and Machine Processable Specifications of Operational Semantics. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA. LNCS, vol. 4530. Springer, Heidelberg (2007)
19. Eichler, H., Soden, M.: An Approach to use Executable Models for Testing. In: Enterprise Modelling and Information Systems Architecture. LNI (2007)
20. Alanen, M., Porres, I.: A Relation between Context-Free Grammars and Meta Object Facility Metamodels. Technical report, TUCS (2004)
21. Fischer, J., Piefel, M., Scheidgen, M.: A Metamodel for SDL-2000 in the Context of Metamodelling ULF. In: System Analysis and Modeling, 4th International SDL and MSC Workshop, SAM (2004)
22. Homepage: openArchitectureWare, <http://www.openarchitectureware.org>
23. Prinz, A., Scheidgen, M., Tveit, M.S.: A Model-Based Standard for SDL. In: Gaudin, E., Najm, E., Reed, R. (eds.) SDL 2007. LNCS, vol. 4745. Springer, Heidelberg (2007)
24. Knuth, D.E.: Semantics of Context-Free Languages. Theory of Computing Systems 2 (1968)
25. Simonyi, C.: The death of computer languages, the birth of Intentional Programming. Technical report, Microsoft Research (1995)
26. Dmitriev, S.: Language Oriented Programming: The Next Programming Paradigm. onBoard (November 2004)

Classification of Concrete Textual Syntax Mapping Approaches

Thomas Goldschmidt¹, Steffen Becker¹, and Axel Uhl²

¹ FZI Research Center for Information Technologies
Karlsruhe, Germany

{goldschmidt, sbecker}@fzi.de

² SAP AG

Walldorf, Germany

axel.uhl@sap.com

Abstract. Textual concrete syntaxes for models are beneficial for many reasons. They foster usability and productivity because of their fast editing style, their usage of error markers, autocompletion and quick fixes. Furthermore, they can easily be integrated into existing tools such as diff/merge or information interchange through e-mail, wikis or blogs. Several frameworks and tools from different communities for creating concrete textual syntaxes for models emerged during recent years. However, these approaches failed to provide a solution in general. Open issues are incremental parsing and model updating as well as partial and federated views. To determine the capabilities of existing approaches, we provide a classification schema, apply it to these approaches, and identify their deficiencies.

1 Introduction

With the advent of model-driven development techniques, graphical modelling languages became more and more popular. However, there are also use cases where a concrete textual syntax (CTS) is more appropriate to edit models. For example, this applies to mathematical, expression-like languages such as query, or constraint languages (e.g. Object Constraint Language (OCL)[1]). Another example where textual syntaxes are preferred over graphical ones are model transformation languages such as QVT. Even in graphical modelling there are parts that can only be expressed and displayed textually in a convenient way, e.g., an operation signature in UML.

Advantages of such textual syntaxes are their clear structure (reading from left to right, from top to bottom, indentation as substructures) and their focus on straight ahead typing. Tools that can handle textual artefacts are widely spread and very mature. Especially software developers are used to having their development artefacts being developed as text. Helpers such as code highlighting, autocompletion, and error annotations elevate the capabilities of textual editors significantly. Diff/merge operations, the construction of patches, etc. are already well understood for text in contrast to graphically-noted models. Furthermore, submitting a part of text representing the model to a discussion forum or writing a mail containing snippets written in the concrete syntax is easy in a textual syntax because, due to its platform and tool independency, everyone can view and edit this text. Further advantages of a textual versus a graphical concrete syntax for users as well as for tool developers are mentioned in [2].

Basically, a CTS approach has to map constructs of a metamodel to the definition of a textual syntax, i.e., a grammar. Tools that translate between the textual representation as well as the abstract representation should be (automatically) derived from the mapping definition. Additionally, an editor could be provided that facilitates features such as syntax highlighting, autocompletion or error markers specific for the particular syntax.

However, for a CTS approach to prove applicable in an enterprise and in a large scale environment a lot of requirements have to be met [3,4]. Important requirements are for instance the incremental updating of existing models and the support for UUID-based repositories or the definition of partial and/or combined views.

A great variety of approaches and tools that provide concrete textual syntax mappings for models emerged recently or have been enhanced to support it. Originating from different communities, from academia as well as industry, their set of features is also very diverse. Some approaches facilitate the translation from text to model by parsing text from time to time in the background while others use a model-view-controller (MVC) pattern to keep the model in sync with the text. Being able to store format information in addition to the actual model, some approaches preserve the original format of the text over subsequent translation runs. However, there are still requirements that are not or insufficiently fulfilled by existing approaches.

The contributions of this paper are (1) a classification schema for CTS approaches, (2) the application of this schema to existing CTS approaches as well as the identification of their deficiencies and (3) the discussion of several important features that are not yet addressed. The presented classification schema is used to describe the necessary as well as extended features of a CTS mapping framework. Ten different approaches were examined for their support of these features. The discussion of the yet unsupported features focuses on the applicability of a CTS approach in an enterprise with a multitude of modelling languages, metamodels and tools and distributed, parallel development.

This paper is structured as follows. An overview on the foundations of a CTS approach is given in Sect. 2. Section 3 presents the classification schema that includes the features of a CTS approach. The actual classification is presented in Sect. 4. Section 5 discusses the findings and requirements for modelling in the enterprise. Related work concerning the classification of CTS approaches is treated in Sect. 6. Conclusions are drawn in Sect. 7.

2 Foundations of a Concrete Textual Syntax Mapping

Several basic components are needed to provide a comprehensive tooling for a CTS approach. To be able to relate constructs from a metamodel to elements of a CTS, a mapping between the metamodel and the definition of this syntax is needed. The definition of a textual syntax is provided by a grammar. To translate the textual syntax to its model representation, a *lexer*, a *parser* as well as a component that is responsible for the *semantical analysis* (type checking, resolving of references, etc.) are needed. Even for an approach that directly edits the model without having an explicit *parser* component for the grammar, a similar component is needed that decides how the text is translated into model elements. For reasons of convenience we will call all kinds and combinations of components that implement the translation from text to model a

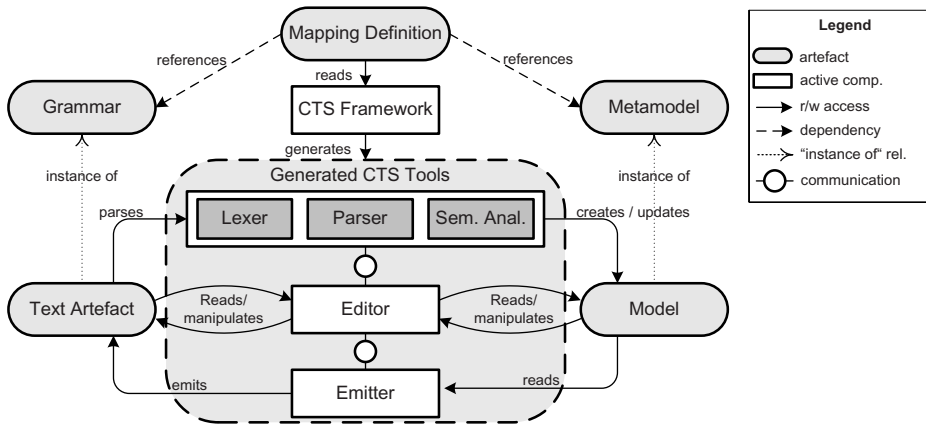


Fig. 1. General structure of a CTS framework

parser. The backward transformation, from model to text, is provided by an *emitter*. Both components can be generated using the above-mentioned mapping definition.

An overview of these components is depicted in Fig. 1. This figure shows that the CTS framework uses the information that is provided in the mapping definition to generate the parser, emitter and editor components. For example, the mapping could define that a UML class *c* is represented in the concrete syntax using the following template: `class <c.name> { <call to templates for contents of c> }`. The framework could then generate a parser that recognises this structure and instantiates a UML class when parsing this pattern and setting the name property accordingly. Furthermore, an emitter can use this template to translate an existing UML class into its textual representation.

The grammar \leftrightarrow metamodel mapping can also be used to generate an editor for the language represented by the metamodel. This editor can then use the generated parser and emitter to modify the text and the model. This editor is then also responsible for keeping the text and the model in sync, e.g., by calling the parser everytime the text has changed. Based on the mapping definition several features of the editor can be generated, such as syntax highlighting, autocompletion or error reporting. Refactoring actions can also be provided with this editor. Having the model as well as the text in its direct access such an editor could, e.g, provide a rename action which updates the name property of an element on the model and then uses the emitter to update all occurrences of this name in the text.

3 Classification Schema

To be able to compare and classify different approaches that exist for the creation of a concrete textual syntax for a metamodel a systematic list of possible features of such an approach is needed. We chose to use feature diagrams [5] to provide an overview on the available features. Figure 2 depicts a feature diagram of the features considered

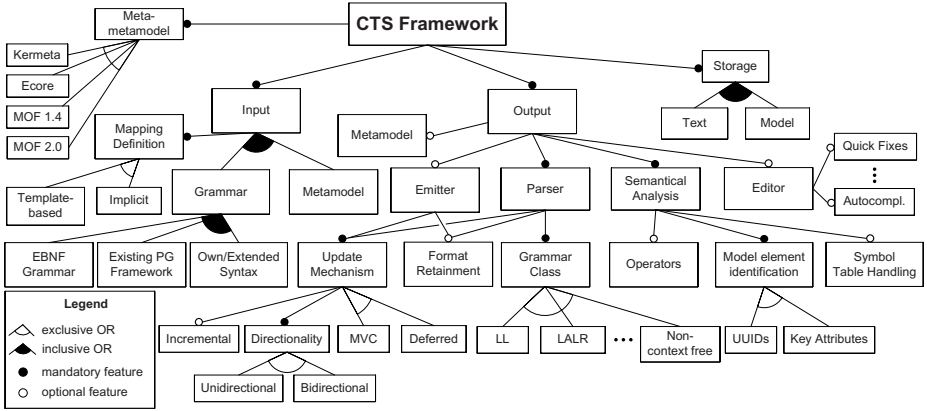


Fig. 2. Feature Diagram of all considered features

in this survey. The features shown in this figure are discussed in detail in the following subsections. How and if these features are provided by the actual approaches is shown in Tab. 1.

3.1 Supported Meta-metamodels (M_3)

Current approaches are based on different meta-metamodels: Ecore [6], different versions of MOF 1.4 [7] or 2.0 [8] or the Kermeta meta-metamodel [9] used by Sintaks [10]. Based on the capabilities of the meta-metamodels also the supported features of the textual syntax approaches vary. For example, MOF 1.4 uses UUIDs (in this case called MOFID) to identify model elements where Ecore uses designated key attributes. Because of these different approaches also the implementation based on one of these meta-metamodels needs to support the respective identification mechanism (see Sect. 5 for a detailed discussion on this problem).

3.2 Input and Output

Depending on the use case for a textual language and its editor, different artefacts may already exist or need to be created. Possible combinations are:

1. Existing language specification, e.g. with a formal grammar, no metamodel exists. This is a typical use case when existing languages, i.e. Domain Specific Languages (DSL), should be integrated into a model driven development environment.
2. Existing metamodel, no specification for concrete syntax. For the development of a new concrete syntax based on an existing metamodel this is an important use case.
3. Both, concrete syntax definition and metamodel exist. For this use case the mapping definition needs to be flexible enough to bridge larger gaps between concrete and abstract syntax (e.g., OCL).

For frameworks which use a grammar as input it should also be distinguished if it is possible to use standard (E)BNF grammars or if a proprietary definition for the CTS constructs is needed. For approaches that specify the concrete syntax based on an existing metamodel a template-based approach that defines how each metamodel element is represented as text may be used. The components needed to translate back and forth between text and model—namely parser and emitter—are considered an output of the CTS framework. This is closely connected to the bidirectionality support of an approach because it is clear that if it only supports one direction, one of these components is not needed. For example, an approach only supporting the translation from the textual syntax to the model representation would not generate an emitter.

In Tab. 1 the following abbreviations denote the input and output parts of the CTS frameworks: E =Emitter, G =EBNF grammar, G_{pg} =Reuse of an existing parser generator grammar definition, G_s =Proprietary grammar definition, M_2 =Metamodel, P =Parser, T =Templates for the concrete syntax.

3.3 Update Mechanism

There are two main possibilities how changes of the text can be reflected in the model. First, a Model View Controller (MVC) like approach may be used. Using an MVC-based editor, all changes to the textual representation are directly reflected in the model and vice versa. This means that there are only atomic commands that transform the model from one consistent state to another. Hence, it is at every point in time consistent. Second, a deferred update approach may be used. The parser is called from time to time or when the text is saved. However, intermediate states of the text may then be out of sync with the model because it may not always be possible to parse the text without syntactical errors. Such an approach is for instance used in the background parsing implementation of the Eclipse JDT project.

These approaches are identified in Tab. 1 by: mvc =Model View Controller, bg =Background parsing.

3.4 Incrementality

If the translation between text and model is done incrementally, only the necessary elements are changed rather than the whole text or model. For example, model elements are kept, if possible, when the text is re-parsed. Vice versa, changes to the model would only cause the necessary parts of the text to be updated. Especially when dealing with models in which model elements are identified by a UUID an incremental update approach becomes more desirable. Here, incrementality is important to keep the UUIDs of the model elements stable so that references from other models outside the current scope do not break. Therefore it is important not to re-create model elements every time a model is updated from its CTS. A detailed discussion on the issues that arise when using a CTS approach on top of a UUID-based repository is performed in Sect. 5.

Even in a non UUID-based environment incrementality becomes important as soon as the textual representation reaches a certain size. Lexing, parsing, semantical analysis and instantiating model elements for the whole text causes a significant performance overhead.

In Tab. 1 the following abbreviations are used to distinguish these possibilities: y =Full support for incremental parsing/updating, n =No support for incremental updates, $y([p|e])$ =Support only for incremental parsing(p) or emitting(e).

3.5 Format Retainment

If an emitter is used to translate models to their textual representation, users would expect that the format information of the text, such as whitespaces, is preserved. Furthermore, elements that are only present in the concrete syntax and not in the metamodel, as for example comments, also need to be retained. Especially when the textual representation is not explicitly stored but rather derived from the actual model (c.f., Sect. 3.12) format information has to be stored in addition to the model.

Possible values for this feature in Tab. 1 are the following: y =Format is retained upon re-emitting, n =No format retainment support.

3.6 Directionality

Bidirectional transformations between the abstract model representation and its CTS means that it is also possible to update existing textual representations if the model has changed. An initial emitter that produces a default text for a model can easily be produced using the information from the grammar or mapping definition. For a more sophisticated emitter, knowledge about formatting rules and format retainment is needed.

For updating existing representations, it would be expected that the user-defined format is retained. Imagine a textual editor that is used to create queries on business objects. Now an attribute in the business object model is renamed. This means that all references in the queries need to be updated. Hence, the queries need to be re-emitted from the model. For this case it would be desirable that the queries' format looks exactly the same as before that change rather than having the default format.

There are some difficult cases that should be considered: Imagine a series of inline `"/"` comments that the user aligned nicely. When the length of the identifier changes, it will be tricky to know what the user wanted with the formatting: aligned comments or a specific number of spaces/tabs between the last character of the statement and the `"/"` marker. Hence, perhaps there needs to be the possibility to specify the behaviour of such formatting rules within the mapping definition.

The following values are possible for this feature in Tab. 1: y =Completely bidirectional transformation, n =No bidirectionality supported, i =Creation of textual representation only initially.

3.7 Grammars Class

The parser component of an approach needs to have a grammar defined to be able to handle the textual input of the concrete syntax. Possible grammar classes are those of general-purpose programming languages such as LL or LALR [11]. However, it might be possible that even non context-free grammars may be used as input. Another possibility where no grammar in a usual form is needed would be a pseudo text editing

approach. In such an approach no text file is edited but all modifications are directly applied to the model using an MVC approach (c.f., Sect. 3.3).

The following grammar classes are considered for Tab. 1: LL(1/k/*), SLR, LR, LALR, ncf=Non-context free, dir.=Direct editing.

3.8 Semantical Analysis

After the parser has analysed the structure of the text document links between the resulting elements need to be created. For example, a method call expression in an OCL constraint that was just parsed needs to be linked to the corresponding operation model element. To represent these links, two different concepts may be used by the model repository, either by their UUID or by designated key attributes (c.f., Sect. 5). As the choice of one of these mechanisms has a great impact on the implementation of the CTS approach (also see Sect. 5) this feature is also listed in this classification schema.

The following abbreviations are used in Tab. 1: UUID=Identification via UUID, Key-Attr.=Identification by designated key attributes.

3.9 Operators

Especially in mathematical expressions the use infix operators is widely spread. During the semantical analysis the priorities, arities and associativities of such expressions have to be resolved. To be able to automatically translate a textual representation of such an expression into its abstract model this information needs to be present in the mapping definition. If such an automated support is present this allows the gap between the metamodel and the grammar to be much bigger. For approaches that generate a metamodel from the mapping definition this feature is mostly implicitly supported since the operator precedence is then directly encoded in the generated metamodel.

In Tab. 1 a *y* means that explicit support for operators is built into the framework, *p* means partial support exists and *n* means that a manual translation is needed.

3.10 Symbol Table

A symbol table is needed to handle the resolving of references within the textual syntax. As there is, mostly, only the containment hierarchy explicitly present within a CTS a symbol table is needed during the parsing process to resolve other references that are stated using e.g., named references. The support for custom namespace contexts (such as blocks in Java) can also be an important feature of the employed symbol table.

Possible values for this feature in Tab. 1 are the following: *y*=full support including custom contexts, *p*=partial support without additional contexts, *n*=no built-in symbol table.

3.11 Features of the Generated Editor

Most approaches also generate a comfortable editor for the concrete syntax. Functionality that is based on the abstract syntax (such as autocompletion or error reporting)

can be provided based on the model. If the tool also supports bidirectional transformation, refactoring support (such as renaming, etc.) may be easily implemented using the model. Other possible features are syntax highlighting or quick fixes.

Table 2 shows an overview on the features of the generated editors of each framework. The following features are considered: autocompletion, syntax highlighting, refactoring, error markers and quick fixes.

3.12 Storage Mechanism

Having two kinds of representation, i.e. concrete textual syntax or abstract model, there are several possibilities to store the model. First, the model may be stored just using the concrete syntax. Second, only the abstract model is stored and the textual representation is derived on the fly whenever the textual editor opens the model. Then formatting information needs to be stored additionally to the model (c.f., Sect. 3.5). Third, both representations could be stored independent from each other. However, this means in most cases that they are not kept in sync with each other. Fourth, a hybrid approach may be implemented that stores the format information and merges them with the model when it is loaded into the editor. This additional format storage may then again be represented as an annotation model to the actual model or as some kind of textual template.

These different possibilities are identified in Tab. 1 using the following abbreviations: text=The textual representation is stored, mod.=The model is stored, both=Both representations are stored, hyb.=Hybrid storage approach.

4 Classification of Existing Approaches

According to the classification schema presented in Sect. 3, we evaluated several approaches that present a possibility to create a model based CTS. Table 1 lists the supported features of each approach. Table 2 shows the features of a potentially generated editor. All evaluations were based on the cited works and prototypes that were available at the time of writing. Future work proposals of these sources were not considered.

Bridging Grammarware and Modelware: Grammar-based approaches are used to automatically generate a metamodel for the CTS. Those metamodels are closely related to the grammar elements for which they were created. This inherently causes the metamodel to be relatively large. Reduction rules can be used to reduce the amount of metamodel elements that are produced for elements in the grammar.

For example, a trivial mapping would generate a class c_{nt_k} for each non-terminal nt_k in the grammar as well as one class c_{alt_i} for each alternative alt_i of nt_k . Furthermore, an association ref_{alt_i} would be generated that connects the c_{alt_i} to their c_{nt_k} . The c_{alt_i} then reference the corresponding c_{nt_j} for the referenced non-terminals nt_j of alt_i . One reasonable reduction rule for this scenario is: if nt_k references only alt_i with only one referenced non-terminal each, the ref_{alt_i} as well as c_{alt_i} could be omitted, reducing the whole structure to a direct generalisation between the c_{nt_j} and c_{nt_k} .

Some of these reduction rules can be applied automatically during the metamodel generation, while others need additional information given as annotations to the grammars.

Wimmer and Kramler [12] present such an approach. A multi-phase automatic generation that facilitates reduction rules as well as manual annotations reduces this amount to make the resulting metamodel more usable. The reduction steps that are applied during these phases then also implicitly define the mapping between the mapping definition. The main area where such an approach is useful is the Architecture Driven Modernisation (ADM)[16] where existing legacy code is analysed for migration, documentation or gathering of metrics.

Frodo: Frodo [13] was developed with the goal to provide a unified solution for the creation of a DSL. This approach presents an end-to-end solution for textual DSLs, providing support for the creation of a CTS as well as back-end support for the target DSL. It also makes initial attempts to derive a debugging support from the mapping specification. Frodo supports several sources for the definition of the CTS. Either a grammar metamodel may be specified or a specific grammar for a supported parser generator (currently ANTLR) could be used. An implicit mapping from the grammar to the DSL metamodel is automatically created. This is done by matching the names of classes and attributes to elements in the grammar rules. Additional mapping rules, such as those needed for the resolving of references between model elements can be specified on the grammar metamodel.

Grammar Based Code Transformation for MDA: The approach elaborated in [14], similar to **Bridging Grammarware and Modelware**, also relies on reduction and annotations to the grammar. However, this approach additionally facilitates the storage of format information as a decorator model attached to the actual model.

Sintaks (TCSSL): Fondement [20] presents a bidirectional approach that generates a parser (based on the ANTLR parser generator) and an emitter (using the JET template engine). The mapping definition is created using the MOF concrete syntax. For complex mappings which need several passes, e.g., for resolving references or performing type checking, a multiple pass analysis can be integrated into the mapping. The main idea of this approach is to have an n-pass architecture for the transformation from code to model. Intermediate models are hereby treated as models decorated with refinements. Model transformations are then used to subsequently transform these models and finally create the abstract model that then conforms to the target metamodel.

TCS: A similar technique is presented in [10]. This approach also provides a generic editor for the syntaxes handled by TCS. For each syntax that is defined using TCS, there is also a definition that can be registered in the editor. Within this definition, it can, for example, be specified how the syntax highlighting should be done. This editor uses text-to-model trace-links that are created during parsing to allow hyperlinks and hovers for references within the text. However, currently these links are implemented as attributes (column and line number of the corresponding text) on a mandatory base class (LocatedElement) for all metamodel elements handled by a TCS editor. If the metamodel classes do not extend these class, the trace functionality is disabled. In later versions of TCS, this issue might be resolved. The mapping definition of a TCS syntax is tightly coupled to the metamodel, which means that for each element in the metamodel there is one rule describing its textual notation. This tight coupling makes the definition of a syntax relatively easy. However, it is therefore not possible to define additional

Table 1. Comparison of related approaches.

Name(s)	Ref.	Input	Output	Bid.	Updates	Inc.	Grammars	Format	M_3	Ident.	Oper.	Symb. Tab.	Storage
Bridging Grammarware and Modelware	[12]	G	M_2, P, E	y	bg	n	LL(k)	n	Ecore	KeyAttr.	n/a		p
Frodo	[13]	G or G_{PG} , M_2 , T^v	P, E	y	bg	n	LL(*)	n	Ecore	KeyAttr.	n		p
Grammar Based Code Transformation for MDA	[14]	G_{PG}^b	M_2, P, E	y	bg	n	LALR(1)	y	MOF 1.4	KeyAttr.	n/a		p
Gymnast	[15]	G	M_2, P, E	y^c	bg	n	LL(k)/LL(*)	y	Ecore	KeyAttr.	n/a		n
HUTN	[23]	M_2	G, P, E	y	bg/mvc ^d	n	n/a^d	n	MOF 1.4	KeyAttr.	n		p
JetBrains MPS	[17]	G_s	M_2, P, E	y	mvc	$y^{e'}$	y^f	y	prop.	y^f	y		mod.
MontiCore	[19, 18]	$I(M_2, G)^s$	P, E	y	bg	n	LL(k)	n	Ecore	KeyAttr.	n		y
TCS	[10]	M_2, T	G, P, E	y	bg	n	LL(*)	n	Ecore	KeyAttr.	y		text
Sintaks(TCSSL)	[20]	M_2, T	G, P, E	y	bg	n	LL(*)	n	Kerneta	KeyAttr.	p		both
TEF	[21]	M_2, T	G, P	n	bg	n	SLR, LR, LALR	n	Ecore	KeyAttr.	y		text ^g
xText	[22]	G_s	M_2, P	n	bg	n	LL(*)	n	Ecore	KeyAttr.	y		p

Legend:

Input/Output: E =Emitter, G =Grammar, G_s =Own Grammar Definition, G_{pg} =Reuse of parser generator framework definition, M_2 =Metamodel, P =Parser, T =Templates for the CS

Updates: bg=Background parsing, mvc.=Model View Controller based

Storage: mod.=Model

^aFrodo allows the import of standard EBNF or ANTLR grammars that can then be annotated with mapping rules.

^bCurrently there are implementations for SableCC and ANTLR.

^cNavigation only, no transformation from model to text.

^dThis depends on the actual implementation of the HUTN standard.

^eSeems to be supported somehow due to MVC approach.

^fThis feature could not be evaluated.

^gThe metamodel as well as the concrete syntax are defined within the same file.

^hIt is possible to directly access the underlying model via a special implementation of the Eclipse DocumentProvider interface.

rules in the mapping that are e.g., needed to resolve left recursions [11] within a LL grammar.

MontiCore: Another approach to integrate a textual concrete syntax with a corresponding metamodel is presented by Krahn et al. in [18]. This approach facilitates an integrated definition where the abstract syntax (the metamodel) is also defined within a grammar like definition of the concrete syntax. For simple languages and especially for languages where only one form of presentation, i.e. the textual syntax, is used, this approach seems to be promising. However, if a metamodel may have several presentation forms, or if only parts of the metamodel are represented as text, the tight integration of concrete and abstract syntax this approach promotes does not seem to be applicable. MontiCore allows the composition and inheritance of different languages and provides a comfortable support for generating editors from these composite specifications [19].

HUTN: The Human-Usable Text Notation (HUTN) approach [23], now specified as a standard by the Object Management Group (OMG) can be used to generate a standard textual language for a given metamodel. It focuses on an easy-to-understand syntax as well as the completeness of the language (it is able to represent all possible metamodel instances). Furthermore all languages, though each language is different, conform to a single style and structure and it is not possible to define an own syntax for a metamodel. In HUTN a grammar for a metamodel, including parser and emitter is generated.

There were currently only two implementations for HUTN. An early implementation was developed by the DSTC, named TokTok. However, this implementation is not available anymore. Another implementation was developed by Muller and Hassenforder [24], who examined the applicability of HUTN as a bridge between models and concrete syntaxes. They identified several flaws in the specification that make it difficult to use.

TEF: The first version of the Textual Editing Framework (TEF) presented in [21] was based on an MVC updating approach. Inherent problems with this concept (see Sect. 5) led to the choice of background parsing as the final method for updating the model. An interesting feature of TEF that is not directly mentioned in the classification schema is the possibility to define multiple syntactic constructs for the same metamodel element. Vice-versa, it is also possible to use the same notation for different elements by providing a semantic function that selects the correct function based on the context.

JetBrains MPS: JetBrains developed a framework called Meta Programming System (MPS) [17,25] that allows to define languages that consist of syntactical elements that look like dynamically-arranged tabulars or forms. This means, that the elements of the language are predefined boxes which can be filled with a value. MPS follows the MVC updating approach that allows for direct editing of the underlying model. This allows the editor to easily provide features such as syntax highlighting or code completion. However, it is not possible to write code that does not conform to the language. Hence, a copy, paste, adapt process is not possible in this approach.

xText: Developed as part of the openArchitectureWare (oAW) framework, xText [22] allows the definition of a CTS within the oAW context. xText generates an intermediate metamodel for the concrete syntax from the mapping specification. For this reason the framework provides an EBNF-like definition language which facilitates features like

Table 2. Editor capabilities

Name(s)	Reference(s)	Autocomp.	Err. mark.	Refactoring supp.	Quick fixes
Frodo	[13]	n	? ^a	n	n
Gymnast	[15]	y	y	y	n
IntelliJ MPS	[17]	y	y ^b	y	y
MontiCore	[19]	y	y	y	n
TCS	[10]	n	y	n	n
TEF	[21]	y	y	y	n
xText	[22]	y	y	y	n

^aThis feature could not be evaluated.

^bNo syntactic errors possible because of resolute MVC concept.

the possibility to specify identity properties or abstract classes. A translation into an instance of the intended target metamodel needs to be done by developing a model to model transformation from this intermediate language into the target abstract syntax. Having such an intermediate metamodel complicates the bidirectional mapping as an additional transformation for the backwards transformation is needed.

Gymnast: Garcia and Sentossa present an approach called Gymnast in [15], which similar to xText generates an intermediate language on which the editor is based. Refactorings, occurrence markings, etc., are provided by the generated editor based. As this work's main focus is on the generation of the editor and its functionality, a mapping to an existing target metamodel has to be developed in addition to the generated tools.

Other Approaches: A recently emerged project on eclipse.org that also aims to tackle the development of CTSs is the **Eclipse IMP** [26]. However, the current focus of the project lies on the easy development of an editor and not on the integration with a model repository. Still, it was stated in the project's declaration of intent that an integration with EMF is projected. Furthermore, there is an eclipse.org project called **Textual Modelling Framework (TMF)**[27] that currently regroups TCS and xText under a common architecture. A different approach is followed by Intentional Software's **Intentional Programming** [28]. Here, it seems to be possible to directly edit the model using a text editor. Multiple syntaxes, also textual ones, can be defined and handled even in the same editor. However, being a proprietary approach, the integration with existing standards-based repositories may be problematic.

5 Discussion

Current CTS implementations span a continuum between text and model affine-frameworks. On the text-end-side there are tools that stem from conventional programming languages. The idea to create development tools that are based on a model rather than on plain text was already developed before model-driven development became prominent. The IDE of Smalltalk systems follows a similar paradigm. Code artefacts are also Smalltalk objects that are edited using a specialised editor. Especially considering refactoring, this is a great advantage. Another example where boundaries between a

textual and a model view on code are starting to blur is the Eclipse Java Development Tools (JDT) project. Even though the Java source files are still stored as plain text files, JDT uses indices and meta-data in the background to be able to provide comprehensive refactoring, navigation and error reporting capabilities. Many tools that were considered here (such as Gymnast or xText) focus on this end of the spectrum. On the other end of this spectrum, there are tools such as the MPS framework treating text as a sequence of frames or compartments containing text blocks. While text-based issues such as diff or merge are solved on text artefacts, solutions for these problems are still immature concerning models. Vice versa, issues that can easily be or are already solved on models, such as the usage of UUIDs for references or partial and combined views on models are challenges that none of the currently available CTS approaches is able to handle.

As it can be seen in Tab. 1 support for incremental model updates is currently not widely available. Only MPS provides some support for this. Furthermore, none of the approaches under evaluation support the UUID identification mechanism for model elements. Even HUTN, which is an OMG standard explicitly specifies that key attributes need to be specified in order to realise model element identification. That lack of these features complicates the application of these approaches in certain environments. The next sections discuss some of the yet untreated issues.

5.1 Universally Unique Identifiers

There are two different possibilities how model repositories can handle links between model elements: either by defining designated key attributes for each model element (as, e.g., in EMF/Ecore) or by assigning each element a Universally Unique Identifier (UUID) that remains stable across the lifetime of the element (e.g, the MOFID in MOF 1.4) [3,4]. Very important issues arise when trying to put a parser-based approach on top of a repository that uses UUIDs to identify model elements. In large scale environments with a high number of model partitions and numerous connections between those partitions, such repositories become very important. In distributed development where developers of one artefact do not always know all referrers from other model partitions to a specific model element it is crucial that elements have stable IDs. Further advantages of the UUID-based approach can be found in [3].

In textual syntaxes identification through UUIDs becomes problematic for several reasons:

- **Storage Mechanism:** If a model artefact is stored using the concrete syntax only, there needs to be the possibility to store these IDs somewhere in the text. However, during development a developer should not see these IDs as they contradict the crisp textual view and make it confusing. On the other hand, if the artefact is solely stored as model, other problems concerning the update from text to model arise (see below).
- **Model Updates:** Updating existing models is an inherent problem of textual notations. In graphical or forms-based modelling only a comparably small set of changes may occur that can easily be wrapped into command structures that are executed in a transactional way, transforming the model from one consistent state to another. In a textual editor this is very difficult, especially when IDs are not present

in the textual representation. A small change, e.g., adding an opening bracket in the text may alter the whole structure of the text, making it difficult to identify which elements in the model are meant to be kept and which are not.

- **Creation and Deletion of Model Elements:** In a graphical editor there are explicit commands to create and delete model elements. Within a textual editor this is difficult mostly because the creation or deletion of model elements is done implicitly. For example, if the name of an element is changed in the textual syntax it may either mean that the old element should be deleted and a new one should be created, or a simple rename of the existing model element may have been intended.

A solution that lets transformations produce stable UUIDs for new model elements was proposed in [4]. Such an approach may, for example, use the ID of the source element and some transformation ID to compute the ID of the target element. However, if text is used as source there is no stable ID for a source element, just properties derived directly from the textual representation, such as a name attribute. Hence, no stable ID for a target element can be computed and the only way to keep the identity is to rely on incremental updates of the model (c.f. Sect. 5.2).

5.2 Update Mechanism

Many of the aforementioned problems may be solved by employing an MVC-based update mechanism. However, there are still other problematic constructs in this concept. Consider for example an expressions such as “(a+b)*c”: At the time of typing the expression in the parentheses, it can not be known that the model that actually should be created would have the “*” expression as root node and the parenthesised expression only as a subnode. A discussion of the advantages and disadvantages of the MVC and the background parsing approach can be found in [21].

Closely related to the update issue is the general problem of incremental updates. In compiler construction literature this problem was already discussed. For example, Wagner [29] developed a methodology that allows incremental lexing as well as parsing. Furthermore, Reps et. al. [30] present an incremental process that allows incremental updates to the attributed trees that result from the semantical analysis. However, such techniques were not adopted by any of the CTS frameworks under evaluation.

5.3 Partial and Combined Views

One advantage of graphical modelling is that it is easily possible to define partial views on models. This means that it is possible to create diagrams that highlight only a specific aspect of the model while hiding other parts. for example, in UML one diagram may be used to display an inheritance hierarchy of classes only while another diagram is used to show the associations between these classes. Defining models with a CTS should also include the possibility to do this. However, this imposes that there are two different modes for deleting elements. One, which deletes only the text and another which deletes the model element from the text *and* the model. Using only standard text editing techniques (typing characters and using backspace or delete to remove them) there is no possibility to distinguish between both commands.

6 Related Work

An analysis of existing approaches for the creation of a CTS was conducted in [13]. However, neither a systematic schema was used to classify these approaches, nor was a comprehensive analysis of the approaches' features provided. Garcia [15] presented an overview of existing approaches and highlighted some of their peculiarities. Nevertheless, an extensive analysis was considered "a lengthy, lengthy, affair" and therefore omitted. Several of the papers presenting the CTS approaches contain a short section on related work, but they also only give hints.

7 Conclusions and Future Work

In this paper we presented a classification schema for CTS mappings and their frameworks. We applied this classification to several academic and industrial approaches. The presented classification is not exhaustive. However, the classification schema can be used to evaluate, classify and compare further approaches. Furthermore, we identified issues that currently complicate the use of these frameworks in a large-scale enterprise MDSD environment. Issues like those highlighted in our discussion need to be solved in order to apply a CTS framework in a larger model-driven environment.

The contribution of this paper is useful for the practitioner, as he can select an adequate approach for his particular requirements based on the presented classification. Furthermore, it is also interesting for academia, as several unresolved issues concerning a CTS mapping are highlighted.

We plan to develop a CTS approach that explicitly supports UUID-based repositories with all consequences, such as the need for incremental parsing and updating. The envisioned approach should also support partial and combined views.

References

1. Object Management Group: Object Constraint Language (OCL) 2.0. Doc. No 05-06-06
2. Grönniger, H., Krahn, H., Rumpe, B., Schindler, M., Völkel, S.: Textbased modeling. In: Proc. of the 4th Int. Workshop on Software Language Engineering (ateM 2007) (2007)
3. Uhl, A.: Model-driven development in the enterprise (2007), <https://www.sdn.sap.com/irj/sdn/weblogs?blog=/pub/wlg/7237>
4. Uhl, A.: Model-driven development in the enterprise. IEEE Software 25(1), 46–49 (2008)
5. Schobbens, P.Y., Heymans, P., Trigaux, J.C.: Feature diagrams: A survey and a formal semantics. re 0, 139–148 (2006)
6. Eclipse Foundation: Eclipse modeling project last visited: 24.01.2008, <http://www.eclipse.org/modeling/>
7. Object Management Group: Meta Object Facility (MOF) 1.4. Doc. No 02-04-03
8. Object Management Group: MOF 2.0 core final adopted specification. Doc. No ptc/03-10-04
9. Muller, P.A., Fleurey, F., Jézéquel, J.M.: Weaving executability into object-oriented meta-languages. In: Proc. of MODELS/UML 2005 (2005)
10. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: GPCE 2006, pp. 249–254 (2006)
11. Muchnick, S.: Advanced Compiler Design and Implementation. Morgan Kaufmann, San Francisco (1997)

12. Wimmer, M., Kramler, G.: Bridging grammarware and modelware. In: Bruehl, J.-M. (ed.) *MoDELS 2005*. LNCS, vol. 3844. Springer, Heidelberg (2006)
13. Karlsch, M.: A model-driven framework for domain specific languages. Master's thesis, University of Potsdam, Hasso Plattner Institute (2007)
14. Goldschmidt, T.: Grammar based code transformation for the model driven architecture. Master's thesis, Hochschule Furtwangen University, Furtwangen, Germany (August 2006)
15. Garcia, M., Sentosa, P.: Generation of Eclipse-based IDEs for Custom DSLs. Technical report, Software Systems Institute (STS), TU Hamburg-Harburg, Germany (2007)
16. Object Management Group: Architecture Driven Modernization (ADM), <http://www.omg.org/adm/>
17. JetBrains: MPS. last visited: 26.03.2008, <http://www.jetbrains.net/confluence/display/MPS/>
18. Krahn, H., Rumpe, B., Völkel, S.: Integrated definition of abstract and concrete syntax for textual languages. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) *MODELS 2007*. LNCS, vol. 4735, Springer, Heidelberg (2007)
19. Krahn, H., Rumpe, B., Völkel, S.: Efficient editor generation for compositional dsls in eclipse. In: *Proc. 7th OOPSLA Workshop on Domain-Specific Modeling (DSM 2007)* (2007)
20. Fondement, F.: Concrete syntax definition for modeling languages. PhD thesis, Ecole Polytechnique Fédérale de Lausanne (2007)
21. Scheidgen, M.: Textual editing framework (2007), <http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/tool.html>
22. Efftinge, S.: Xtext reference documentation (2006), http://www.eclipse.org/gmt/oaw/doc/4.1/r80_xtextReference.pdf
23. Object Management Group: Human-Usable Textual Notation (HUTN) Specification. Doc. No formal/04-08-01 (2004)
24. Muller, P.A., Hassenforder, M.: HUTN as a bridge between modelware and grammarware - an experience report. In: *4th Workshop in Software Model Engineering WiSME 2005* (2005)
25. Dimitriev, S.: Language oriented programming: The next programming paradigm. *onBoard Magazine* 2 (2005)
26. Fuhrer, R.M., Charles, P., Sutton, S., Vinju, J., de Moor, O.: Eclipse IDE Meta-tooling Platform (The Eclipse IMP) (2007), <http://www.eclipse.org/proposals/imp/>
27. Eclipse Foundation: Textual modeling framework. last visited: 24.01.2008, <http://www.eclipse.org/proposals/tmf/>
28. Simonyi, C.: *Intentional software* (2007), <http://www.intentsoft.com/>
29. Wagner, T.A.: *Practical Algorithms for Incremental Software Development Environments*. PhD thesis, University of California at Berkeley (1998)
30. Reps, T., Teitelbaum, T., Demers, A.: Incremental context-dependent analysis for language-based editors. *ACM TOPLAS* 5(3), 449–477 (1983)

Metamodel Syntactic Sheets: An Approach for Defining Textual Concrete Syntaxes

Javier Espinazo-Pagán, Marcos Menárguez, and Jesús García-Molina

University of Murcia, Spain
{jespinazo,marcos,jmolina}@um.es
<http://gts.inf.um.es/>

Abstract. The development process of Domain Specific Languages (DSL) can be tackled from different technical spaces such as XML, Grammarware or Model Driven Engineering (MDE). In the case of using MDE, the definition of a concrete syntax for a textual DSL requires commonly building a bridge between this technical space and Grammarware. Several bridging approaches have been recently proposed in which the existing coupling between concrete and abstract syntaxes causes information duplication in the development process of DSLs. Moreover, reusability of concrete syntaxes has received no attention in these approaches.

In this paper we present the MSS (Metamodel Syntactic Sheets) approach for defining textual concrete syntaxes. MSS is intended to promote the reuse of textual concrete syntaxes and to avoid information duplication. In MSS, metamodels are annotated with syntactic properties and a propagation mechanism reduces the number of annotations required as well as the coupling between concrete and abstract syntaxes. Textual concrete syntaxes can be reused by annotating syntactically the metamodeling language. This reuse makes possible to share syntactic idioms (textual conventions) among different DSLs.

1 Introduction

The development process of Domain Specific Languages (DSL) can be tackled from different technical spaces such as XML, Grammarware or Model Driven Engineering (MDE). MDE techniques and tools provide clear advantages to the development process. First, it supports the definition of domain models by applying the expressiveness of metamodeling languages. Second, it facilitates the consecution of automatic development processes. However, textual DSLs usually need a grammar to specify the concrete syntax, so a bridge between MDE and Grammarware technical spaces is required.

As noted in [1], there is significant redundancy between the grammar of the textual concrete syntax and the metamodel of the abstract syntax. This problem is caused by the coupling between concrete and abstract syntaxes, and is common to all the existing approaches for bridging Grammarware and MDE ([1], [2], [3], [4], [5]). On the other hand, reusability of textual concrete syntaxes has received no attention in these approaches. Thus, proposals intended to avoid

information duplication and to promote the reuse in the development process of DSLs are necessary.

Families of textual DSLs are an example of the need for reusing concrete syntaxes. A textual DSL is often used in combination with other textual DSLs for user communities. These communities establish usability criteria based on syntactic conventions or standards. In this paper we introduce the *syntactic idiom* term to refer to the syntactic conventions shared within the user community of a DSL. Applying syntactic idioms commonly involves the repetition of syntactic constructs along the specification of the concrete syntax, what increases the development and maintenance costs. In this sense, reuse techniques would make the definition of syntactic idioms more efficient.

MSS (Metamodel Syntactic Sheets) is an approach for the specification of textual concrete syntaxes for metamodels, pursuing three main goals: to avoid information redundancy between metamodels and concrete syntaxes; to facilitate reuse of concrete syntaxes; and to support the definition of syntactic idioms. In MSS, a textual concrete syntax consists of *syntactic sheets* containing a set of *syntactic properties* applied to the metamodel elements. A propagation mechanism spreads syntactic properties through the existing relationships between metamodeling concepts such as the *inheritance* in metaclasses. This mechanism also supports the annotation of the metamodeling language, which permits to create syntactic idioms.

This paper is organized as follows. The next section motivates the proposed approach and introduces a running example. Section 3 defines the concept of syntactic property. Section 4 introduces the DSL used to specify textual concrete syntaxes. Section 5 presents the property propagation mechanism. Section 6 describes the processing of syntactic sheets. Finally, in Section 7 related work is presented and Section 8 presents some conclusions and future work.

2 Motivation and Running Example

In MDE, metamodeling techniques are applied to create DSLs, which are devised for specifying models representing different aspects of the system. The abstract syntax of DSLs is represented by a metamodel expressed in a metamodeling language (e.g. EMOF, Ecore or KM3). The formalism used to specify the concrete syntax of a DSL depends on the nature of the notation. In the case of a textual notation, the concrete syntax is commonly specified by a grammar, which associates syntactic properties to metamodel elements.

In this paper we will present an approach for defining textual concrete syntaxes through the annotation of metamodels with syntactic information. In this section we will illustrate the approach through a simple DSL whose metamodel is shown in Figure 1. The purpose of this DSL, named *DataForm*, is to define data forms. A data form consists of a set of groups, and each group includes a number of data fields.

In this example, we propose a *DataForm* DSL specification with a nested syntactic structure, what is very common in textual DSLs. According to this

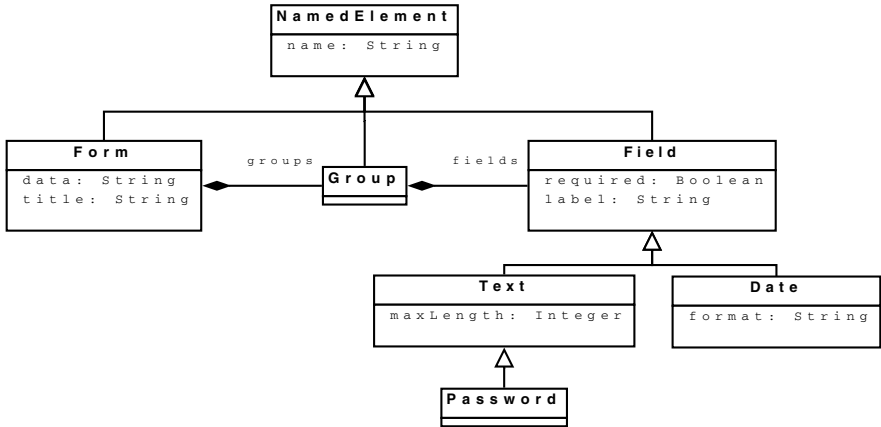


Fig. 1. Abstract syntax of the *DataForm* DSL

structure, a *DataForm* specification consists of a set of syntactic constructs describing metamodel concepts. Each *concept construct* has two parts: *header* and *body*. A header consists of a keyword, whose name is the same as the meta-class name corresponding to the concept, followed by a name identifying a concrete instance of the concept (e.g., `Form "Basic User"`). The body consists of a list of constructs describing the attributes and associations of the concept (metaclass). This list is enclosed between curly brackets and their constructs are separated by a semi-colon. These constructs can be of two kinds: *feature construct* and *aggregation construct*. A *feature construct* consists of the feature identifier and its value, separated by the colon (e.g., `data: "User"`). An *aggregation construct* encloses between square brackets a list of concept constructs separated by a comma. The following text shows a *DataForm* specification.

```

Form "BasicUser" {
  data: "User";
  title: "User Registration";
  groups: [
    Group "Personal Data" {
      fields: [
        Text "name" {
          required: true;
          maxLength: 256;
        },
        Date "birthDate" {
          required: false;
          label: "Birth Date";
          format: "mm/dd/yyyy";
        }
      ]
    }
  ]
}

```

```

    }
  ]
}

```

An excerpt of the grammar of the DSL *DataForm* is presented below. For the sake of clarity, the names of the non-terminal symbols are defined according to the metamodel elements; for instance, `metaclass_form` refers to the declaration of a `Form` metaclass and `feature_form_data` corresponds to the definition of the feature data of the metaclass `Form`.

```

metaclass_form := 'Form' ID '{' feature_form_data ','
               feature_form_title ',' feature_form_groups '}' ;

feature_form_data := 'data' ':' STRING ;

feature_form_title := 'title' ':' STRING ;

feature_form_group := 'groups' ':'
                    ( '[' ']' | '[' metaclass_group (',' metaclass_group)* ']' );

metaclass_group := 'Group' ID '{' feature_group_fields '}'

feature_group_fields := 'fields' ':'
                    ( '[' ']' | '[' metaclass_field (',' metaclass_field)* ']' );

metaclass_field := ( metaclass_text | metaclass_date
                  | metaclass_password ) ;

...

```

Note that the metamodel and the nested syntactic structure of the notation establish the *shape* of the production rules. These rules could be classified according to the three kinds of constructs defined above. Production rules belonging to a category share the same structure, so similarity between rules arises. For instance, the `feature_form_data` and `feature_form_title` productions specify *feature constructs* and they only differ in the first terminal symbol, 'data' and 'title', respectively; and the `feature_form_groups` and `feature_group_fields` productions specify aggregation associations and they have the same structure.

Therefore, each kind of syntactic construct can be considered a *syntactic pattern* or template that can be parameterized with terminal symbols. For instance, the pattern referring to metaclass features would have a parameter used to establish the terminal symbol that appears between the feature name and its value; in the previous grammar, this parameter would have the colon as value. The MSS approach for defining textual concrete syntaxes was influenced by this idea of grammar syntactic pattern. We will use the *syntactic property* term to refer to the pattern parameters, and the *syntactic annotation* term to the assignment of a value to a property.

We have identified a set of useful syntactic patterns to specify textual concrete syntaxes. These patterns and the related syntactic properties are presented in the next section. In MSS, metamodel elements are annotated with syntactic properties in order to derive a grammar for processing the textual notation. To express syntactic annotations, we have defined a DSL which is presented in Section 4.

Besides identifying a set of basic syntactic properties for defining textual concrete syntax, the idea of syntactic pattern has been very useful to discover how to achieve reuse of textual concrete syntax. A pattern has a *scope* establishing the set of metamodel elements to which is applied. The scope can refer to elements of a metamodeling language (i.e. metaclass, attribute or association) or elements of the metamodel (i.e. the metaclass *Form* or the attribute *label* of the metaclass *Field*), that is, the scope can refer to M3 and M2 levels of the four-level architecture of the metamodeling paradigm. Therefore, the pattern scope allows us to distinguish between specific patterns coupled to a DSL metamodel and general patterns that can be applied to several DSLs.

Obviously, general patterns are more interesting since the annotation of syntactic properties at metamodeling language level allows for reusing textual concrete syntax. In this sense, MSS incorporates a propagation mechanism, which is in charge of propagating the syntactic annotations through the relationships between metamodel elements and between a metamodel and its meta-metamodel. In this way, MSS avoids information redundancy between metamodels and concrete syntaxes, and makes possible sharing *syntactic idioms* among DSLs. Section 5 explains in detail the propagation mechanism of syntactic annotations.

3 Syntactic Properties

In this section we identify the set of syntactic properties included in MSS. These properties have been extracted from a set of basic syntactic patterns which has been defined to reach a trade-off between expressiveness and simplicity. For instance, we have considered that each concept (metaclass) uses one feature as identifier. Next, syntactic patterns are presented enclosing the name of the syntactic properties between parentheses.

1. The specification of a DSL concept starts with the name of the metaclass (*metaclass identifier*) and is followed by the identifier of the concept (*main feature*).
2. The features of a concept are enclosed between two delimiters (*content begin/end delimiter*) and the features are separated by a string (*content separator*).
3. Boolean attributes can be located before the name of the concept (*prefix features*). These features can be used as *modifiers* of the concept. The values of these attributes have to be written following lexical patterns (*true/false boolean pattern*), e.g. a true value could be represented by the name of the attribute prefixed by "is-".

4. Some features may appear after the identifier of the concept (*header features*).
5. The definition of a metaclass feature starts with its name (*feature identifier*) and is followed by a string (*value separator*), this preceding the value of the feature.
6. The values of metaclass features with cardinality greater than one are defined as lists. The elements are enclosed between two delimiters (*multivalued begin/end delimiter*) and separated by a string (*multivalued separator*).
7. The value of a non-composite association (i.e. cross reference) is formed by the name of the metaclass of the opposite side of the association (*reference metaclass identifier*) and the identifier of the concept to which it refers.

As it can be noted, the syntactic patterns 1-4 define syntactic properties for metaclasses, pattern 5 is referred to metaclass features, pattern 6 is applicable to multivalued features, and pattern 7 to non-composite associations. Moreover, all the properties applicable to metaclass features can also be applied to metaclasses in order to reduce the coupling between the definition of the textual concrete syntax and the metamodel. The propagation mechanism explained in Section 5 spreads syntactic annotations from metaclasses to their features.

Syntactic properties corresponding to the name of a metamodel element, i.e. *metaclass identifier*, *feature identifier* and *reference metaclass identifier*, can be established with special values such as "auto" or "auto-uppercase", which means the exact name of the element or the name of the element in uppercase, respectively.

To illustrate the syntactic properties identified above, we show an example of concept definition according to the *DataForm* metamodel of Figure 1, which is based on the patterns introduced above.

```
not-required DATE "birthDate" label: "Birth Date" {
    format: "mm/dd/yyyy";
}
```

In this example a **Date** field of a data form is defined. Notice that the metaclass **Date** has four attributes: **name** and **required** are inherited from the metaclasses **NamedElement** and **Field**, respectively, and **label** and **format** are declared in the metaclass. The attribute **required** is a *prefix feature* defined according to the textual pattern "not-" (*false boolean pattern*), and then the *metaclass identifier* is in uppercase. The attribute **name** is the *main feature* so it is defined only with its value. The attribute **label** is a *header feature* and the attribute **format** is enclosed between curly brackets (*content delimiters*). Both attributes are defined according to the same pattern: the name of the feature (*feature identifier*) is followed by colon (*value separator*) and then the value. In the next section we will describe a DSL for specifying syntactic annotations and we will define the textual concrete syntax for the *DataForm* metamodel.

The syntactic properties identified above allow us to generate a grammar for processing DSL specifications. However, the parser also needs to convert a tree-based representation produced by a grammar into a graph-based representation

in order to create models as instances of metamodels. Therefore, non-composite associations have to be resolved in order to obtain the cross-references among model elements. In MSS, the syntactic property *main feature* of the metaclass participating in an association denotes the feature used to match reference values. Since candidate objects can be fetched locally (i.e. from the concept that is being defined) or globally (i.e. starting from the root concept), a property named *association context* will define the search scope, and besides the metamodel is constrained to having one root concept.

4 Metamodel Syntactic Sheets

This section describes the DSL proposed for annotating metamodels with syntactic properties. We also refer to this DSL as MSS (Metamodel Syntactic Sheet). The metamodel that represents the abstract syntax of MSS is depicted in Figure 2. MSS provides the *syntactic sheet* and *syntactic rule* concepts in order to organize the annotation of syntactic properties for a metamodel. A syntactic sheet consists of a set of syntactic rules expressing one or more syntactic properties to be applied to a target metamodel element. A rule has a selector identifying the target element (i.e. metaclass or feature) and a body enclosing the list of syntactic annotations.

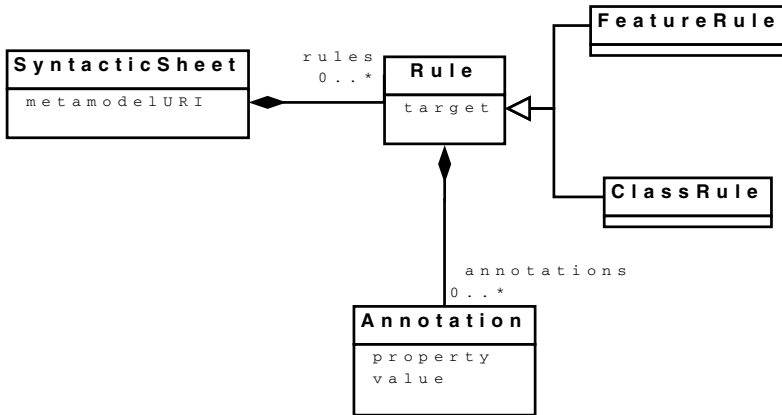


Fig. 2. Abstract syntax of the MSS DSL

A syntactic sheet for the *DataForm* metamodel of Figure 1 is shown below. The *class* and *feature* keywords are used to distinguish the two kinds of rule.

```
metamodel "http://gts.inf.um.es/data-form/";
```

```
class NamedElement priority: 1 {
    main-feature: "name" ;
    metaclass-identifier: "auto";
}
```

```

class Group {
    value-separator: "->";
}
class Field {
    metaclass-identifier: "auto-uppercase";
    prefix-features: "required";
    header-features: "label";
}
feature Form.groups {
    feature-identifier: "";
    value-separator: "";
}

```

Note that a statement defining the reference URI of the metamodel to be syntactically annotated precedes the set of rules in the syntactic sheet. In this example, the metamodel is annotated by means of four rules. Three of them are applied to metaclasses and one rule is applied to a metaclass feature. The purpose of these rules is described next.

The first rule is applied to the abstract metaclass **NamedElement**. It expresses two properties: i) the attribute **name** defined in the metaclass has to be used as *main feature* to identify the concepts of the DSL, and ii) the definition of the DSL concept starts with the name of the metaclass. This rule is applied to a metaclass that is ancestor of all the metaclasses of the DSL, so the propagation mechanism will spread the two syntactic annotations by means of the inheritance relationship. Finally, a priority value is assigned to this rule. Prioritized rules are relevant to choose an annotation in the presence of syntactic collisions. In Section 5 the technique used to resolve collisions is discussed as part of the propagation mechanism.

The second rule is applied to the metaclass **Group**. This rule defines the string "**– >**" as the *value separator* for the features of the metaclass. The metaclass **Group** only has one feature (i.e. **fields**), but this rule is applicable to new features. So, this rule relies on the containment relationship between metaclasses and features to propagate the syntactic annotations.

The third rule is applied to the metaclass **Field**. It expresses three properties: i) the declaration of data fields uses the name of the metaclass in uppercase, ii) the feature **required** prefixes the declaration, and iii) the feature **label** is included in the header of the concept declaration. These annotations may be propagated to the descendants of the metaclass **Field** (i.e. **Text**, **Password** and **Date**) if the properties are not annotated for these metaclasses.

The last rule is applied to the feature **groups** belonging to the metaclass **Form**. This rule contains two properties denoting that the name of the feature and the separator have to be omitted. This is the most specific rule in the syntactic sheet since the annotations are only applied to a metamodel element and no propagation is possible.

As we indicated previously, MSS allows the definition of *syntactic idioms* by annotating the metamodeling language with syntactic properties, which are

shared by several DSLs. Sharing is achieved by the propagation of syntactic annotations from meta-metamodel concepts to metamodels. The syntactic conventions established for a family of DSLs are expressed by a syntactic sheet defined for the metamodeling language. The syntactic sheet below is an example of syntactic idiom defined for the Ecore metamodeling language. The first rule defines syntactic properties common to all the metaclasses because the target element is `EClass`, whereas the second one is applied to the metaclass `EStructuralFeature`, and it consists of syntactic annotations for structural features of metaclasses, i.e. attributes and associations.

```
metamodel "http://www.eclipse.org/emf/2002/Ecore";
```

```
class EClass {
    metaclass-identifier: "auto-lowercase" ;
    feature-identifier: "auto" ;
    value-separator: ":" ;
    content-begin-delimiter: "{" ;
    content-end-delimiter: "}" ;
    content-separator: ";" ;
    true-boolean-pattern: "is-*" ;
    false-boolean-pattern: "not-*" ;
}
class EStructuralFeature {
    multivalued-begin-delimiter: "[" ;
    multivalued-end-delimiter: "]" ;
    multivalued-separator: "," ;
}
```

The definition of a textual concrete syntax can require one or two syntactic sheets. A syntactic sheet for the metamodel is enough, although another sheet for the metamodeling language can be used. The metamodeling language sheet could even be enough if a DSL has no specific notation. In this case, the coupling between the textual concrete syntax and the metamodel would be completely eliminated. In this section we have introduced two syntactic sheets in order to illustrate the advantages of reusing a syntax idiom and the option of adjusting syntactic properties on specific metamodel elements.

5 Propagation of Syntactic Annotations

The mechanism in charge of propagating syntactic property annotations through metamodel elements is described in this section. The mechanism is illustrated by the propagation of the annotations of the two syntactic sheets introduced in the previous section.

In a sheet, syntactic rules are used to define the values assigned to the target element properties. However, a rule needs not to provide a value for each property, because property annotations can be obtained by *propagation*. The

propagation is carried out through the existing relationships among metamodeling concepts. Specifically, three relationships are used: i) *inheritance* relationship between metaclasses, ii) *containment* relationship between a feature and the metaclass containing it; and iii) *instanceof* relationship between a metamodel element and its metaclass in the meta-metamodel (e.g. **Form** is an instance of the Ecore metaclass **EClass**).

In the propagation process metaclasses can obtain syntactic annotations from their parents, whereas metaclass features can obtain the annotations from the metaclass they belong to. Combining propagation through inheritance and containment relationships reduces notably the number of syntactic annotations to be included in the syntactic sheet of a metamodel. For example, the annotation of abstract metaclasses, such as **NamedElement** in Figure 1, allows us to share the definition of syntactic properties among all the elements of a metamodel. However, the syntactic annotation of the meta-metamodel provides the highest reuse of textual concrete syntaxes because the syntactic property annotations can be shared among several DSLs.

Given a metamodel element, the propagation of syntactic annotations may cause the collision of values annotating a property of the element, if various annotations are applicable. This problem is partially resolved by prioritizing the three relationships used for the propagation. The inheritance and metaclass-feature containment relationships cannot be applied to the same element (class or feature) at the same time. So, it is only necessary to define the priority of the instanceof relationship in relation to the inheritance and containment relationships. We decide to prioritize inheritance and containment over instanceof, since the syntactic annotations defined on the metamodel are more specific than the annotations defined in the metamodeling language.

On the other hand, since metaclasses can have more than one parent, collisions may appear in a inheritance-based propagation. This situation cannot be automatically resolved, so information is required in the syntactic rules applied to the involved metaclasses. In the metamodel syntactic sheet of Section 4, it can be noted that a priority value has been assigned to the annotations of the metaclass **NamedElement**. This value is used in case of inheritance collision, but the resolution of the collisions depends on the correct assignment of the priorities by the user.

The propagation mechanism works in the following way. Firstly, the syntactic property annotations of each metaclass are checked. Then, missing annotations are required from the parent metaclasses. If a property can be obtained from various parent metaclasses, then the priority values are applied. The propagation through the inheritance relationship does not guarantee obtaining all the properties. Thus, the annotations not solved by the inheritance can be obtained from the meta-metaclass (**EClass** in Ecore language) applying the instanceof relationship. Therefore, all the concepts of the Ecore language are expected to be fully annotated as we discuss in Section 6. Once the propagation has been applied to the metamodel metaclasses, then the propagation process goes on with the features of metaclasses. Now, the propagation uses the containment

relationship instead of inheritance relationship, and the instanceof relationship is used to propagate annotations from the meta-meta-classes representing attributes (**EAttribute** in Ecore) and associations (**EReference** in Ecore).

Now we will illustrate the propagation mechanism by analyzing annotations applied to the metaclass **Group** and its feature **fields** of the *DataForm* meta-model. The syntactic annotations are taken from the syntactic sheets presented in Section 4. Figure 3 shows all the propagations that take place, which are explained below.

In the metamodel syntactic sheet, the metaclass **Group** is annotated by a rule which has only one property value, so the rest of syntactic annotations have to be obtained by propagation (syntactic properties required for metaclasses are presented in Section 3). Firstly, the inheritance relationship would provide two annotations from the metaclass **NamedElement**. Note that a priority value has

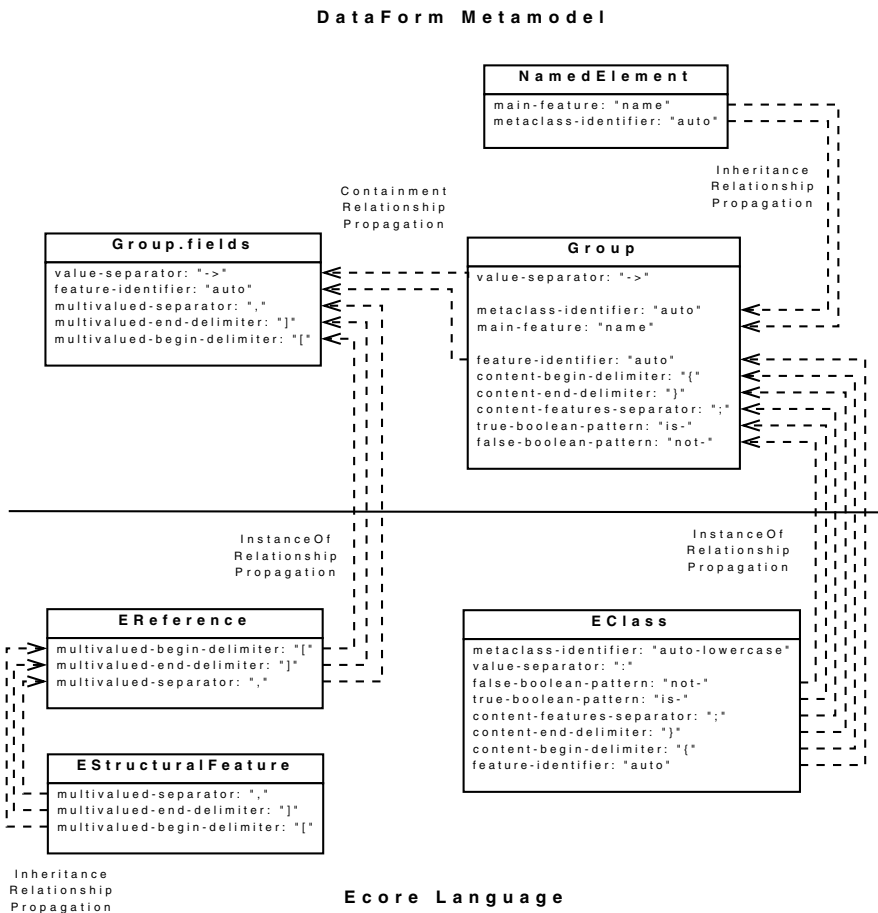


Fig. 3. Propagation of syntactic annotations for the metaclass **Group** and its feature **fields**

been assigned to the annotations of the metaclass `NamedElement`, but it is not used because the metaclass `Group` only inherits from `NamedElement`.

As the application of the inheritance relationship is not enough to complete the syntactic annotations required for the metaclass `Group`, the rest of missing annotations must be obtained through the `instanceof` relationship. So, six annotations are propagated from the Ecore metaclass `EClass`. Note that there exists a collision for the property *metaclass-identifier*, but the inheritance prevails over the `instanceof` relationship as it is indicated above.

Once the metaclass `Group` is fully annotated, the propagation mechanism has to complete the annotations of their features. In this case, the feature `fields` has not been annotated in the metamodel syntactic sheet so it is necessary to obtain annotations for all their properties. Initially, annotations are fetched from the metaclass `Group` containing the feature, but only two annotations are obtained. Therefore, the missing properties have to be obtained from the metamodeling language. The feature `fields` is an instance of the metaclass `EReference` in Ecore. This metaclass has not been annotated in the Ecore syntactic sheet, but inherits the syntactic annotations from `EStructuralFeature`. Afterwards, the feature `fields` is completely annotated with the syntactic properties needed for the generation of the grammar.

6 Processing of Textual Concrete Syntaxes

A textual concrete syntax processor is a tool whose purpose is to generate the software artifacts required to bridge Grammarware and MDE technical spaces: a parser that creates a model equivalent to a DSL textual specification, and a tool that generates the textual representation of a model. Our approach is supported by a processor which generates these tools taking as input the syntactic sheets defining the concrete syntax and the metamodel of the abstract syntax.

Processing MSS concrete syntaxes requires all the metamodel elements to be completely annotated with syntactic properties. The propagation mechanism is used to fulfill this requirement, but it is not enough. Since this mechanism relies on the `instanceof` relationship to obtain the missing annotations, it is necessary to establish all the syntactic annotations in the metamodeling language. However, a syntactic sheet defined for the metamodeling language could neglect some property. This problem can be resolved by defining a complete syntactic sheet for the metamodeling language with default values for all the syntactic properties, and always processing this sheet first. Another syntactic sheet for the metamodeling language can be provided in order to replace the default values.

Figure 4 shows the elements involved in the processing of the *DataForm* textual concrete syntax in MSS. As it can be noted, the MSS processor may have as input three syntactic sheets: the default value syntactic sheet, the metamodeling language syntactic sheet, and the metamodel syntactic sheet, but only the first one is mandatory and it is provided by the tool. Four files are generated as result of processing these sheets: i) a textual file containing a complete syntactic sheet with the annotations of all the metamodel elements; ii) another textual

file containing a grammar definition with no semantic actions; iii) a parser definition for ANTLRv3 that creates models conforming to the metamodel; and iv) a model containing all the annotations of the metamodel elements is obtained, which will be used by an ANT task for the generation of the textual representation of models. The MSS processor is implemented in Java and uses EMF for model management.

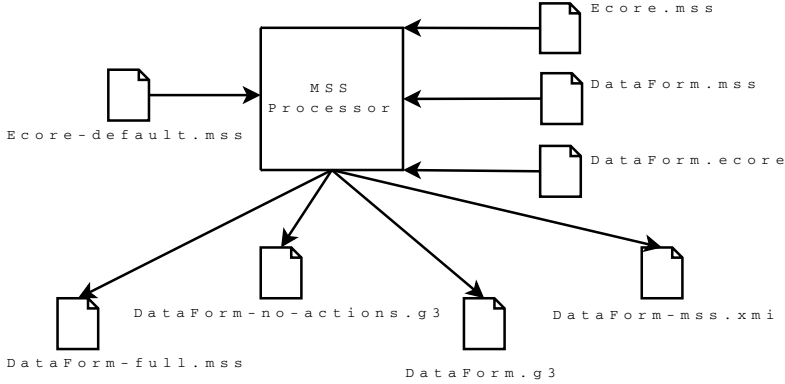


Fig. 4. Diagram of the processing of *DataForm* textual concrete syntax

7 Related Work

There are several approaches for bridging Grammarware and MDE technical spaces. These approaches can be categorized in two groups. The first one focuses on generating metamodels from grammars, whereas the second one goes in the opposite direction with grammars being generated from metamodels. The grammar-based approaches are represented by the xText [2] tool and the works of Wimmer *et al.* [3] and Kunert [4], and the metamodel-based approaches are represented by TCS [1] and XMF-Mosaic [5] tools.

xText is a component of the toolkit OpenArchitectureWare [2]. It is used to build textual DSLs in the Eclipse platform. xText provides an EBNF-based DSL to specify textual concrete syntaxes. The processing of the concrete syntax generates the Ecore metamodel of the abstract syntax of the DSL. The derivation of metamodels from concrete syntaxes relies on a number of rules that, for instance, are used to identify concept hierarchies or to generalize features in a hierarchy of concepts. Despite these rules, the quality of metamodels is poor because grammatical aspects of concrete syntaxes remain in metamodels. Except for simple DSLs, it is usually necessary to define a more suitable metamodel for the DSL. In this case, the definition of a model transformation from the metamodel generated by xText to the defined metamodel is required.

Wimmer *et al.* [3] propose a generic framework for bridging Grammarware and MDE. The framework introduces a basic set of rules to produce metamodels from grammars in a first stage. Next, the application of heuristics improves

the quality of metamodels, but a user-driven stage is needed to refine the metamodels. Similarly, Kunert [4] chooses to add annotations to the grammar in order to drive the generation process, but the annotations are rather limited. These approaches face the problem of the poor quality of the derived metamodels since the expressiveness of grammar formalisms is lower than metamodeling languages.

TCS [1] is a tool integrated in the AMMA platform which provides a DSL to specify textual concrete syntaxes for metamodels. A textual notation consists of grammatical templates to syntactically annotate metamodels. In this sense, MSS adopt the same approach for defining textual concrete syntaxes as TCS. However, the analysis of several DSLs specified in TCS [6] reveals redundancy in templates and high coupling between the concrete syntax and the metamodel, since TCS does not provide a framework to reuse syntactic annotations within a metamodel and between DSLs. In MSS, syntactic annotations are reused by means of a propagation mechanism that takes advantage of relationships defined in the metamodeling language.

XMF-Mosaic [5] includes a DSL named XBNF for defining the textual processing of DSLs. XBNF is an extension of BNF that associates grammar production rules with metaclasses and includes constructions to create models. In fact, this approach is fairly similar to TCS since grammar production rules are templates for processing metaclasses of the metamodel. Consequently, the XMF-Mosaic approach faces the same problems as TCS, i.e. high coupling between concrete and abstract syntaxes and lack of support for reusing textual concrete syntaxes.

The reusability of textual concrete syntaxes has been addressed in [7]. The authors state the advantages of using a common and usable textual notation for metamodels and an approach to derive grammars from metamodels according to the HUTN specification [8] is proposed. In this sense, the Epsilon project [9] provides a textual editor for processing HUTN concrete syntaxes of Ecore metamodels in Eclipse. However, both approaches are restricted to HUTN and it is not possible to define other common textual notations. MSS allows us to define HUTN and other syntactic idioms by annotating the metamodeling language syntactically. Moreover, it supports the adaptation of the textual concrete syntax throughout the development process of a DSL to the specific requirements of its metamodel.

8 Conclusions and Further Work

In this paper we have presented the MSS approach for defining textual concrete syntaxes. Compared to the existing approaches, MSS is characterized for two main concepts: syntactic property and propagation of annotations of syntactic properties. A textual concrete syntax consists of a set of syntactic property annotations applied to metamodel elements. MSS provides the so-called syntactic sheets as a convenient way of organizing the metamodel annotations by means syntactic rules.

The propagation of syntactic annotations facilitates two ways of reusing textual concrete syntaxes: i) reusing annotations within of the same metamodel, and

ii) reusing syntactic idioms defined by annotating the metamodeling language. The main contributions of the MSS approach would be:

- Provision of reuse techniques for the development of textual concrete syntax.
- Creation of syntactic idioms.
- Reduction of information duplication due to the decrease of coupling between abstract and concrete syntaxes.
- A software processor which generates the software tools to bridge Grammarware and MDE technical spaces.

MSS has been applied for prototyping DSLs in several research projects. It also has been used for providing textual concrete syntaxes to existing DSLs such as the model transformation language RubyTL [10]. Currently we are considering new syntactic patterns and properties in order to enhance the expressiveness of concrete syntaxes and MSS is being integrated in a framework for DSL development.

Acknowledgments

This work has been partially supported by Fundación Séneca, grant 05645/PI/07, and by Government of Región de Murcia (Spain), grant TIC-INF 06/01-0001.

References

1. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In: Jarzabek, S., Schmidt, D.C., Veldhuizen, T.L. (eds.) GPCE, pp. 249–254. ACM, New York (2006)
2. OpenArchitectureWare, <http://www.openarchitectureware.org>
3. Wimmer, M., Kramler, G.: Bridging Grammarware and Modelware. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 159–168. Springer, Heidelberg (2006)
4. Kunert, A.: Semi-automatic Generation of Metamodels and Models from Grammars and Programs. In: Fifth Intl Workshop on Graph Transformation and Visual Modeling Techniques. Electronic Notes in Theoretical Computer Science (2006)
5. Clark, T., Sammut, P., Willans, J.: Applied Metamodelling, A Foundation for Language Driven Development, 2nd edn. Xactium (2008)
6. TCS Zoo, <http://wiki.eclipse.org/TCS/Zoo/>
7. Muller, P.-A., Hassenforder, M.: HUTN as a bridge between modelware and grammarware. In: WISME Workshop, MODELS / UML 2005, Montego Bay, Jamaica (October 2005)
8. Object Management Group. Human-Usable Textual Notation 1.0. omg/ 2004-08-01. OMG document (2004)
9. Epsilon project, <http://www.eclipse.org/gmt/epsilon/>
10. Sánchez, J., García, J., Menárguez, M.: RubyTL: A Practical, Extensible Transformation Language. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 158–172. Springer, Heidelberg (2006)

Graphical Concrete Syntax Rendering with SVG

Frédéric Fondement

ENSISA, MIPS
Université de Haute Alsace
12, rue des frères Lumière
F-68093 Mulhouse, France
frederic.fondement@uha.fr

Abstract. Model-based techniques place modeling at the cornerstone of software development. Because of the large number of domains and levels of abstraction one can find in software systems, a large number of modeling languages is necessary. Modeling languages need to be properly defined regarding concrete syntax in addition to abstract syntax and semantics. Most modeling languages use a graphical concrete syntax, and solutions to model those syntaxes appeared. If those solutions are convincing to support the rapid development of graphical modeling tools, they are often restrictive in the range of possible concrete syntaxes for a given abstract syntax, and rely on dedicated technologies. In previous works, we proposed such a solution based on a representation model which was more flexible in that it abstracted away purely graphical concerns. Those concerns include actual design for representation icons, how the design reacts to representation variations within the icons, possible interactions with an icon, and synchronization between the graphical representation and the graphical model. In this paper, we show how to solve those four last points using the SVG open standard for vector graphics. We propose to define representation icons by SVG templates complemented by layout constraints, a predefined and extensible library of possible user interactions using DOM, and a specific approach based on events to synchronize the graphical representation with the graphical model. Thus, our solution solves the concrete realization of an modeling environment cumulating advantages of a clear separation between abstract and concrete syntaxes at the modeling level, while benefiting from the expertise of the vector graphics community.

Keywords: MDE, MDA, Language Engineering, Graphical Concrete Syntax, XML, SVG, DOM.

1 Introduction an Related Works

Model-based techniques to software-intensive system engineering, such as Model Driven Engineering (MDE) [1], place models at the cornerstone of development activities. In parallel, long held research showed advantages of Domain Specific Languages over general purpose languages, provided those languages are properly supported and able to interoperate [2]. Of course, this DSL approach also applies for modeling languages [3]. As a consequence, because of the multiplicity of domains

and levels of abstraction implied even in a single software-intensive development project, there is a need for a large number of well-supported modeling languages. Thus, much is to be awaited from comprehensive and ergonomic techniques to modeling-language engineering.

A (modeling) language is properly defined by an abstract syntax, semantics, and a set of concrete syntaxes [4]. Metamodeling is a convincing technique to capture the abstract syntax of a language in which a model (so called a metamodel) states the vocabulary and the taxonomy of a language. Thanks to these metamodels, automated tools make possible to manipulate and exchange conforming models, such as MDR [5]. Capturing semantics is still a research issue, even though solutions for support were already proposed, for example in [6]. Concrete syntaxes may be either textual or graphical, but are usually a mix of both. As an example, we proposed in [7] a domain specific language as a mean to support textual editing and representation of models.

Solutions like GEF [8], Topcased [9] or MetaEdit [3] apply the same approach to support graphical concrete syntaxes for modeling languages: a domain specific language makes possible to describe how a modeling language is to be graphically represented. Automatic tools turn this specification into a complete graphical editing environment. Approaches such as AToM³ [10] or Tiger [11] are similar, even though their DSLs are given graphical (or hybrid) concrete syntaxes, and better apply lessons learned in the visual language community (e.g. by permitting a precise definition of user interactions). A problem with these approaches is that they restrict the range of possible concrete syntaxes for a given metamodel since structure of abstract syntax constraints structure of concrete syntax (with the notable exception of AToM³). They also need to provide a specific language for graphical depicting of icons.

Another approach is to describe the mapping to a representation language, as applied in [12], by a bidirectional model transformation. A problem with this approach is that model transformation languages are not as well suited as a DSL (as those presented above) for expressing graphical concrete syntaxes.

A last kind of approach we presented in [13] makes use of a concrete syntax graph which is synchronized with the abstract syntax graph (i.e. the model) while it is edited, (following in this the philosophy of AToM³). Constraints are designed to prescribe the synchronization schemes, leaving the possibility to let a (verifiable) model transformation or a constraint solver realizing the actual synchronization. An important advantage is that abstract and concrete syntaxes are completely decoupled thus encouraging reusability of concrete syntaxes and avoiding pollution of the abstract syntax by concrete syntax concerns. Moreover, the approach follows the results of the visual language community (the interested reader may refer to a synthesis in [14]) to modeling languages. However, an important drawback is that the graphical description of icons is left unclear.

In this paper, extended from [15], we complement the latter approach by formally defining icons, trying to reuse best practices in the field of vector graphics. We propose to port to the metamodeling technological space the open standard Scalable Vector Graphics (SVG) [16], for clearly defining icons involved in the concrete syntax of modeling languages. An advantage is that the SVG standard can formally define 2D graphics. Moreover, designers who use SVG to represent icons of a language do not need to be (meta-)model specialists.

To specify a graphical concrete syntaxes, one has to state what are the *icons* of the representable concepts of the abstract syntax, what are the *variation points* and their synchronization with the model (e.g. an editable text to represent a name feature), and how the icons *react to variations*. Moreover, in order for the specification to be turned into a graphical editing environment, language engineers need to specify possible *user interactions*, e.g. that an icon can freely be moved on the diagramming scene, or that a path can be added an intermediate point.

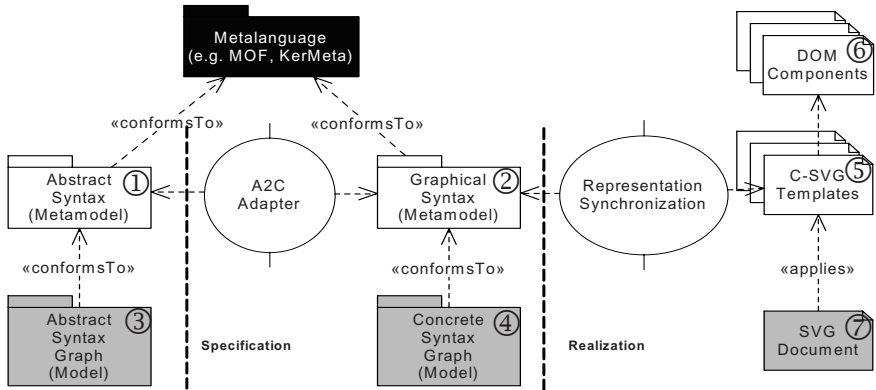


Fig. 1. General Architecture

Figure 1 depicts the overall process. As said above, approach described in [13] *specify* the concrete syntax of a modeling language by formalizing the structure of the concrete syntax graph under the form of a metamodel ②. The concrete syntax graph ④ is kept synchronized with the model ③ as specified with constraints. Moreover, additional constraints fix the spatial relationship between representation icons. The approach presented in this paper complements the specification by *realizing* the graphical representation. Icons are described by SVG templates ⑤. Constraints have long proven their ability to handle variability in graphical environments [17]. C-SVG [18] is an environment for supporting constraints in SVG that we propose as a mean to handle variability within icons. SVG templates should thus be complemented with constraints that can be expressed in the C-SVG language. Since SVG is an XML dialect, we propose an extensible set of predefined user interactions using the DOM API [19] to manipulate XML documents at runtime ⑥. We propose a lightweight mean for synchronizing representation with the concrete syntax graph based on events, in order to render variation points. Finally, the modeling environment is an interactive SVG document ⑦ (the diagram) dynamically showed in an SVG renderer.

The rest of the document is organized as follows. Section 2 presents an example for a modeling language and its specification following the approach we presented in [13] ①②. Sections 3, 4, and 5 detail the approach along with the same example by presenting icon definition (including reactions to variations) ⑤, user interactions ⑥, and relation with the concrete syntax graph (further called the graphical model) ④, respectively. Section 6 end the paper with concluding remarks.

2 Specifying Concrete Syntax for Statecharts

In this section, we detail an example for a modeling language. The abstract syntax of the language is specified by a metamodel, and a graphical concrete syntax is specified as presented in [13].

Statecharts are described in [20]. We show here a metamodel to state its abstract syntax (see figure 1 ①). For sake of simplicity and readability, we will restrict ourselves to a simplified subset of these concepts, as shown by figure 2. State vertices might be connected by transitions. A transition has exactly one source vertex and one target vertex. A vertex is either a pseudo state (initial state, choice,...) or a state, which is in turn either a composite state (i.e. containing other vertices and transitions), a simple state, or a final state. Transitions are triggered by events. A state machine is given by its top state.

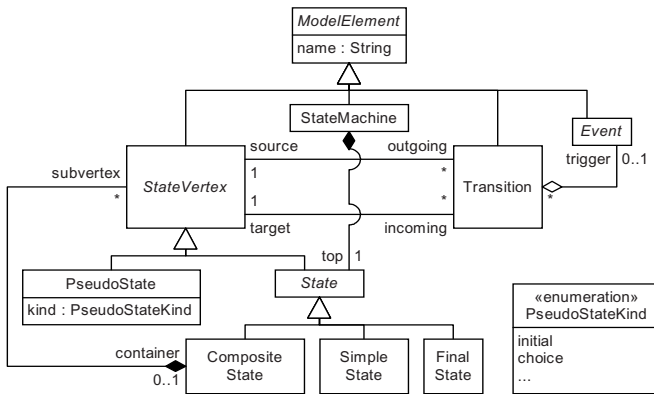


Fig. 2. The Simplified Statechart Metamodel

The concepts of the statechart language can be represented by the symbols shown in figure 4. There is no need to define the `StateMachine` symbol since a state machine cannot be represented. “event”, “name”, and “contents” parts of icons are variation points of the icons: the “event” text should be replaced (if necessary) by the name of the event that triggers the transition, “name” text should be replaced by the name of the represented state, and “content” text points the placeholder for sub-states of the represented composite state. The icons can be freely moved and resized in the diagram, except icons for transitions that have to connect representations for the source and target states of the transition.

Figure 3 shows an excerpt of the specification for the graphical concrete syntax of the statechart language informally described above. The figure is separated in three parts. The part in the left recalls the metamodel. The part in the right defines the graphical elements as a metamodel (see figure 1 ②) by decomposing the graphical icons in different elements. Each graphical element extends the `GraphicalElement` abstract class, which holds relations exposing spatial relationships. As an

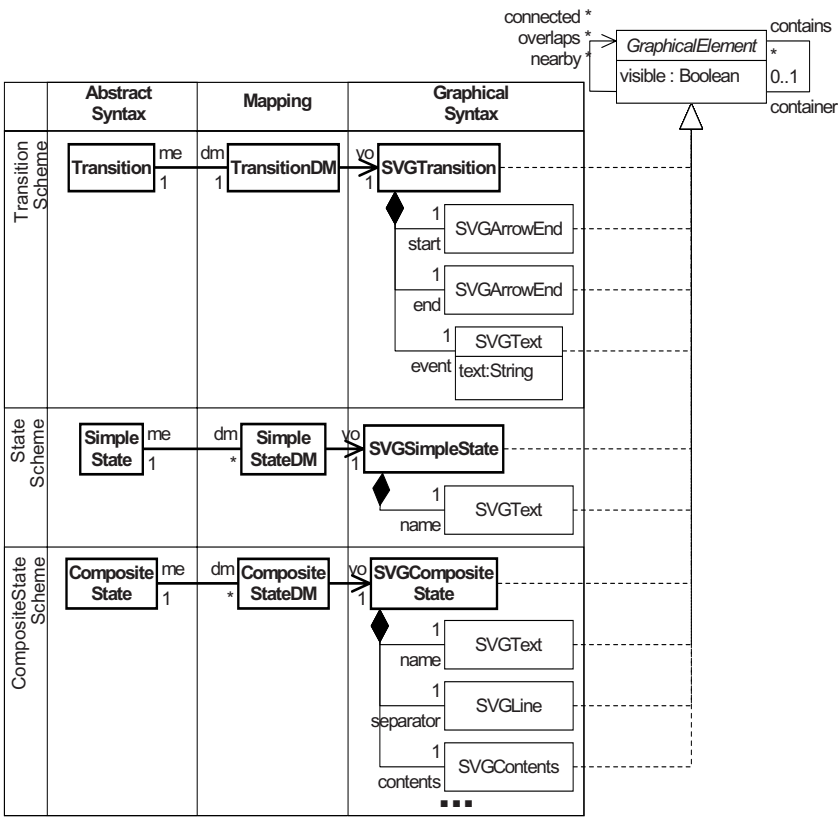


Fig. 3. Excerpt of the Statechart Graphical Concrete Syntax Specification

Transition	SimpleState	Composite State	FinalState	PseudoState (initial)	PseudoState (choice)

Fig. 4. Symbols for the Statechart Concepts

example, the icon for CompositeState is decomposed into a text, a line, and a placeholder for contained states. Possible variations in the elements of graphical objects state the possible variations in the icons (e.g. the value of the text attribute in SVGText). The mapping between abstract syntax and graphical syntax is described using mapping classes as shown in the middle part of figure 3. Those classes are

```

context TransitionDM inv:
  if self.me.trigger->isEmpty()
  then self.vo.event.text.size() = 0
  else self.vo.event.text = self.me.trigger.name
endif

```

Fig. 5. Synchronization Constraint: Text shown on Transitions is Name of Triggering Event

connected to representable classes of the metamodel to graphical elements. Constraints, that can be written in OCL [21], can make more precise synchronization (as exemplified in figure 5), and fix spatial relationships between graphical elements (as exemplified in figure 6).

```

context CompositeStateDM
inv: self.me.subvertex->includesAll(
  State.allInstances().dm
  ->select(sdm|self.vo.contains(sdm.vo)).me)

```

Fig. 6. Spatial Relationship Constraint: Containment of Composite States

A major problem with that approach is that concrete representation of graphical elements and its evolution (including information held by the `GraphicalElement` class) is left unspecified. In the rest of the paper, we propose an approach to overcome this impediment.

3 SVG Templates

In this section, we detail the process of defining SVG template icons of a graphical modeling language, with the ability to react to variation.

Principle of the approach is the following: a diagram is an *SVG document* (see figure 1 ⑦) in which a system engineer may freely add new predefined SVG elements as copied from *SVG templates* (see figure 1 ⑤). Each one of these SVG templates corresponds to a *main graphical element* (see figure 3, right part) i.e. a graphical class that has a connection to a mapping class. In the example of figure 3, main graphical elements are `SVGTransition`, `SVGSimpleState`, and `SVGCompositeState`. Composed graphical elements should be described in the template of their topmost container: in the example, a section of the SVG template for `SVGSimpleState` must describe the name part. Note that it is possible to synchronize the representation directly with the model (see figure 1 ③), but in this case, structure of the concrete syntax is forced to follow structure of the abstract syntax.

When a system engineer decides to add a new element to his/her model, say a `SimpleState`, a copy of the SVG template for `SVGSimpleState` is integrated into the SVG document (see figure 1 ⑦). In the meantime, an `SVGSimpleState` and an `SVGText` graphical objects are created, and a relation between the template

copy (i.e. the template instance) and the `SVGSimpleState` graphical object is maintained. According to specification described in section 2, the creation of an `SVGSimpleState` graphical object should trigger the creation of a `SimpleState-EDM` manager. Finally an associated `SimpleState` object, together with a synchronization between the value of the name slot of the `State` object and the value of the text slot of the `SVGText` graphical object should be created, as described in section 2. Relation between the template instance and the graphical object (in a model repository) will be further discussed in section 5, while possible interactions with the template instances in the diagramming scene will be described in section 4.

Figure 7 exemplifies this template-based approach. Main display classes, which are emboldened on the figure, have a corresponding SVG template, and each one of contained display classes has an SVG counterpart in the template. As an example, a start section appears in the SVG template for `SVGTransition`, which corresponds to the contained `start` `SVGArrowEnd` display object. Note that the SVG section for the `end` display object is different, even though it corresponds to the same `SVGArrowEnd` display class. When the system engineer decides to place a new transition in the diagram (i.e. the SVG scene), any `$$` occurrence in the SVG template is replaced by an identifier specific to the template instance in the SVG diagram so that the various SVG elements in the scene can be identified as part of a specific template instance.

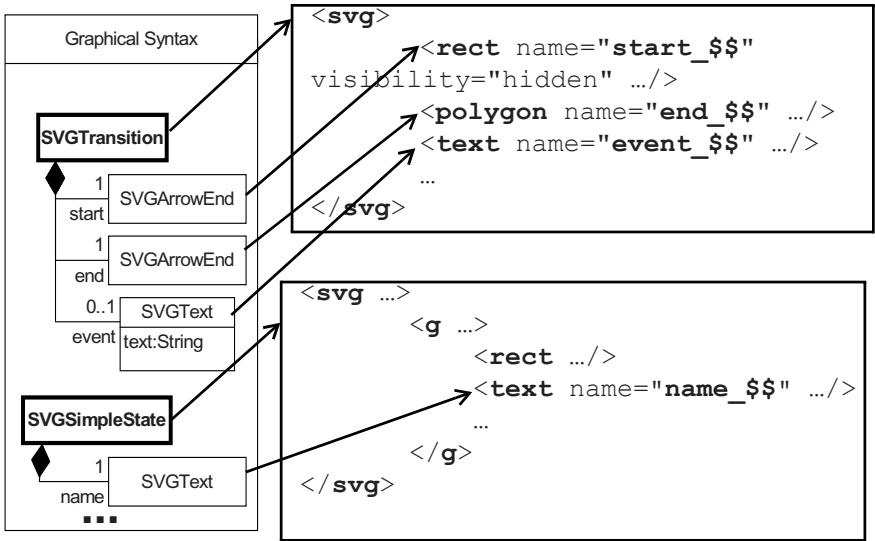


Fig. 7. SVG Templates for Statecharts

As explained before, template instances are subject to variations according to user interactions. In the example of simple states, if name is changed, the containing rectangle needs to grow accordingly. This means that template instances have some dynamic behavior, and may need to be reorganized. Templates thus need to specify a

layout mechanism to state how an automatic reorganization may happen. Constraints have long proven to be a comprehensive mean to specify such layout mechanism [17], and we decided to rely on C-SVG [18] that is specialized in constraining SVG documents. In the simple state template, the growing-name problem is solved as shown in figure 8.

```
<svg ...> <g ...>
  <c:variable name="w_$$" value=
    "c:max(c:width(c:bbox(id('name_$$')))) + 20, 150)" />
  <rect ...>
    <c:constraint attributeName="width" value="$w_$$" />
  </rect>
  <text id="name_$$" ...>Simple State Name</text>
</g> </svg>
```

Fig. 8. SimpleState SVG Template: CSVG Constraint to Handle Text Growth

First, a variable named `w_$$` tracks an arithmetic expression in which the computed width of the `name_$$` text plays a central role. A constraint, placed in the rectangle, forces that rectangle to be as wide as the value of the `w_$$` variable. Automatic tools can place listeners in the SVG documents so that the C-SVG constraint keep satisfied. If contents of `name_$$` is changed, the computed value for `w_$$` is updated, which triggers a new computation for the rectangle's width.

4 User Interactions

SVG documents are not primarily intended to interact, e.g. by mean of mouse or keyboard, as it is necessary for modeling a system. In this section, we show how to enable user interactions in template instances.

The principle we propose is the following: SVG is an XML dialect, and an SVG document is an XML tree. DOM is an API that programming languages such as Java use to read and alter XML trees [19]. Thus, a program making use of the DOM API may alter an SVG document. We will further call such kind of program a *DOM component*. We chose an architecture in which user actions (e.g. mouse moved, mouse clicked, or key hit) trigger execution of some DOM components, which may alter the SVG document that represents the diagram scene. Those DOM components may behave differently depending on the context (e.g. what are the selected elements, what are the elements under the mouse). It is important for the SVG graphical renderer to dynamically adapt the shown diagram to alterations of the SVG document, as it is the case for the Apache Batik toolset [22].

The user interactions we propose may be compared to graph grammars to enable user interactions. The difference is that they transform XML trees rather than graphs. An advantage is that the SVG language (or more precisely the DOM interface) automatically brings genericity so that an interaction only poorly rely on the transformed elements.

Possible user interactions with representation icons (or parts of them) are recurrent. For instance, behaviors like move, connect, or resize, apply in a wide range of contexts, regardless they should impact the model (see figure 1 ③) or not. An important point is to have the possibility to choose exactly where those interactions are enabled. That is why we developed a library of standard interactions independent from the context, for them not to be implemented again and again depending on the SVG kind of element that has to expose the behavior. We developed those interactions as parametrized DOM components with the help of the DoPidom framework [23]; as a consequence, the result of those interactions can only alter the SVG document that represent the diagram. The interactions are triggered by a controller that treats mouse and keyboard events. Parameters are stored in the SVG diagram. Note that some of these interactions are pure queries and do not alter the diagram; these query interactions are intended to be used by other interactions. We list below some of the interactions we implemented (the interested reader may find a more complete list in [24, 25]). Of course, if one needs an additional interaction, it is possible to make the list evolve thanks to DoPidom.

Interactions dedicated to position:

- `Locatable`: finds the coordinates of the holding node in the scene in terms of position, width and height.
- `BorderFindable`: finds a list of points in the scene drawing the outline of the holding SVG node.

Interactions dedicated to movement:

- `Translatable`: moves an SVG node according to given a vector.
- `BorderSlidable`: makes the holding SVG node affix to the outline of an SVG element exposing the `BorderFindable` interaction, as can be found thanks to an `attachedComponent` parameter.
- `Resizable`: emphasizes the holding SVG node of a given factor.

Interactions dedicated to text editing:

- `CharacterHitable`: places a caret at a given index of a `Text` SVG node.
- `CharacterInsertable` and `CharacterDeletable`: inserts/removes a given character at a given position of a `Text` SVG node.

Interactions dedicated to scene management:

- `Selectable`: places the holding SVG node as part of the selection of the scene.

Interactions dedicated to connection-based languages (see [14]):

- `Link`: such an SVG node will be holder for a connection and should not be rendered in the scene. A `Link` interaction makes the connection between SVG nodes participating in a connection by declaring what is the SVG node for the path (by mean of a `curvedLine` parameter) and what are the elements represented at the ends of the connection (by mean of `start` and `end` parameters).
- `CurvedLine`: such an SVG node may be placed as a connection for a link. They overload a possible `Selectable` behavior by creating line handler in order to change its route (i.e. its intermediate points). Original link is registered by mean of a `parentLink` parameter.

- **Arrow:** such an SVG node can be placed at the start or the end of a link. A `position` parameter states whether the SVG node is at the beginning or at the end of the connection.

Interactions dedicated to containment:

- **Container:** such an SVG node is able to contain SVG nodes declaring the `Containable` interaction. Contained elements are placed in a `contents` parameter. Interaction changes an eventual `Translatable` behavior by making the contained nodes follow the same movement. This notion is independent from the notion of SVG group.
- **Contained:** such an SVG node may be part of the contents of an SVG node declaring the `Container` interaction. Container is placed in a `container` parameter. Interaction changes the `Translatable` behavior by attaching or detaching the SVG node from its container according to its target position.

```
<svg ...>
  <g dpi:component="Translatable, Contained,..." ...>
    <rect id="border_$$" dpi:component="BorderFindable,
..."/>
    <text name="name" dpi:component="CharacterHitable,
CharacterInstertable, CharacterDeletable, ..." .../>
    ...
  </g>
</svg>
```

Fig. 9. Simplestate SVG Template: Declaring DOM Components

An example, shown in figure 9, is the template definition for `SVGSimpleState`. In this code snippet, the first element is an SVG group that declares the `Contained` and `Translatable` interactions. These declarations makes thus possible both to move freely the representation for a simple state template instance as a whole, and to make it containable by another SVG element declaring the `Container` interaction (as it should be the case for the composite state template). The group contains a rectangle that is responsible for being the outline of the state in that it declares the `BorderFindable` interaction; as such, another SVG node declaring the `BorderSlidable` interaction can be affixed to the rectangle (e.g. an end in the representation for a transition). The group also contains a text that is the placeholder for the name of the state. That is why this text must be editable and declares the `CharacterHitable`, `CharacterInsertable` and `CharacterDeletable` interfaces.

5 Relationship with the Model

Our choice to develop reusable behaviors, which only act on the SVG representation, prevents from directly updating the information of the graphical elements (and of the

model as an indirect result). We show here how to complement SVG templates and predefined events for the modeling information to be updated.

To do so, we propose to add listeners that can be declared on the SVG templates to synchronize atomic information in the SVG document with atomic information in the graphical model. We call atomic information a datum that is either a character string, a boolean, an integer or a real. This information may be processed in the document (e.g. using a C-SVG constraint or an XSL transformation) to be properly represented.

Before realizing the synchronization, the relationship between the SVG representation and the graphical model needs to be established. The solution we propose is to maintain variables in the SVG document for each SVG node that corresponds to a graphical object. Those variables are to be filled at template instantiation time using an action language. Moreover, actions may have to be performed in case the template instance is removed from the scene. Variable declarations, initialization and removal actions can be specified in the SVG templates, as exemplified in figure 10.

```
<svg onCreate="{Java|
  t = model.getSVGText().createSVTText();
  s = model.getSVGSimpleState().createSVGSimpleState();
  s.setName(t);}"
  onDelete="{Java|
    s.refDelete();}" ...>
<g id="$ $" ...>
  <m:variable name="self" value="$s" />
  <rect id="border_$ $" .../>
  ...
  <text id="name_$ $" ...>
    <m:variable name="self" value="$t" />
    newState</text>
</g>
</svg>
```

Fig. 10. SimpleState SVG Template: Variables

In the figure, the SVG template for simple states is complemented by a creation action written in the java language (using the JMI API [26]) where a variable `model` plays the role of the model repository. In the action, two graphical objects (an `SVGSimpleState` and an `SVGText` further referred to `s` and `t`, respectively) are instantiated and associated as prescribed in figure 3. A deletion script states that the `s` object should be deleted when suppressing the template instance (note that the `t` object does not need to be suppressed explicitly because of the composition relationship between the `SVGSimpleState` and `SVGText` elements). Moreover, new variable XML nodes are added to the SVG template to handle the local dependencies of the representation to the graphical model, as suggested by arrows in figure 7. At template instantiation time, the action is executed, and the local variables are set to the references resulting from the execution of the action. In the simple state example, those variables are either initialized to `s` (for the group and the rectangle) or to `t` (for

the text) as declared by the values of the `variable` XML nodes. Note that an SVG node may declare different variables.

Once the relationship between the SVG representation and the graphical model is established, it is possible to synchronize atomic information between them. To do so, we introduce a new update XML node. The `location` XML attribute of the update node states, with an XPath expression [27], where does the atomic information to synchronize appears in the document. Two more XML attributes of the update XML node state what are the graphical object and the slot to observe.

```
<svg ...>
<g id="$ $" ...>
  <rect id="border_$ $" .../>
  ...
  <text id="name_$ $" value="newState"
    dpi:component="CharacterIntertable, ..." ...>
    <m:variable name="self" value="$t" />
    <m:variable name="displayed" value="newState" />
    <c:tval
      value="..../variable[@name='displayed'] [1]/@value" />
    <m:updater var_source="$t" slot="text"
      location="..../variable[@name='displayed'] [1]/@value"
    />
  </text>
</g>
</svg>
```

Fig. 11. SimmpleState SVG Template: Updater

Figure 11 shows such a declaration of synchronization in the case of the simple state template: the `text` slot of the `SVGText` graphical object `t` has to be rendered in the (editable) `text` SVG node. To do so, we introduce a new variable `displayed` which will be rendered by the `text` SVG node thanks to a `tval` C-SVG constraint. An updater synchronizes the value of the `display` variable with the information in the model. When the text is edited in the representation, the C-SVG constraint changes the value of the `display` variable, which triggers propagation of the new text to the corresponding graphical object. When the text changes in the model, the updater is notified and changes the value for the variable, which is then rendered according to the C-SVG constraint.

The last information that has to be reflected both in the graphical objects and in the SVG diagram is the information held by the `GraphicalElement` class, which are the spatial relationships and the visibility. The information is automatically updated by a double synchronization mechanism implemented by observers. On one hand, observers track actions performed by the interactions. On the other hand, other observers track changes in graphical objects as stored in the `isVisible`,

container, contained, nearby, overlaps, and connected features. `isVisible` (as found in the graphical object maintained by the `self` variable of the representation node) is synchronized depending on the `display SVG` attribute. The other slots are updated during the execution of interactions e.g. `Container`, `Contained`, or `BorderSlidable`, i.e. any interaction able to change spatial relationship. Note that all these interactions can be vetoed (in case a constraint fails at model level - see section 2) or forced (in case the graphical model changes).

6 Conclusion

We proposed here a technique for concretely representing a model in a diagram, once the abstract syntax (i.e. the metamodel) is known. We took advantage from the widely accepted SVG standard to specify vector graphics. The approach is intended to be used in conjunction with the approach presented in [13], which clearly separates modeling data from graphical data, and which leads the concrete realization, following the example of a component realizing its specification interfaces.

When combining those two approaches, the steps to specify a graphical concrete syntax are thus the following:

1. create a mapping class for each model element of the metamodel that needs to be represented,
2. create for each mapping class one or more graphical class and its different parts reflecting the structure and the variability of the icon,
3. write the constraints on the mapping classes for abstract/concrete synchronization, representation alternatives and inter-icons relationship (e.g. spatial relationship),
4. write the SVG template for each root graphical class,
5. specify allowed interactions from a reusable and generic library acting on the SVG representation,
6. complement the SVG templates with graphical constraints (e.g. using C-SVG) to handle intra-relationships between the various SVG elements composing the templates,
7. create variables and initialization/deletion scripts to establish relation of the representation to the graphical model,
8. declare updaters so that the representation and the graphical model keep synchronized.

The approach we propose is certainly more verbose than other existing approaches (as GMF or GME). However, it manages a broader range of graphical concrete syntaxes. For example, the approach is not limited to connection-based languages thanks to its explicit management of spatial relationships. Moreover, we rely on one hand on metamodeling techniques (for the specification part) which is properly mastered by modeling language engineers, and on the other hand on SVG which is properly mastered by graphical designers. Finally, the modeling language engineer only needs a

few knowledge about SVG to place action scripts, variable and updaters in the SVG templates. As such, if other approaches seem more adapted to prototype a graphical language, we believe that our approach is more adapted to realize the graphical concrete syntax of a modeling language.

Compared to [15], we simplified the process of synchronizing the representation with the graphical model. Indeed, in [15], interactions were explicitly sending events that had to be answered by dedicated action scripts as parameters. Here, representation and graphical model are more tightly coupled thus avoiding such mechanisms. Moreover, those event action scripts were over-specifying synchronization rules between the model and the graphical model, thus dramatically limiting the interest of the graphical model and the implied reusability.

One of the main drawback of our approach is that the specification of the synchronization rules between the model and the graphical model is done using constraints. Thus, one need either a constraint solver (which are usually slow) or an additional bidirectional and incremental model transformation that realizes the constraints, which needs to be proved. Moreover, there is redundancy of information between the model and the graphical model. We plan to change this synchronization specification by making the graphical model a view on the model, following the example of the concept of view in databases.

A prototype implementation can be found in [28]. More insight about interactions is given in [24] and about variables and action scripts in [25].

References

- [1] Kent, S.: Model Driven Engineering. In: Butler, M., Petre, L., Sere, K. (eds.) IFM 2002. LNCS, vol. 2335, pp. 286–298. Springer, Heidelberg (2002)
- [2] Iivari, J.: Why Are Case Tools Not Used? *Commun. ACM* 39(10), 94–103 (1996)
- [3] Pohjonen, R.: Boosting Embedded Systems Development with Domain-Specific Modeling. *RTC Magazine*, 57–61 (April 2003)
- [4] Harel, D., Rumpe, B.: Meaningful Modeling: What's the Semantics of "Semantics"? *Computer* 37(10), 64–72 (2004)
- [5] Sun Microsystems: Metadata repository (MDR) (December 2005)
- [6] Scheidgen, M., Fischer, J.: Human comprehensible and machine processable specifications of operational semantics. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA. LNCS, vol. 4530, pp. 157–171. Springer, Heidelberg (2007)
- [7] Muller, P.A., Fleurey, F., Fondement, F., Hassenforder, M., Schneckenburger, R., Gérard, S., Jézéquel, J.M.: Model-Driven Analysis and Synthesis of Concrete Syntax. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) *MoDELS 2006*. LNCS, vol. 4199, pp. 98–110. Springer, Heidelberg (2006)
- [8] Eclipse Consortium: Eclipse Graphical Editing Framework (GEF), <http://www.eclipse.org/gef>
- [9] Vernadat, F., Percebois, C., Farail, P., Vingerhoeds, R., Rossignol, A., Talpin, J.P., Chemouil, D.: The TOPCASED Project - A Toolkit in OPEN-source for Critical Applications and SysEm Development. In: *Data Systems In Aerospace (DASIA)*, Berlin, Germany, European Space Agency (ESA Publications) (May 2006), <http://www.esa.int/publications>

- [10] Guerra, E., de Lara, J.: Event-driven grammars: relating abstract and concrete levels of visual languages. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 317–347. Springer, Heidelberg (2004)
- [11] Ehrig, K., Ermel, C., Hänsen, S., Taentzer, G.: Generation of visual editors as eclipse plug-ins. In: Redmiles, D.F., Ellman, T., Zisman, A. (eds.) ASE, pp. 134–143. ACM, New York (2005)
- [12] Clark, T., Evans, A., Sammut, P., Willans, J.: Applied Metamodelling: A Foundation for Language-Driven Development (2005)
- [13] Fondement, F., Baar, T.: Making Metamodels Aware of Concrete Syntax. In: Hartman, A., Kreische, D. (eds.) ECMDA 2005. LNCS, vol. 3748, pp. 190–204. Springer, Heidelberg (2005)
- [14] Costagliola, G., Lucia, A.D., Orefice, S., Polese, G.: A Classification Framework to Support the Design of Visual Languages. *J. Vis. Lang. Comput.* 13(6), 573–600 (2002)
- [15] Fondement, F.: Concrete syntax definition for modeling languages. PhD thesis, École Polytechnique Fédérale de Lausanne (EPFL) (2007)
- [16] Jackson, D., Northway, C.: Scalable Vector Graphics (SVG) Full 1.2 specification. World Wide Web Consortium, Working Draft WD-SVG12-20050413 (April 2005)
- [17] Borning, A., Marriott, K., Stuckey, P.J., Xiao, Y.: Solving Linear Arithmetic Constraints for User Interface Applications. In: ACM Symposium on User Interface Software and Technology, pp. 87–96 (1997)
- [18] McCormack, C.L., Marriott, K., Meyer, B.: Constraint SVG. In: WWW Alt. 2004: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters, pp. 310–311. ACM Press, New York (2004)
- [19] Hors, A.L., Hégaret, P.L., Wood, L., Nicol, G., Robie, J., Champion, M., Byrne, S.: Document Object Model (DOM) level 3 core specification. World Wide Web Consortium (April 2004)
- [20] Harel, D.: Statecharts: A Visual Formulation for Complex Systems. *Science of Computer Programming* 8(3), 231–274 (1987)
- [21] Adaptive Ltd., Boldsoft, France Telecom, International Business Machines Corporation, IONA Technologies, Object Management Group: Object Constraint Language specification, v2.0. OMG Document formal/06-05-01 (May 2006)
- [22] Apache Foundation - XML Graphics Project: Batik SVG toolkit, <http://xmlgraphics.apache.org/batik/>
- [23] Beaudoux, O.: DoPIDom: une approche de l'interaction et de la collaboration centrée sur les documents. In: IHM 2006: Proceedings of the 18th international conference on Association Francophone d'Interaction Homme-Machine, pp. 19–26. ACM Press, New York (2006)
- [24] Hong, F.: Provide behaviour to XML-SVG. Bachelor Semester Project, École Polytechnique Fédérale de Lausanne (EPFL) (2005)
- [25] Rohrer, F., Helg, F.: Synchronization between display objects and representation templates in graphical language construction. Bachelor Semester Project, École Polytechnique Fédérale de Lausanne (EPFL) (2006)
- [26] Java Community Process: Java(TM) Metadata Interface API specification 1.0 final release. JSR-000040 (June 2002)
- [27] Clark, J., De Rose, S.: XML path language (XPath). World Wide Web Consortium (November 1999)
- [28] Fondement, F.: SVG-based modeling tools (2007), <http://fondement.free.fr/lgl/projects/probxs/>

Semantics Preservation of Sequence Diagram Aspects

Jon Oldevik^{1,2} and Øystein Haugen^{2,1}

¹ University of Oslo, Department of Informatics

² SINTEF Information and Communication Technology, Oslo, Norway
jonold@ifi.uio.no, oystein.haugen@sintef.no

Abstract. Although some techniques for aspect oriented modelling focus on semantic-based composition, there has been a general lack of focus on how to preserve semantics. We address semantics preservation in the context of sequence diagram aspects. We define semantics preservation based on two properties: monotonicity of aspect composition with respect to refinement, and preservation of events in the composed result. We analyse and compare existing sequence diagram aspect techniques with respect to semantics preservation and find that both syntactic based and semantic based aspect weaving techniques have serious challenges to meet semantics preservation.

1 Introduction

Sequence diagrams are frequently used as a formalism in system specification and are useful for reasoning about system behaviour at different detailing levels, ranging from requirements specification towards detailed design. The mechanisms available in UML sequence diagrams allow for specifying complex control flow behaviour between interacting system roles/objects. Formal refinement techniques, such as STAIRS [1], provide the means for reasoning about system consistency during refinement of sequence diagrams.

Although UML sequence diagrams provide compositional mechanisms that allow concerns to be separately specified and integrated through interaction uses and decompositions, these are limited in that they require these separate concerns to be integral parts of the same system; it is not straightforward to describe concerns as separate systems with subsequent composition into one coherent system. This has been addressed by the aspect-oriented community for some time, initially at a programming level [2,3,4,5]. As model-driven development and generative programming becomes more widespread, similar ideas have been dispersed to the modelling community [6,7,8,9], but with no converging towards a standard way of supporting this.

Semantics preservation of aspect composition is a much neglected issue of aspect-orientation in general, and aspect-oriented modelling specifically. Establishing awareness of semantics preservation is important to reduce risk of design errors and increase consistency of specifications.

We give a definition of semantics preservation for sequence diagram aspect composition and analyse existing sequence diagram aspect techniques with respect to this definition. Our view on semantics preservation relies on formal definitions of sequence diagram refinements from STAIRS [1,10,11]. We define two properties that contribute

to semantics preservation: *monotonicity of aspect composition with respect to refinement and preservation of events.*

2 Defining the Concepts of Sequence Diagram Aspects

Aspect-oriented specification and composition techniques provide flexible means for separating and composing cross cutting concerns. Ensuring that using such techniques does not break down consistency of our specifications is an important, albeit much overlooked, concern. By adhering to principles that preserve semantics, greater confidence can be given to our systems.

In this paper, we do not prescribe any specific technique for sequence diagram aspects. Rather, we seek to unfold some characteristics that are fruitful in the context of semantics preservation. A range of different techniques for aspect-oriented specification and composition of sequence diagrams have been described [12,13,14,15,16], but there is currently no standard approach for this. Each approach has its strengths and weaknesses; some are based on syntactical matching and composition, some semantical.

We address semantics of sequence diagrams and make a definition of *semantics preservation* for sequence diagram aspect composition. In this Section, we look at STAIRS semantics for refinement as a basis for our definition. In Section 3, we analyse existing techniques with respect to our semantics preservation definition.

2.1 Sequence Diagram Semantics

In STAIRS [10,11], the semantics of interactions is defined by the set of event traces it represents. This set of traces are all possible sequences of events from a run of the interaction (the system). These event sequences are constrained by the causality and weak sequencing properties of interactions: a send event must occur before its corresponding receive event (causality) and the events on a lifeline have the same (relative) ordering in a trace as on the lifeline (weak sequencing).

A trace can be *positive*, *negative*, or *inconclusive* with respect to an interaction. Negative traces represent disallowed behaviour and are produced by explicitly *negating* or *asserting* fragments in an interaction. Positive traces represent allowed behaviour and are all traces described by the interaction that are not negative. Inconclusive traces are all other traces, i.e. the traces that are not covered by the interaction.

In [10,11], trace semantics are used to define and analyse refinement between interactions. Three refinement relations are defined in STAIRS: supplementing, narrowing, and detailing. Supplementing increases the set of positive traces by moving traces from inconclusive to positive i.e. increases allowed behaviour. Narrowing refinement reduces the set of positive traces making them negative, i.e. reduces allowed behaviour. Detailing is refinement by means of decomposition.

Refinement in STAIRS is transitive: if $A \rightsquigarrow B$ and $B \rightsquigarrow C$, then $A \rightsquigarrow C$ (\rightsquigarrow is the refinement relation). It is also monotonic with respect to the sequence diagram composition operators *alt*, *opt*, *neg*, *seq*, and *par*: if $A \oplus B$ is the composition with one of those operators of interactions A and B , and A' and B' are refinements of A and B , respectively, then $A' \oplus B'$ is a refinement of $A \oplus B$. (Proof can be found in [17].) This

characteristic is valuable to ensure system consistency during system evolution, so that a system can be specified and refined in parts and later composed.

2.2 Defining Semantics Preservation for Sequence Diagram Aspects

Monotonicity of Aspect Composition with Respect to Refinement. In general system engineering dividing and conquering is a valuable strategy for managing complexity. As described in the previous Section, the monotonicity property of refinement provides this for sequence diagrams. This allows individual components to be analysed separately. When composing them, the analysis results will also be valid for the compositions.

We would like similar properties for sequence diagram aspect compositions. Specifically, we want composition of aspects with different sequence diagrams in a refinement relation to conserve the refinement relation.

Given two sequence diagrams I_1 and I_2 , where $I_1 \rightsquigarrow I_2$ (I_2 is a refinement of I_1). Let A be a sequence diagram aspect and $I_1 \oplus A$ the composition of sequence diagram aspect A with I_1 . Now, we would like $I_2 \oplus A$ to be a refinement of $I_1 \oplus A$: $(I_1 \oplus A = IA_1) \wedge (I_2 \oplus A = IA_2) \rightarrow (IA_1 \rightsquigarrow IA_2)$. In other words, we would like sequence diagram aspects to be monotonic with respect to STAIRS refinement.

Fig.1 illustrates the monotonicity property with respect to refinement using an aspect representation similar to that of Klein et al. [18], where pointcut and advice is described in separate sequence diagrams. When applying this aspect on the *base* model, the result will be the *result* model. The *base'* model is a supplementing refinement of *base*. When applying this aspect on *base'*, the result will be the *result'* model, which is a refinement of the *result* model.

Event Preservation. When a sequence diagram aspect is composed with another sequence diagram, the traces of events, and hence the semantics, defined by the original sequence diagram is altered. An intuition of semantics preservation under these circumstances is that the behaviour we could observe in the base model still should be

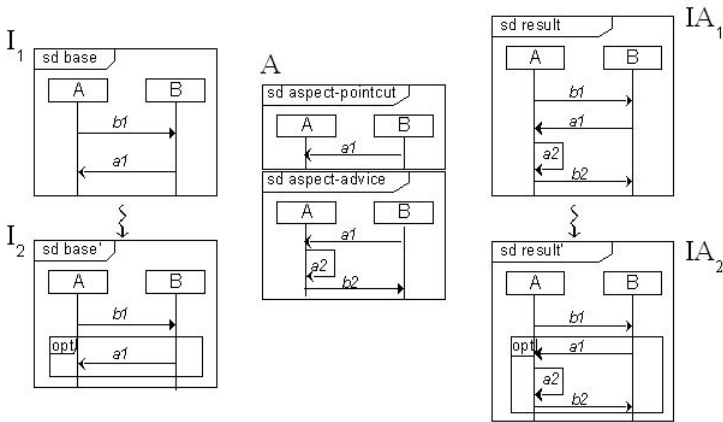


Fig. 1. Monotonicity of Aspect Composition with Respect to Refinement

observable in the composition. The events defined by the base sequence diagram should be present with the same relative order in the composed sequence diagram.

To be more precise, we define a hiding operator that hides events in a composition that originating from one of the composition operands (e.g. the aspect). Let I_1 be a base sequence diagram with traces T_{I_1} , A an aspect with traces T_A , and IA_1 the composition of the aspect with the base model with traces T_{IA_1} . We define the hiding operator $\text{hiding}(T_{IA_1}, T_A) \rightarrow T_{I_2}$ such that:

1. For each trace in the base model ($t \in T_{I_1}$), there exists a trace in the composition ($\text{hiding}(T_{IA_1}, T_{I_1})$) with the same event set.
2. For each trace in the composition ($\text{hiding}(T_{IA_1}, T_{I_1})$), there exists a trace in the base model, which is either equal or contains all the same events.

(1) ensures that the composition cannot be empty, as it must have a representation of all traces of the base. (2) ensures that the traces that do represent a base trace are equivalent and that new traces can be introduced by supplementing refinement.

The implications of *event preservation* is that a composition cannot remove or replace fragments originating from the operands of the composition.

Semantics Preservation. In this paper, we define **semantics preservation** of sequence diagram aspect composition in terms of **monotonicity with respect to refinement** and **event preservation**.

In the following, we will address how syntactic- and semantic-based matching and composition approaches relate to the monotonicity property. Any such approach that does not remove traces will be event preserving.

2.3 Syntactic Matching and Composition

When an end user models the system she/he will be concerned with the concrete syntax provided as well as the semantics of what she/he describes. However, when doing sequence diagram aspect composition, there are difficulties in maintaining a semantically consistent model. We will show that monotonicity with respect to refinement and aspect composition is problematic when working on the syntactical level, and it therefore is problematic to be semantics preserving.

First we look at some examples of sequence diagrams and refinements and how a particular sequence diagram aspect technique (Whittle et al. [15]) provides compositions on these models.

Fig.2 illustrates three models: a base model ($abc1$) and two refinements ($abc1'$ and $abc1''$). The two refinements are supplementing refinements of the base model, as they add positive behaviour. There is one new positive trace in each of $abc1'$ and $abc1''$.

Fig.3 illustrates a sequence diagram aspect (Aspect-A) that we want to bind to the base model in Fig.2 i. It defines the creation of two messages x and y by using the «create» stereotype. The search criteria (pointcut) of the aspect is defined by the non-stereotyped messages ($b1$ and $a1$). So, the aspect looks for the pattern $b1$ followed by $a1$ and creates an x message before and a y message after every match. The result of composing this aspect with $abc1$ in Fig.3 i is shown in Fig.4 i.

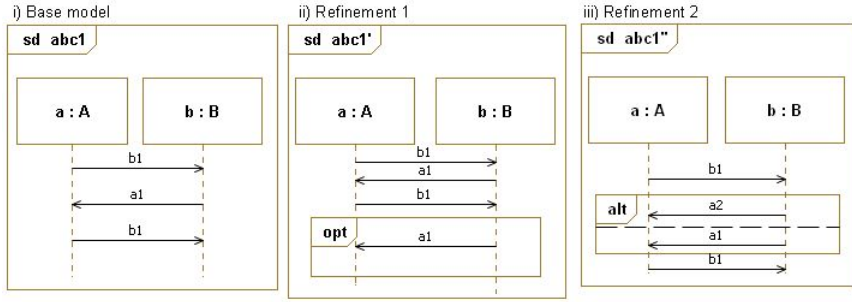


Fig. 2. Base Model abc1 with Two Refinements

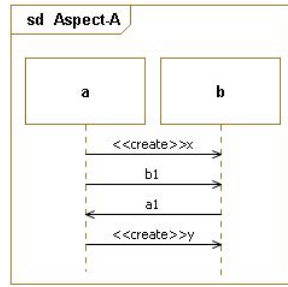


Fig. 3. Sequence Diagram Aspect and Binding Query

Now, we want to investigate the effect of applying the same aspect on the refinements of the base model. The results of these compositions are shown in Fig.4 ii and iii. Refinement 1(*abc1'* in Fig.2) introduced a new optional fragment that leads to a new potential match for the aspect. The approach in [15] works on concrete syntax and hence allows an aspect query (pointcut) to match fragments belonging to different fragment containers. As a result, there will be two matches for our aspect in this refined sequence diagram. The resulting composition (*abc1'-aspectised* in Fig.4 ii) is not a refinement of the base model with the aspect (*abc1'-aspectised*), since not all traces from *abc1'-aspectised* are traces in *abc1'-aspectised*. In fact, none of the traces are present.

If the approach disallows these kinds of matches, i.e. requiring fragments to have the same owner, similar problems occur: the refinement *abc1''* introduced an alternative combined fragment with *a1* in one operand and *a2* in another. When applying the aspect, there will be no matches for the aspect pointcut. The result (*abc1''-aspectised* in Fig.4 iii) will not be a refinement of *abc1'-aspectised*.

Narrowing refinement resulting from introduction of negative fragments in the base model have similar problems with respect to monotonicity.

We conclude that pure syntactical aspect compositions may easily lead to non-monotonicity with respect to refinement. It is possible to avoid this by introducing limitations on what is allowed in a composition. These limitations, however, are so severe that they will greatly diminish the value of sequence diagram aspect compositions.

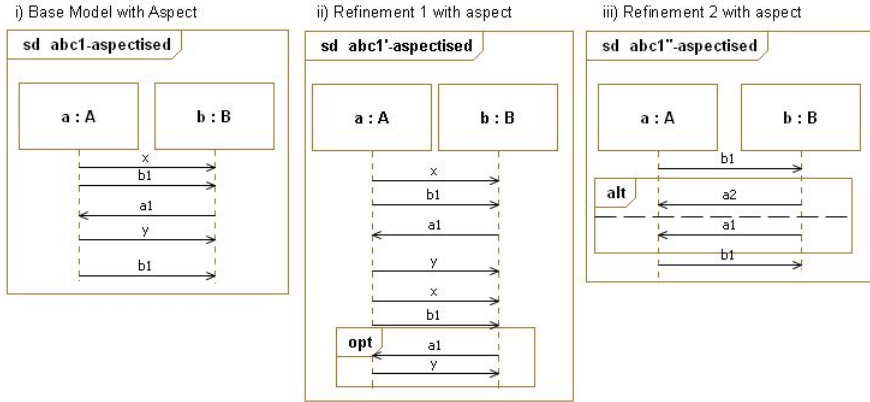


Fig. 4. Base Model and Refinements With Composed Aspect

Due to the problems with pointcuts that match across fragment containers (e.g. inside and outside an optional fragment), as illustrated in Fig. 4 ii and problems in matching several subsequent fragments (due to introduction of e.g. an alternative), as illustrated by Fig. 4 iii, a syntactical approach could have the following restriction to be semantics preserving: *a pointcut can only match single messages or single combined fragments.*

2.4 Semantic Matching and Composition

Matching and composition based on the sequence diagram semantics rather than the concrete syntax should make it simpler to preserve semantics; this seems a somewhat tautologous statement, but it depends on our understanding of semantics preservation. By our definition of semantics preservation, which relies on refinement semantics, it is very likely that a matching- and composition-system based on trace semantics will be semantics preserving. Since a refinement will never remove a trace, but only add new traces to the positive traceset from the inconclusive traceset, or move traces from the positive traceset to the negative traceset, a matching-system will always re-find the traces it found in the refined model. Thus, applying the same aspect on a base model and its refinements will result in models in which the compositions of the refinements are also refinements of the base model composition.

2.5 Refinement of the Aspects

So far, we have addressed monotonicity of base model refinement and aspect composition, where $I_1 \oplus A = IA_1 \wedge I_2 \oplus A = IA_2 \rightarrow IA_1 \rightsquigarrow IA_2$. To generalise this, we should also address refinement of the aspect: $I_1 \oplus A = IA_1 \wedge I_2 \oplus A' = IA_2 \rightarrow IA_1 \rightsquigarrow IA_2$, where aspect A' is a refinement of A ($A \rightsquigarrow A'$).

To consider refinement of an aspect, we need to view it in terms of its two logically distinct parts: its pointcut or query part and its advice. The pointcut of an aspect is closely related to the mechanism of a weaving operation. Any refinement of this may

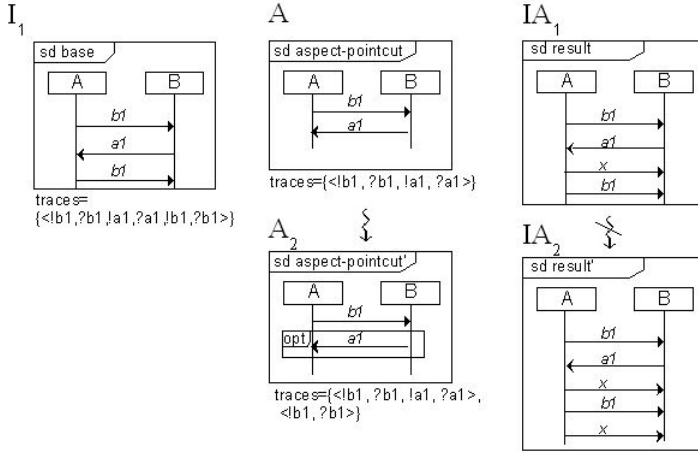


Fig. 5. Refinement of Pointcut with Traces

alter the search criteria and the results. This will easily lead to compositions that are not semantics preserving.

Fig. 5 illustrates the problem with pointcut refinement using a trace-based matching semantics. It shows a simple example where a sequence diagram aspect pointcut (A) will result in a single match in the base model (I_1). The advice used in the composition is from Fig. 6 and is assumed composed after pointcut matches. The pointcut A matches the event sequence $\{<!b1, ?b1, !a1, ?a1>\}$ in the base model. The pointcut refinement (A_2), however, will get two matches in the base: first, the trace $\{<!b1, ?b1, !a1, ?a1>\}$ and then $\{<!b1, ?b1>\}$. The result model IA_2 is *not* a refinement of the result model IA_1 .

We conclude that pointcut refinement easily results in non-monotonicity with respect to aspect composition and refinement. This will be the same for semantical and syntactical approaches.

Therefore we limit our scope to refinement of the advice part of aspects. A refinement of an aspect advice should be any legal sequence diagram refinement according to STAIRS. Fig. 6 shows how a refined advice influences the composition result. We use the pointcut A described in Fig. 5. The advice A_2 is a refinement of the advice A and the corresponding composition IA_2 is a refinement of IA_1 .

Many aspect modelling approaches define special constructs in order to describe an aspect. This makes the semantics of refinement unclear for that aspect. For any such approach, the semantics of refinement must be defined with respect to these special constructs.

Another issue with respect to refinement arises if pointcut and advice parts are mixed in the aspect specification, which is the case in several approaches. In Whittle et al. [15], the pointcut and advice is mixed in the same sequence diagram. It is not given how to refine the aspect without modifying the pointcut part, and specifications of that semantics must be defined to allow this. The approach proposed by Deubler et al. [12] has a similar mixing of pointcut and advice, which displays the same problems with respect

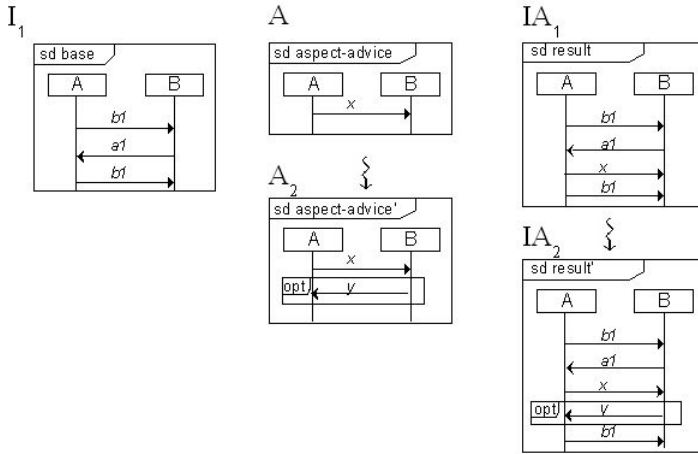


Fig. 6. Refinement of Advice

to advice refinement. We conclude that mixing of pointcut and advice in a sequence diagram makes refinement and hence semantics preservation difficult.

In the next Section, we analyse existing sequence diagram aspect techniques with respect to our definition of semantics preservation.

3 Analysis of Existing Sequence Diagram Aspect Techniques

We give an overview and analysis of different sequence diagram aspect techniques, specifically with the semantics preserving properties in mind. We use the *buyItem* example in Fig.7 to illustrate some of the approaches. The figure shows a base model sequence diagram (*buyItem*) and a modified sequence diagram that is a composition of the base model and a transaction aspect that adds new fragments for transaction handling to our model. The aspect adds a *startTransaction* message before and a *commit/abort alternative* after the existing *sendPayment*.

3.1 Syntactic-Based Approaches

Solberg et al. Solberg et al. [13] have developed a UML-based language for describing aspects for sequence diagrams. The approach relies on a role-based extension to UML [19] where classes, objects, operations, and properties can be marked as templates. They distinguish primary models from aspect models. Primary models are explicitly tagged using specific combined fragment styles called *simpleAspect* and *compositeAspect*, i.e. the joinpoints are explicitly tagged in the primary model, i.e. there is no query mechanism. The tags bind the content fragments to a specific aspect and pass necessary parameters to the aspects (Fig.8).

A *simpleAspect* is always bound only to a single lifeline in the primary model. A *compositeAspect* is more elaborate, covers several template lifelines, and may define

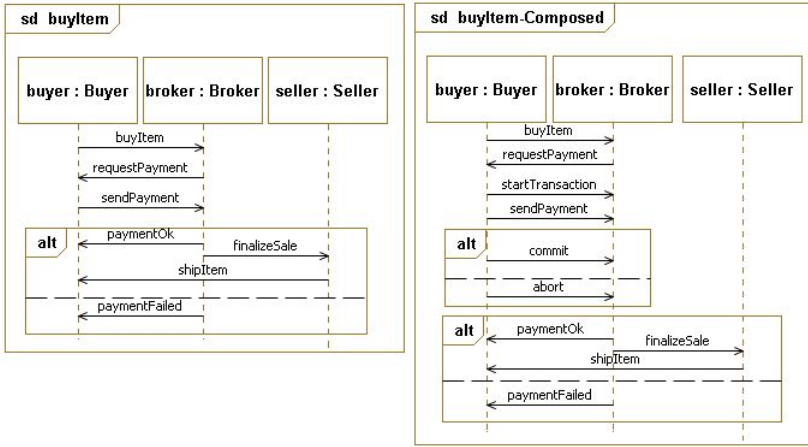


Fig. 7. Sales Example - buyItem and buyItem with Transaction

different kinds of advice fragments, begin, before, body, after, and end, which are modelled as special stereotyped combined fragments. The advice fragments allow messages to be inserted at the beginning and end of the compositeAspect tag, before or after each message within the tag. The body advice specifies a merging of the contents of the compositeAspect fragment in the primary model with the messages in the body advice. The semantics of the body advice requires it to contain one or more template operations. In this approach, the base model knows about the aspect since references to the aspect is placed in the base model (i.e. it is not oblivious). The aspect on the other hand, is oblivious of the base model.

In this approach, there is no query model that can modify the selected joinpoint for an aspect, but an explicit combined fragment that tags the selection in the base model. This implies that the aspect will *always* apply to that designated tag and its contents, whatever refinement is done on the base model. An open issue in this model, however, is what refinement of a model containing an aspect tag means. If this is defined such that the aspect tag fragments never can be moved into new optional/alternative combined fragment, aspect composition in this approach will be monotonic with respect to base model refinement.

Refinement of the aspect itself also needs semantical clarifications for the different constructs (i.e. tags) used in an aspect. If we assume that a refinement is only allowed to make changes inside these tags (e.g. add an optional fragment), the approach is also monotonic with respect to aspect advice refinement.

In this approach, an aspect can only add fragments to the base model. It will as such provide compositions that are event preserving.

Whittle et al. Whittle et al. [15] uses graph transformations to specify and compose UML-based aspects in the MATA tool, a general purpose tool for aspect composition in any language with a well-defined metamodel. A special profile is defined with stereotypes that allow manipulation of language elements to support composition (*create*, *delete*, *context*, *any*). Pointcuts and advice (aspects) are combined in a single MATA

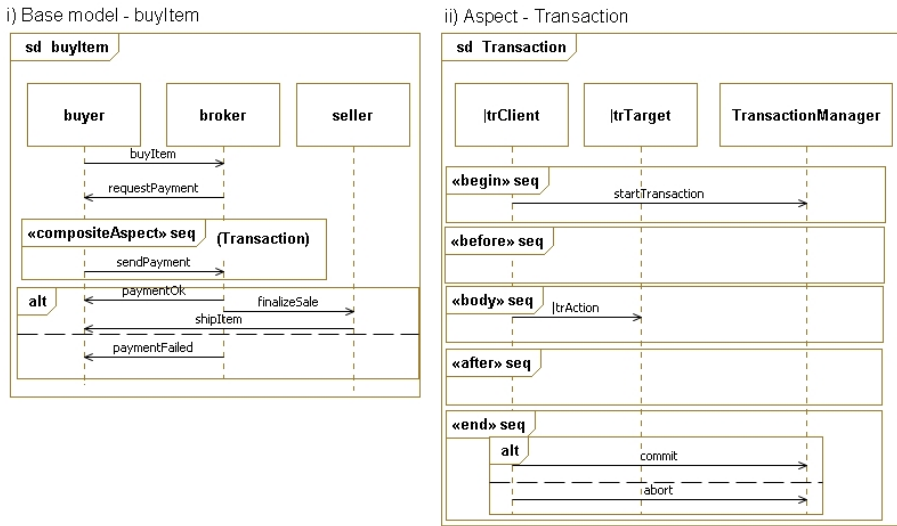


Fig. 8. Aspect modelling Solberg et al. Style

rule that is used for matching and composition with a base model. They use the concrete syntax of UML with the stereotypes to describe the aspects. In sequence diagrams, messages or combined fragments can be associated with these stereotypes to define the transformation/composition. The strength of this approach, which may also be its weakness, is the specification of pointcut and advice in the same sequence diagram. For the end user of a particular aspect in a particular context, it gives an intuitive view of the concern. However, it makes each aspect less reusable, as they reference the base model explicitly. In this approach, the base model is oblivious to the existence of the aspect (Fig.9).

The approach by Whittle et al. will break monotonicity with respect to refinement of the base and aspect advice, since matches may be lost and/or compositions may work

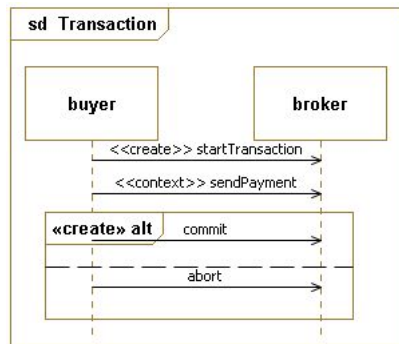


Fig. 9. Aspect Modelling Whittle et al. Style

differently in refinements, which is consistent with the problems described for syntactic aspect composition in Section 2.3. The approach allows removal of fragments from the base model, and does as such not provide event preserving compositions.

Clarke and Baniassad. Clarke and Baniassad have developed the Theme method [16], covering the lifecycle of software development using model-based concerns. The background of themes is work on composition in subject-oriented modelling [8]. Theme also addresses composition of sequence diagrams defined as behavioural part of themes and composition of aspect themes. A sequence diagram in an aspect theme defines template parameters that are used as names of messages in the sequence diagram. These are bound to concrete names during composition or *binding*. A theme can have several template parameters; for sequence diagrams, these refer to messages and operations of the lifeline types (Fig.10).

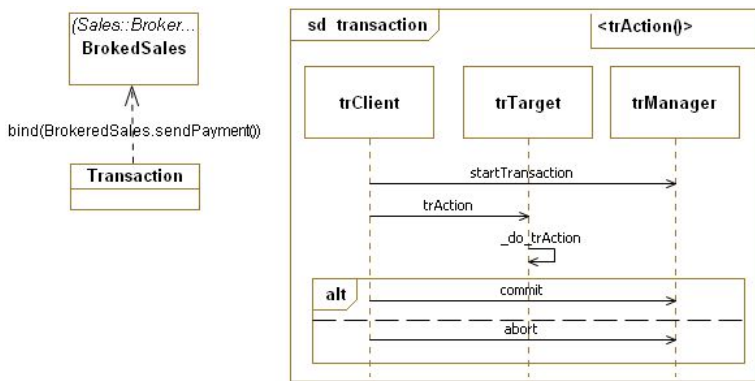


Fig. 10. Aspect Modelling Theme Approach

A crosscutting theme does not specify any compositional semantics with base theme sequence diagrams. It specifies how base behaviour is placed within crosscutting behaviour at the message/operation level; the execution of the base behaviour is always kept in its whole. There is an indirect reference to base operations that represent the original base behaviour. If we consider that this referenced base operation has associated sequence diagram behaviour, it could be inserted as an interaction use. The theme approach is designed for composing complete behaviours with crosscutting behaviour and does not allow merging of behaviour into a base sequence diagram.

In this approach, the base model and the aspect themes are mutually independent. The only links between them are in the bindings. A composition will always reference/use the base behaviour; hence, the approach is event preserving. If there is any base model sequence diagram behaviour, a refinement of it would not influence monotonicity. Refinement of the theme aspect sequence diagram will also result in monotonicity of theme composition.

Stein et al. Stein et al. [20] defines a syntax and semantics for definition of join-point queries in UML called Joint Point Designation Diagrams (JPDD). In this work,

Stein et al. address joinpoint queries within different UML diagram types, including sequence diagrams. They do not address the weaving/composition process. The notation for JPDD sequence diagrams is sequence diagrams extended with name patterns and wildcards, indirect call graph paths, and identifier variables. Since Stein et al. do not address the composition process, this approach is not relevant with respect to monotonicity of the compositions. However, their JPDD notation is an alternative graphical query language for sequence diagram aspects. It is based on UML, but extends it with various notational elements to provide advanced search mechanisms.

3.2 Semantic-Based Approaches

Klein et al. Klein et al. [18] address issues of semantic weaving of sequence charts, in contrast to syntax-based weaving. They define a semantics-based weaving algorithm for High-Level Message Sequence Charts (HMSC), which analyses the behaviour of sequence charts to identify the execution paths that matches a given pointcut. Pointcuts and advice are described in terms of basic MSCs, and the weaving algorithm computes the behavioural matches to the pointcut and refactors the base HMSC to the semantically desired result. This work is followed up in [21], focusing on weaving of several aspects that may be interfering with each other. They use sequence diagrams to describe pointcuts and advice of an aspect. In this approach, aspects may remove, complement, or replace matched behaviour. The main focus of Klein et al. is the weaving algorithm itself and its capability with respect to semantics-based weaving. Fig. 11 illustrates pointcut and advice sequence diagrams in this approach.

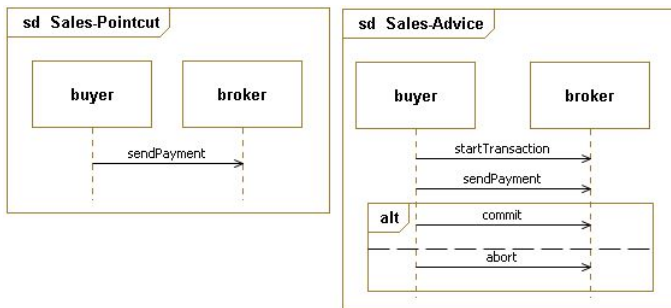


Fig. 11. Aspect Modelling Klein et al. Style

The base models are oblivious of the aspect, and the aspect references the base model. The advice part of an aspect is implicitly dependent of the pointcut part, as the advice copies the elements from the pointcut in order to replace the matches found. This does not influence refinements of the advice with respect to monotonicity. However, since the advice contains a copy of the elements from the pointcut, advice refinements may result in generation of new traces in which fragments from the pointcut is gone; still, the original pointcut fragments will always be present in some traces as long as the original advice specification contained a pointcut copy.

The approach provides monotonicity with respect to refinement of the base and aspect advice models. As it allows for deletion and replacement it is not event preserving.

Grønmo et al. Grønmo et al. [22] take a different approach to semantics-based weaving of sequence diagrams, based on mapping the sequence diagrams to traces and performing the weaving on the trace level. Using the semantic representation of traces of sequence diagrams of base models and aspects a close relationship with the semantic domain of sequence diagrams is ensured, and compositions done at this level will be monotonic with respect to refinement of the base model. They consider only additive aspects, i.e. aspects that do not remove or replace anything. The approach is thus event preserving.

They represent pointcut and advice in separate sequence diagrams, similarly to [18]. The base model is thus oblivious of the aspect, and the aspect pointcut references the base model. The same dependency between pointcut and advice exists as in [18]. However, this approach is also monotonic with respect to refinement of the aspect advice.

Deubler et al. Deubler et al. [12] defines an aspect-oriented modelling technique for services based on sequence diagrams. In this approach, an aspect is described as a sequence diagram with special kinds of lifelines representing pointcut and advice information. These identify join points of a base model and define what will happen with the aspect at these join points in terms of execution. Four types of join points are defined: before, after, wrapping, and overriding. They use names to identify join points, which may be grouped within combined fragments to specify more complex joinpoint expressions. They allow using wildcards for identifying joinpoints.

In this approach, the base model is oblivious of the aspect. The aspect itself mixes pointcut and advice in one sequence diagram, which makes refinement of the aspect advice hard, since a refinement of this model may modify the aspect pointcut. Deubler et al. do not specify the result of applying aspects to a base model. Rather, their approach is based on inserting the aspect behaviour at execution time of the scenarios. As such, their approach is semantic-based and it would cater for monotonicity with respect to refinement of the base model. However, since refinement of the aspect may modify pointcuts, the approach is not monotonic with respect to aspect advice refinement. Since their aspects may override behaviour, a composition may refrain from executing the behaviour from the base model. Hence, the approach is not event preserving.

Table 1 shows a feature overview of the different approaches. We see that four of six approaches indeed are monotonic with respect to base model refinement. The two approaches that mix pointcut and advice within a single sequence diagram are not monotonic with respect to advice refinement. One of the syntactical approaches is non-monotonic both with respect to base and advice refinement. In one case, we cannot conclude on the monotonicity properties of neither base nor advice refinement.

Our analysis shows that three out of six approaches provide monotonicity with respect to aspect composition and refinement. Only two of these are event preserving and hence *semantics preserving* under the definition given in Section 2.2. However, one could only restrict the last one with respect to fragment removal and it would be event preserving and semantics preserving.

We see that for the syntactical approaches, one provides monotonicity with respect to aspect composition and refinement. The same is true for two of the semantical

Table 1. Feature Table for Aspect Oriented Sequence Diagram Approaches

	Monotonic wrt refinement of base	Monotonic wrt refinement of aspect advice	Event Preserving
Solberg et al. (<i>syn</i>)	See Remark 1	See Remark 1	Yes
Whittle et al. (<i>syn</i>)	No	No	No
Clarke et al. (<i>syn</i>)	Yes	Yes	Yes
Deubler et al. (<i>sem</i>)	Yes	No	No
Klein et al. (<i>sem</i>)	Yes	Yes	No
Grønmo et al. (<i>sem</i>)	Yes	Yes	Yes

Remark 1. This approach requires the semantics of special tags in base and aspect models to be defined in order for refinement to be well-defined. Therefore, it is problematic to conclude on the monotonicity properties of this approach. However, if we absorb the assumptions outlined in Section 3.1, the approach will be monotonic.

approaches. We would expect that the syntactical approaches were not semantics preserving, but under the definition given in this paper, this is not necessarily so. *Why is this?* The answer lies in the quite restrictive mechanisms provided by these syntactical approaches: the approach in [16] is restricted in that it does not query base model behaviour to provide composition and does not *merge* base sequence diagrams with crosscutting behaviour; it only does sequential composition of crosscutting behaviour and complete base model sequence diagrams. The approach in [13] is restricted by the static identification of joinpoints in the base model, which restrain the flexibility of the aspect and controls where it can act.

4 Conclusions

We have defined semantics preservation for sequence diagram aspect composition based on refinement semantics and event preservation. Under this definition, a sequence diagram aspect approach is semantics preserving if it is monotonic with respect to base model and advice refinement and at the same time is event preserving, i.e. does not remove any fragments in a base model sequence diagram.

This definition of semantics preservation provides two things: it enables base models and aspects to be refined, and ensures that resulting composition after refinement are refinements of the compositions prior to refinement. It also ensures that no events that were part of the base model are lost or removed; events from the base model can be found within the composition traces if events from the aspect are hidden.

A lot of work has been done on defining the semantics of sequence diagrams, its compositional operators, and refinement [10,23,24]. Various semantics preservation characteristics have been addressed with respect to model evolution and model transformation: Engels et al. [25] address model consistency in model evolution by using model transformations that represent evolution steps. Baar et al. [26] address semantics preservation of model refactorings using graph transformation representation of OCL constraints. Katz and Katz [27] describe an approach for verifying the conformance of

a system to scenario-based aspect specifications. This is done by checking if system executions have a *convenient* computational representation in the scenario specifications. This work addresses a complimentary problems to ours. It addresses the adherence of aspect scenario semantics to a refinement (the implementation), while we address the preservation of refinement relationships when applying sequence diagram aspects.

Our analysis of sequence diagram aspect techniques addressed techniques with both syntactical and semantical matching and composition semantics. It shows that several approaches are semantics preserving under our definition. A general observation is that mixing of pointcut and advice in a single sequence diagram is negative with respect to monotonicity of aspect refinement, and hence semantics preservation. For syntactical approaches, we observe that they too can be semantics preserving, if they are strongly restricted in their querying mechanism. The more liberal approach results in compositions that are not semantics preserving.

In future work we will address if and how sequence-diagram aspects can be flexible, syntactically based, and still preserve semantics.

References

1. Haugen, Ø., Stølen, K.: STAIRS - Steps To Analyze Interactions with Refinement Semantics. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863. Springer, Heidelberg (2003)
2. Bergmans, L., Aksit, M.: Composing Crosscutting Concerns Using Composition Filters. *Commun. ACM* (10), 51–57 (2001)
3. Harrison, W., Ossher, H.: Subject-oriented Programming: a Critique of Pure Objects. *SIGPLAN Not* 28(10), 411–428 (1993)
4. Schärli, N., Ducasse, S., Nierstrasz, O., Black, A.P.: Traits: Composable units of behaviour. LNCS, pp. 248–274. Springer, Heidelberg (2003)
5. Kiczales, G.: Aspect-oriented programming. *ACM Comput. Surv.* 4es, 154 (1996)
6. Araujo, J., Whittle, J., Kim, D.K.: Modeling and Composing Scenario-based Requirements with Aspects. In: Requirements Engineering Conference, 2004. Proceedings. 12th IEEE International, pp. 58–67 (2004)
7. Baniassad, E., Clarke, S.: Theme: an Approach for Aspect-Oriented Analysis and Design. In: 26th International Conference on Software Engineering (ICSE), pp. 158–167 (2004)
8. Clarke, S., Walker, R.J.: Composition Patterns: An Approach to Designing Reusable Aspects. In: International Conference of Software Engineering (ICSE) (2001)
9. Oldevik, J., Haugen, Ø.: Architectural Aspects in UML. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735. Springer, Heidelberg (2007)
10. Haugen, Ø., Husa, K.E., Runde, R.K., Stølen, K.: STAIRS Towards Formal Design with Sequence Diagrams. *Software and Systems Modeling*, 355–357 (2005)
11. Runde, R.: STAIRS - Understanding and Developing Specifications Expressed as UML Interaction Diagrams. PhD thesis, Department of Informatics, University of Oslo (2007)
12. Deubler, M., Meisinger, M., Rittmann, S., Krüger, I.: Modeling Crosscutting Services with UML Sequence Diagrams. In: Bruehl, J.-M. (ed.) MODELS 2005. LNCS, vol. 3844. Springer, Heidelberg (2006)
13. Solberg, A., Simmonds, D., Reddy, R., France, R., Ghosh, S., Aagedal, J.: Developing Distributed Services Using an Aspect Oriented Model Driven Framework. *International Journal of Cooperative Information Systems* 15, 535–564 (2006)

14. Stein, D., Hanenberg, S., Unland, R.: Query Models. In: 7th International Conference of Modelling Languages and Applications. Springer, Heidelberg (2004)
15. Whittle, J., Jayaraman, P.: MATA: A Tool for Aspect-Oriented Modeling based on Graph Transformation. In: Eleventh International Workshop on Aspect-Oriented Modeling (AOM) (2007)
16. Clarke, S., Baniassad, E.: Aspect-Oriented Analysis and Design, The Theme Approach. Addison-Wesley, Reading (2005)
17. Haugen, Ø., Husa, K.E., Runde, R.K., Stølen, K.: Why Timed Sequence Diagrams Require Three-Event Semantics. Research Report 309, University of Oslo (2004) ISBN 82-7368-261-7
18. Klein, J., Helouet, L., Jezequel, J.: Semantic-based Weaving of Scenarios. In: AOSD 2006: The 5th International Conference on Aspect-oriented Software Development, pp. 27–38. ACM Press, New York (2006)
19. France, R.B., Kim, D.K., Ghosh, S., Song, E.: A UML-Based Pattern Specification Technique. *IEEE Trans. Softw. Eng.* 3, 193–206 (2004)
20. Stein, D., Hanenberg, S., Unland, R.: A UML-based Aspect-Oriented Design Notation for AspectJ. In: AOSD 2002: Proceedings of the 1st international conference on Aspect-oriented software development, pp. 106–112. ACM Press, New York (2002)
21. Klein, J., Fleurey, F., Jézéquel, J.M.: Weaving Multiple Aspects in Sequence Diagrams. *Transactions on Aspect Oriented Software Development (TAOSD)* (2007)
22. Grønmo, R., Sørensen, F., Møller-Pedersen, B., Krogh, S.: Weaving of UML Sequence Diagrams using STAIRS. Research Report 367, University of Oslo (2007)
23. Küster, J., Bowles, F.: Decomposing interactions. *Algebraic Methodology and Software Technology*, 189–203 (2006)
24. Bowles, J.: Decomposing Interactions. In: Johnson, M., Vene, V. (eds.) *AMAST 2006*. LNCS, vol. 4019, pp. 189–203. Springer, Heidelberg (2006)
25. Engels, G., Heckel, R., Küster, J.M., Groenewegen, L.: Consistency-Preserving Model Evolution through Transformations. In: 5th International Conference on The Unified Modeling Language, pp. 212–226 (2002)
26. Baar, T., Markovic, S.: A Graphical Approach to Prove the Semantic Preservation of UML/OCL Refactoring Rules. *Perspectives of Systems Informatics*, 70–83 (2007)
27. Katz, E., Katz, S.: Verifying Scenario-Based Aspect Specifications. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) *FM 2005*. LNCS, vol. 3582, pp. 432–447. Springer, Heidelberg (2005)

Generic Reusable Concern Compositions^{*}

Aram Hovsepyan, Stefan Van Baelen, Yolande Berbers,
and Wouter Joosen

Katholieke Universiteit Leuven, Distrinet,
Celestijnenlaan 200A, B-3001 Leuven, Belgium
{Aram.Hovsepyan, Stefan.VanBaelen, Yolande.Berbers,
Wouter.Joosen}@cs.kuleuven.be

Abstract. The increasing complexity of software applications requires improved software development techniques in order to cope with, a.o., software reuse and evolution, the management of heterogeneous concerns, and the retargeting of systems towards new software platforms. The introduction of AOSD (aspect-oriented software development) and the support for MDD (model-driven development) are two important and promising evolutions that can contribute to better control of software complexity. In this paper we present an AOM (Aspect-Oriented Modeling) based framework to promote and enhance the reuse of concerns expressed in UML. We have developed a prototype composition engine implemented in ATL that can be used to compose concern models specified in UML.

1 Introduction

Aspect-Oriented Modeling (AOM) is a recent development paradigm that aims at providing support for separation of concerns at higher levels of abstractions [1]. Following an AOM approach, one can model parts of a complete solution separately, and compose them afterwards using model composition techniques.

In order for AOM to bring an increase of efficiency in software development, it must be possible to model concerns once and reuse them in different contexts. If one has to create a concern every time from scratch, the gains of AOM will diminish substantially. Also, there is a need to define criteria that evaluate whether a given concern is reusable.

In this paper we define and characterize reusable concern models. We provide several key criteria that are crucial in improving concern model reusability. We have developed a graphical framework that can be used to model the structure and the behavior of certain types of concerns, and we specify their composition in a composition model. We have implemented a generic composition engine in ATL [2], which can compose UML concern models specified in our framework.

In section 2, we define the key requirements that are necessary to improve concern reusability. In section 3, we present the Generic Reusable Concern Compositions (GRCCo) approach in detail, and describe the composition specifics of our approach.

^{*} The described work is part of the EUREKA-ITEA MARTES project, and is partially funded by the Flemish government institution IWT (Institute for the Promotion of Innovation by Science and Technology in Flanders), by the Interuniversity Attraction Poles Programme Belgian State, Belgian Science Policy, and by the Research Fund K.U.Leuven.

In section 4, we illustrate our approach by applying it to a case study. In section 5, we evaluate how GReCCo improves reusability by discussing each of the reusability requirements presented in section 2. In section 6, we present related work. Finally, we conclude and outline future work.

2 Concern Reuse

Concerns are the primary motivation for organizing and decomposing software into manageable and comprehensible parts [3]. We use the term *concern* to uniformly refer to what AOSD practitioners often call an *aspect concern* and a *base* or *component concern* [4]. The *component concern* usually represents the functional core of a given application, whereas different *aspect concerns* represent functional and non-functional parts that extend the core.

To our knowledge, there is no clear notion of reuse for concern models in AOM. We therefore define and characterize what makes a given concern model more or less reusable.

2.1 Requirements for Reuse

We define a reusable concern model as a known solution for a certain problem, which can be used in several contexts to produce required assets. Different contexts mean for instance different applications, projects, companies, application domains, etc. However, using this definition we cannot measure how reusable a concern model is. That is why we introduce several more concrete qualities of a concern model and an AOM approach as a whole that are important in increasing the reusability of concerns.

Composition Obliviousness: Concerns, represented by their models, should ideally be modeled independently from a concrete context in which they are going to be applied. Reusable concerns usually represent generic solutions to known software problems, and therefore such concerns cannot be completely oblivious [5] of the context in which they will be applied. However, a given concern is more oblivious of the composition if it allows more variations in the composition. Consider the following simplified security concern in Figure 1 that represents a solution to realize secure logging [6]. The concern designer provides a complete generic solution, declaring that the *Client* class is a template entity that should be instantiated by a concrete element whenever this concern is used. However, if the concern needs to be composed with an application that already implements a *LogFactory*, it would be impossible to reuse this concern as it is. Figure 2 shows a different version of the same concern, which is relatively more oblivious [5] as it allows any element to be instantiated by a concrete element. Of course, a template entity may have a full implementation, which will be used in case it is not instantiated during a composition.

Composition Symmetry: All concerns should be treated uniformly, i.e., we should be able to compose any two given concerns. A non-symmetric (asymmetric) composition approach allows only *base-aspect* compositions. However, this will constrain the concern reuse as we will not be able to use concerns within other concerns.

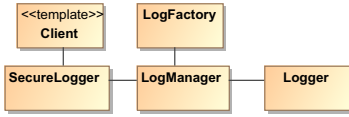


Fig. 1. Secure Logger Concern

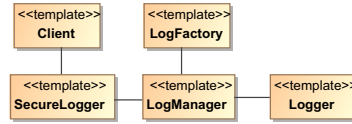


Fig. 2. Secure Logger Concern with Increased Obliviousness

Interdependency Management: There are many (often hidden) interactions between the different concerns, since they are not always completely orthogonal to each other. Such concern interactions can be classified to one of the following five categories: dependency, mutual exclusiveness, alternatives, conflict and mutual influence [7]. Hence, it is essential to be able to declare such a potential interdependency explicitly, and take it into account when composing different concerns. Otherwise a reused model may create an invalid composition by, e.g., introducing a dependency that is never resolved or adding a conflicting set of concerns.

In this paper we introduce a conceptual framework for representing concerns and specifying compositions with other concerns that improves support for reuse by tackling each of these requirements.

3 Generic Reusable Concern Compositions (GReCCo)

In this section, we first describe the general principle behind the GReCCo approach. Then we present the specifics of the composition of concern models. In addition, we discuss the problem of concern interactions and show how this can be tackled by integrating an existing complementary solution [7] into our approach.

3.1 General Description

The GReCCo approach is used to compose concern models. As presented on Figure 3, we represent each composition step as the Greek letter upsilon (Υ). The left and the right branches of the upsilon contain two concern models. Our approach is symmetric (**composition symmetry**) in the sense that we treat *component* and *aspect* concerns uniformly. In order to combine the concern models, we provide a composition model that instructs the model transformation engine how the two models should be composed.

Concern Models 1 and *2* (fig. 3) describe the structure and the behavior of the concerns using respectively UML class and sequence diagrams. The *Composition Model*, which also consists of UML class and sequence diagrams using the GReCCo profile, specifies how the concern models are composed by defining all composition-specific parameters and their bindings. This assures a higher degree of reusability of the two concerns as they can be used in different contexts (**composition obliviousness**). The *Composition Model* describes how the source concern models should be composed, and is therefore by definition depending on these models. Using a generic composition engine, we generate the output *Composed Model* from the input composition and concern models.

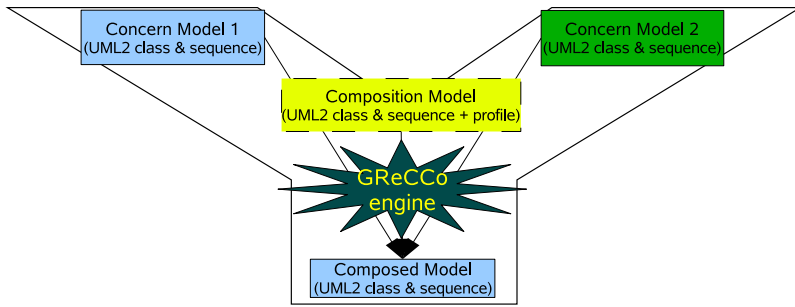


Fig. 3. General Approach

3.2 Composition Specification

In order to compose two concern models, we need to specify the composition by defining the *Composition Model*. We consider the structural and behavioral concerns of the composition separately. Elements that are not directly referenced in the *Composition Model* are copied to the *Composed Model*. Other elements are modified by the composition engine according to the composition specification. The next subsections describe the composition specifics in detail and illustrate some of them using simple examples. A complete set of illustrations can be found in [8].

3.2.1 Structure

We distinguish five structural composition directives in total. From the point of view of a single concern model, we distinguish the following directives that involve one element: (1) we can *add* a new element, (2) we can *modify* the properties of an existing element, and (3) we can *remove* an existing element. When two concern models are composed, there are some additional composition directives that we can specify for the input elements. We can (4) *merge* two elements to obtain a single entity with the combined properties. Some concerns introduce roles and/or template parameters as semantic variation points, which we can (5) *instantiate* by using concrete UML elements. As we are dealing with structure, we distinguish between four main types of UML elements: class, property, operation and association.

Add. The composition of two concerns can add new elements to the composed model. For instance, we may need to link a class from one of the input concern models with a class from the other one. In order to do so, we have to add the element to the composition model and tag it with the UML stereotype `<< add >>`, which will indicate that this is a new element that must be added to the composed model. Consider two input concerns containing the classes *A* and *B* respectively. Fig. 4 shows a composition model that shows how we can link *A* and *B* with an association.

Modify. During the composition, we may need to modify some properties of the existing elements. We can modify any UML property of a structural element. We specify this by marking the to be modified element with the `<< modify >>` stereotype in the composition model. In addition, we indicate the UML meta-property that needs to be

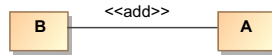


Fig. 4. Add Usage (Composition Model)

modified and its new value using a stereotype attribute (called a tag in UML 1.x). Each stereotype attribute will have a *name*, indicating the UML property that should be modified, a *type*, which is the same as the UML type of the property to be modified, and a *value*, indicating the new value. For instance, if we wish to modify the classname of a given class, we need to place a `<< modify >>` stereotype on that class, and fill out the *className* stereotype attribute with the new classname. If several UML properties of a given UML element need to be modified we will use several stereotype attributes on the same `<< modify >>` stereotype.

Remove. Due to the composition of concerns, certain entities may become unnecessary in the composed model. For instance, a concern may introduce an indirect association between two entities that are already connected directly in another concern. We realize the removal of elements by putting the `<< remove >>` stereotype on the element that needs to be removed.

Merge. When two concern models are involved, we sometimes have to deal with elements that represent a different view on the same entity. We need to merge these elements to obtain a single entity with combined properties. In order to merge two elements, we need to add a UML association between the elements and mark it with the `<< merge >>` stereotype. The merge results in a new element that is composed from the properties of the original two elements. The name of the composed element is set to the concatenation of the names of the original elements. Conflicts such as name clashes, mutually exclusive properties, etc. should be resolved explicitly by using the modify strategy. In the same manner, elements with the same name that are appearing in two models and that should be merged, must be explicitly labeled using the merge strategy.

Instantiate. A concern can contain a number of template elements (e.g., roles) that must be bound to concrete elements when the concern is composed. As we aim for oblivious compositions, we model all concern elements as concrete ones. On the composition model, however, we use the instantiation directive, which tells the composition engine that a given concern element is considered to be a template that must be instantiated by a concrete element from the other model. In practice, instantiation is similar to merge with the only difference that all conflicts are resolved by taking the properties of the concrete element. In addition, the name of the composed element is kept the same as the name of the concrete element. To specify an instantiation, we need to place an `<< instantiate >>` dependency link from the entity that must be considered to be a template to the concrete element.

3.2.2 Behavior

We use UML sequence diagrams to describe the behavior of concerns, which must be in correspondence with their structural counterpart specified by UML class diagrams.

A concern model may contain several sequence diagrams. Each sequence diagram represents a certain scenario. Scenarios from the input concern models that are completely independent of each other, are simply copied to the composed model. We define two scenario's as independent of each other if they can be executed in parallel, meaning that the messages can be freely interleaved. For overlapping behavior scenarios we need to be able to address the following use-cases: (1) specify the sequence of messages between the input behavior scenarios, (2) indicate that a call from one input scenario is the same as a call from the other input scenario, and (3) replace a call or a set of calls from one input scenario by a call or a set of calls from the other input scenario.

General Ordering. To realize the first use-case, we introduce the notion of *general ordering* partially borrowed from the UML 2.0 specification [9]. A general ordering is a binary relation between two interaction fragments to describe that one of the fragments must occur before the other one. Each interaction fragment contains a set of events. The resulting scenario defines a partial ordering of the input events. To specify this on a model we use a dependency between the event(s) that should precede another event(s) and mark it with the $\ll genordering \gg$ stereotype. The interpretation by the composition engine is that the dependency client fragment should immediately precede the dependency supplier fragment. Events that are not involved in any general ordering relation are put in parallel fragment blocks.

Merge. In order to specify the second use-case we use a dependency marked with the $\ll merge \gg$ stereotype between the two calls. If we do not use this dependency the calls will appear in duplicate on the composed sequence diagram. Note that event call merging is only possible when the call receiver classes are the same or if they have also been merged on the structural composition diagram.

Replace. Finally, to realize the last use-case, we group the event(s) that are to be replaced as well as the replacing events in interaction fragments. We place a $\ll replace \gg$ dependency from the replacing to the to-be-replaced fragments. Grouping in interaction fragments may be omitted in case of a single event or if the set of events is already grouped in an interaction fragment.

We illustrate the *merge* and *general ordering* concepts on the example of two behavior concerns that were described by Klein et al. [10]. Figure 5 shows an example of the composition specification. The left part of the composition represents the behavior of an authentication concern, while the right part describes the set of events for a logging concern. We would like to obtain a composed scenario that performs authorization and in case of a successful authentication writes it to a log. If the authorization fails, the authorization concern itself is in charge of saving the failed attempt. The *login()* events from the two concerns should be considered the same, therefore we place a $\ll merge \gg$ dependency link between them. In addition, we need to put the *logEvent()* event after the *OK* message. We specify this by placing a $\ll genordering \gg$ dependency from *OK* to *logEvent()*. As a result we obtain a combined sequence scenario that performs both authentication and logging in case of a successful authentication.

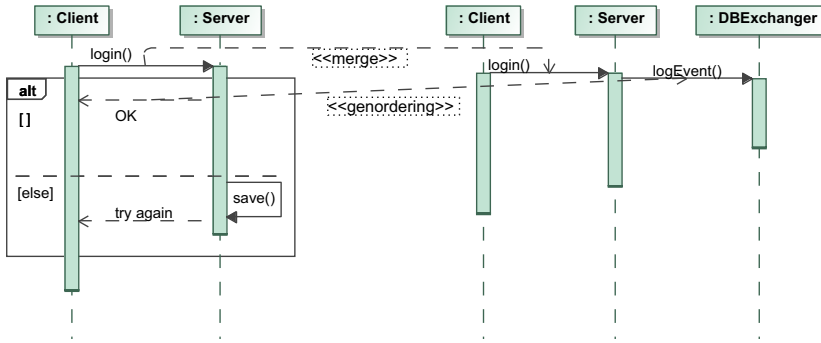


Fig. 5. Behavior Composition

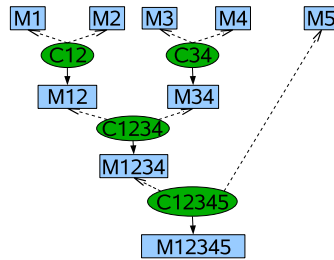


Fig. 6. Concern composition chain

3.3 Concern Interactions

One of the goals of our approach is the ability to build a system by reusing a number of existing concerns. Fig. 6 is an example of a composition tree that can be obtained if we combine five concern models. The rectangle boxes represent the UML models of five concerns. The concerns are composed in a certain order to obtain the final combined model (M12345). The ovals represent the composition models.

Concern models are rarely completely orthogonal to each other, but can relate to each other in a variety of different ways. A concern can be involved in an arbitrary number of interactions with one or more other concerns. Sanen et al. [7] distinguish between five different classes of concern interactions: dependency, conflict, choice, mutex and assistance. They also provide a conceptual Concern Interaction Acquisition (CIA) expert system for describing the relevant information about interactions between concerns that need to be captured. Figure 7 shows a simplified overview of the CIA framework. Domain experts add expertise about interactions between concerns into the CIA system. In order to use the CIA system for the investigation of concern interactions in GReCCo, the GReCCo tool has to provide the concern composition specification, which provides the CIA system with the information on concern selection. The list of the selected concerns is analyzed by the CIA system and the list of interactions is presented back to the GReCCo tool.

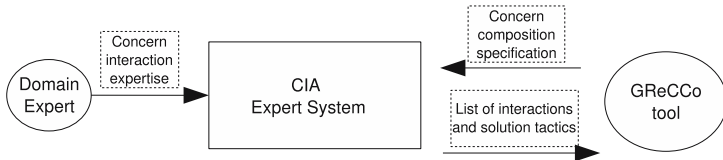


Fig. 7. Architecture of the CIA expert system from [7]

Our approach is currently constrained to the composition of only two concerns at a time. However, if we keep a composition history we could query about interactions with concerns that are already composed during the previous steps. An existing prototype of the CIA expert system is not yet incorporated in the GReCCo engine. However, its integration is conceptually straightforward and will be realized in the near future.

4 Case-Study

In this section, we present an application from the domain of Electronic Health Information and Privacy (EHIP). We start from a description of the primary model of the application. On top of this application, we apply several reusable concerns using the GReCCo methodology described in the previous section. Because of space restrictions some of the models will be shown only partially. For a complete description of this case study refer to [8].

4.1 Screening Application

Screening application represents an information system of a screening lab. Fig. 8 presents a UML class diagram for the screening lab application. Patients (*ScreeningSubject*) make an appointment to have their radiographic pictures (*Screening*) taken by a *Radiographer*. Two different *Radiologists* perform a *Reading* of the radiographic screening. In case the reading results are the same, an automatic *Conclusion* is generated. Otherwise, a third reading takes place, whereafter the third radiologist creates a final conclusion. In addition to the system itself, we have realized an additional client-server mechanism so

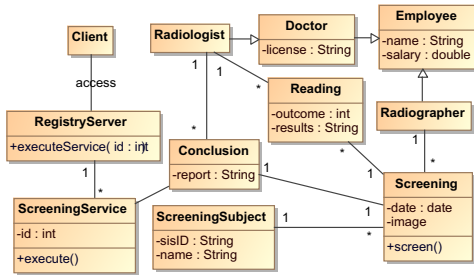


Fig. 8. Screening Lab Application Model

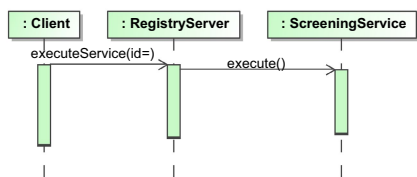


Fig. 9. Execute service

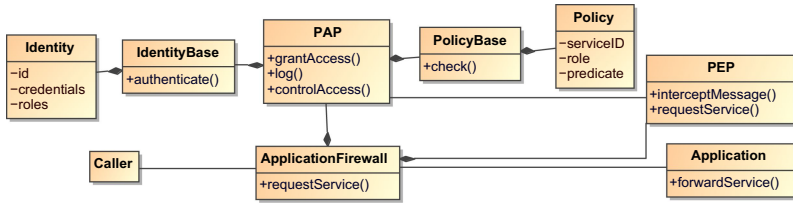


Fig. 10. Application Firewall Concern

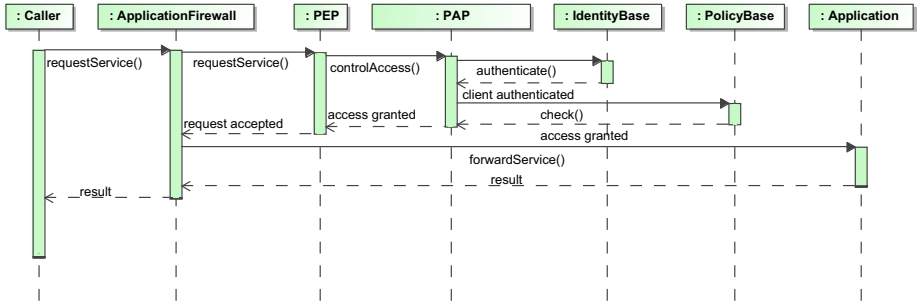


Fig. 11. Request Service

that patients can consult their own data at home using, e.g., a web browser (*Client*). *RegistryService* offers a set of *ScreeningServices*, each having its own *id*. Fig. 9 represents the scenario where patients execute a service in order to obtain their own data.

4.2 Application Firewall Concern

The registry server, introduced in the previous section, must ensure that only authenticated and authorized clients may use a given service. A sound solution in this case would be to interpose an application-level firewall [6] that can analyze incoming requests for the registry services and authorize them. A client can access a service of the registry server only if a specific policy authorizes it. Policies for each application are centralized within the *ApplicationFirewall* and they are accessed through a policy authorization point (*PAP*) (fig. 10 and 11).

We want to compose the application firewall concern with the screening application. We follow the framework described in the previous section and define the composition model for the structure. We specify that *Application*, *Caller* and *forwardService()* should be instantiated by *RegistryServer*, *Client* and *executeService()* elements respectively. In addition, we need to remove the direct association between *RegistryServer* and *Client* as the application firewall concern will introduce an indirect link between the two. For illustration purposes, we rename the *RegistryServer* to *Server* by placing a *« modify »* stereotype on the class, using a tag *name* to indicate the new name (fig. 12). Because of space restrictions we do not show the composition specification of the behavior models. For that we would need to place a *« replace »* dependency link from a fragment, which includes *executeService()* and *execute()* events, to *forwardService()* event.

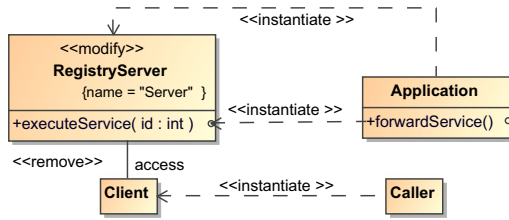


Fig. 12. Composition Model

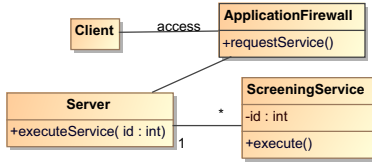


Fig. 13. Screening Application with Application Firewall Structure

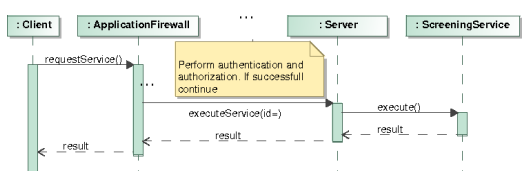


Fig. 14. Screening Application with Application Firewall Behavior

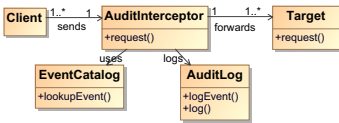


Fig. 15. Audit Interceptor Structure

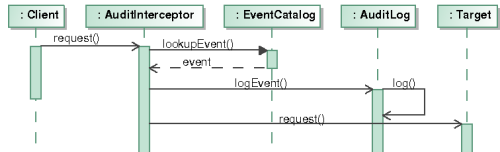


Fig. 16. Audit Interceptor Behavior

The structure and behavior of the resulting composed model are partially shown on fig. 13 and 14.

4.3 Audit and Secure Logger Concerns

In the next step of our case study we would like to add auditing support to our design. We have selected the Audit Interceptor concern (fig. 15 and 16) to centralize auditing functionality. An Audit Interceptor intercepts business tier requests and responses and creates audit events.

Audit Interceptor depends on some secure logging facility without which it is impossible to guarantee the integrity of the audit trails. This is why we introduce also the Secure Logger concern (fig. 17 and 18) that will ensure this additional requirement.

For illustration purposes, we chose to combine the audit interceptor and secure logger concerns using GReCCo first, and then apply the combined concern on the application model (fig. 13 and 14). The *AuditLog* entity in the Audit Interceptor concern represents the *Client* entity in the Secure Logger concern. Hence, in order to combine the two concerns, we have to merge the two entities by relating them with a $\ll merge \gg$ association (fig. 19). For the behavior composition we place a general ordering relation between the

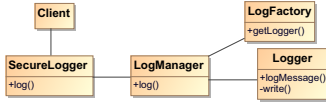


Fig. 17. Secure Logger Structure

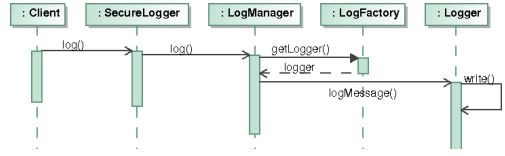


Fig. 18. Secure Logger Behavior

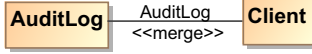


Fig. 19. Structure Composition Model

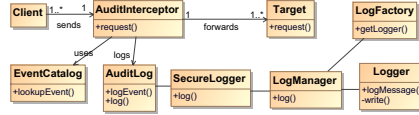


Fig. 20. Combined Structural Model

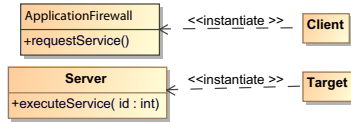


Fig. 21. Structure Composition Model

log() event from the *AuditLog* object and the *log()* event from the *SecureLogger*. The structure of the combined concern is shown on fig. 20.

4.4 Final Refined Application

In the final step we will compose the combined Audit and Secure Logging concerns (fig. 20) with the combined screening application and application firewall concern (fig. 13). In order to realize the structural composition, we specify that the *Client* class from the combined concern should be instantiated by the *ApplicationFirewall* class from the main application. We also instantiate the *Target* class by the *Server* class from the main application (fig. 21).

Fig. 22 shows the structure of the final application model with Application Firewall, Audit Interceptor and Secure Logger concerns composed into it.

5 Evaluation

Our approach represents a framework for AOM using reusable concerns, in which each concern is modeled using UML class and sequence diagrams. The approach comes with a prototype generic composition engine written in ATL, that can compose two concern models based on the composition directives defined in a dedicated composition model. In this section, we evaluate how our approach tackles the reusability requirements presented in section 2.

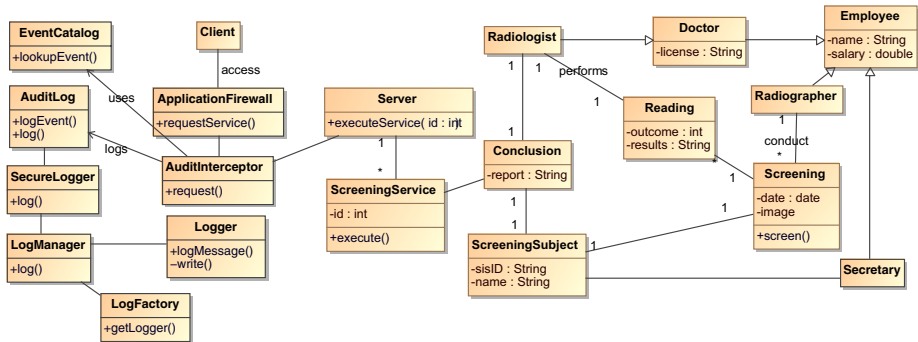


Fig.22. Final Application Model

5.1 Key Requirements for Reuse

We have illustrated in our case study how we can use the GReCCo approach to reuse modularized concerns. Each of the concerns used in our case-study can be reused in another application simply by defining an appropriate composition model.

Composition Obliviousness. In our approach, each concern is modeled independently from a concrete context in which it is going to be applied. All composition specifics, such as template parameters, structural and behavioral modifications, etc., are specified in a separate composition model, which is unique per composition. We are aware that the maximized reuse of concerns comes at the cost of a potentially larger and more complex composition model. However, we believe that we can minimize the effort required to build a composition model by introducing reusable composition models. We will investigate this in our future work.

Composition symmetry. Our framework supports a generic approach to concern composition, i.e., symmetrical composition. We use the term *concern* to uniformly refer to *base* and *aspect* concerns and we allow compositions between any two *concerns*.

Interdependency management. We have tackled this requirement by reusing an existing generic framework that helps us to detect and manage the different interactions when composing different concerns. The implementation of the interdependency management framework integration in GReCCo will be available in the near future.

5.2 Proof-of-Concept Implementation

We have developed a generic composition engine that realizes our approach on top of ATL. A current working version of GReCCo can be found on the GReCCo website [11]. All proposed composition directives for the structural composition have been implemented and evaluated with a number of examples [8]. The behavioral composition is not yet fully supported.

6 Related Research

There are many AOM approaches, each pursuing a distinguished goal and providing different concepts as well as notations. We consider them categorized by the alignment to phases criteria introduced by Op de beeck et al. [12]:

6.1 AOM Approaches Aligned to Implementation

Some AOM approaches offer no high level modeling concept that can map the design to the concerns identified during requirements phase. Typically they focus on modeling AOP concepts such as join points, pointcuts, advices, etc. [13,14,15,16]. Even though this allows the modeling of a given concern in a certain manner, these approaches remain too close to the implementation level. All these approaches typically come with a relatively rich tool support. However, these approaches do not score high given our reusability criteria. Almost all implementation centric approaches support composition asymmetry, which only supports *base-aspect* compositions. The composition context is very much pre-defined by the *aspect concerns* (composition obliviousness). In addition, as far as we know, these approaches provide little means to declare and detect concern interdependencies.

6.2 AOM Approaches Independent from Implementation

Several other approaches are implementation independent and provide higher-level mechanisms for concern modeling and composition. These approaches by default score better concerning reusability as they allow a more abstract concern representation and composition. Our approach belongs to this group.

The Theme approach of Clarke et al. [17] provides means for AOM in the analysis phase with *Theme/Doc* and in the design phase with *Theme/UML*. A *Theme* represents a modularized view of a concern in the system. It allows a developer to model features and aspects of a system, and specify how they should be combined. Theme supports composition symmetry and allows any given themes to be composed.

The Aspect-Oriented Architecture Models (AAM) approach of France et al. [18] presents an approach for composing aspect-oriented design models, where each aspect model describes a feature that crosscuts elements in the primary model. Aspect and primary models are composed to obtain an integrated design view. This approach is asymmetric and allows only aspect models to be composed with the primary model.

Klein et al. [19] present an approach for specifying reusable aspect models that define structure and behavior. The approach allows expressing aspect dependencies and weaving them in a dependency-consistent manner. The behavioral composition is realized using a semantic composition algorithm. Klein's approach is symmetric as well and allows does not differentiate between aspect and base models what concerns the composition.

Our approach is similar to these approaches, however, there are several key differences that result in a better concern reuse. All three approaches use a template-based mechanism for crosscutting concern compositions. Each crosscutting concern comes with a set of template parameters that are instantiated during the composition by concrete elements from the other concern. Template parameters already pre-define the

composition context as it is not easy to reuse a given concern with a different set of parameters (composition obliviousness). Our approach allows any element of a reusable concern to be parametrized. This provides a more flexible composition mechanism as the same concern can be reused in more contexts using a different set of instantiated parameters. Many concerns should (or must) be used with an a priori given set of parameters. However, there are concerns where it makes sense to allow a more flexible selection of parameters. For example, even though the Application Firewall concern (fig. 10 and 11) comes with a generic policy authorization point (PAP), we would like to allow the possibility of overriding it with a more specific PAP.

Klein's approach is the only one that allows the definition and detection of concern interdependencies. However, only one sort of interdependency, namely dependency, is supported. The CIA framework, on the other hand, is a systematic approach that supports the complete set of possible interdependencies.

6.3 Other Approaches

An approach conceptually similar to GReCCo, is the hyperspace approach [3]. The hyperspace approach is more generic as it allows separation of concerns along multiple dimensions called *hyperspaces*. GReCCo separates them only along class and concern dimensions. The concepts of class and concern in GReCCo can be seen as *hyperslices*, which represent concerns in a given hyperspace. The *declarative completeness* criteria, which requires hyperslices to declare everything to which they refer, is similar to the *composition obliviousness* criteria that requires concerns not to refer to any specific composition. *Declarative completeness* also implies the *composition symmetry* criteria. Finally, the *composition model* in GReCCo is close to a *hypermodule*, which integrates hyperslices. Hyper/J and Theme are the most prominent implementations of the hyperspace approach. However, Hyper/J is a code-level solution and Theme's crosscutting templates do not quite conform to the declarative completeness criteria. Notice that both implementations support only two dimensions/hyperspaces, similar to GReCCo.

7 Conclusions and Future Work

In this paper we have listed and discussed what in our view are key characteristics for the enhancement of concern reuse: composition obliviousness, composition symmetry and concern interdependency management. We have described a new approach for specifying concerns and their compositions and illustrated it on a case-study from the Electronic Health Information and Privacy (EHIP) domain. We have evaluated how the GReCCo approach can help us tackle each of the key qualities for improving reuse.

In the future we plan to investigate the possibility to reuse the composition models. We will also extend the GReCCo engine to support the behavioral composition and integrate it with the Concern Interdependency Acquisition (CIA) framework. We are also investigating the possibility to use domain-specific modeling languages for certain concerns. In addition, we plan to formalize the composition mechanisms and evaluate the scalability of our approach.

References

1. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: *Future of Software Engineering 2007* (2007)
2. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Bruel, J.-M. (ed.) *MoDELS 2005*. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
3. Ossher, H., Tarr, P.: Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In: *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development* (2000)
4. Kiczales, G.: Aspect-oriented programming (ACM Comput. Surv.) 154
5. Filman, R., Friedman, D.: Aspect-oriented programming is quantification and obliviousness. In: *Workshop on Advanced Separation of Concerns, OOPSLA 2000*, Minneapolis (2000)
6. Steel, C., Nagappan, R., Lai, R.: Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management (2005)
7. Sanen, F., Truyen, E., Joosen, W.: Managing concern interactions in middleware. In: Indulska, J., Raymond, K. (eds.) *DAIS 2007*. LNCS, vol. 4531, pp. 267–283. Springer, Heidelberg (2007)
8. Hovsepyan, A., Van Baelen, S., Berbers, Y., Joosen, W.: Generic reusable concern compositions (GReCCo): Description and case study. Technical Report CW 508, Department of Computer Science, K.U.Leuven (2008)
9. OMG: UML superstructure, v2.0. OMG Document number formal/05-07-04 (2005)
10. Klein, J., Fleurey, F., Jézéquel, J.M.: Weaving multiple aspects in sequence diagrams. *Trans. on Aspect Oriented Software Development* (2007)
11. Generic Reusable Concern Composition Engine, <http://www.cs.kuleuven.be/~aram/implementation.html>
12. Opdebeeck, S., Truyen, E., Boucké, N., Sanen, F., Bynens, M., Joosen, W.: A study of aspect-oriented design approaches. In: Technical Report CW435, Department of Computer Science, Katholieke Universiteit Leuven (2006)
13. Hanenberg, S., Stein, D., Unland, R.: From aspect-oriented design to aspect-oriented programs: tool-supported translation of jpdds into code. In: *AOSD*, pp. 49–62 (2007)
14. Pawlak, R., Seinturier, L., Duchien, L., Martelli, L., Legond-Aubry, F., Florin, G.: Aspect-oriented software development with java aspect components. In: *Aspect-oriented software development*, pp. 343–369 (2004)
15. Fuentes, L., Sánchez, P.: Execution of aspect oriented uml models. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) *ECMDA-FA*. LNCS, vol. 4530. Springer, Heidelberg (2007)
16. Cottenier, T., van den Berg, A., Elrad, T.: Joinpoint inference from behavioral specification to implementation. In: Ernst, E. (ed.) *ECOOP 2007*. LNCS, vol. 4609, pp. 476–500. Springer, Heidelberg (2007)
17. Baniassad, E., Clarke, S.: *Aspect-Oriented Analysis and Design: The Theme Approach*. Addison-Wesley, Reading (2005)
18. Reddy, Y.R., Ghosh, S., France, R.B., Straw, G., Bieman, J.M., McEachen, N., Song, E., Georg, G.: Directives for composing aspect-oriented design class models, pp. 75–105 (2006)
19. Klein, J., Kienzle, J.: Reusable aspect models. In: *Proc. of the 11th Int. Workshop on AOM* (2007)

Modeling Human Aspects of Business Processes – A View-Based, Model-Driven Approach

Ta'id Holmes, Huy Tran, Uwe Zdun, and Schahram Dustdar

Distributed Systems Group, Institute of Information Systems
Vienna University of Technology, Vienna, Austria
{tholmes, htran, zdun, dustdar}@infosys.tuwien.ac.at

Abstract. Human participation in business processes needs to be addressed in process modeling. BPEL4People with WS-HumanTask covers this concern in the context of BPEL. Bound to specific workflow technology this leads to a number of problems. Firstly, maintaining and migrating processes to new or similar technologies is expensive. Secondly, the low-level, technical standards make it hard to communicate the process models to human domain experts. Model-driven approaches can help to easier cope with technology changes, and present the process models at a higher level of abstraction than offered by the technology standards. In this paper, we extend the model-driven approach with a view-based framework for business process modeling, in which models can be viewed at different abstraction levels and different concerns of a model can be viewed separately. Our approach enables developers to work with meta-models that represent a technical view on the human participation, whereas human domain experts can have an abstract view on human participation in a business process. In order to validate our work, a mapping to BPEL4People technology will be demonstrated.

1 Introduction

In a process-driven, service-oriented architecture (SOA), process activities invoke services to perform the various tasks of the process. In such processes, often humans play a central role, and hence process activities must be provided that model human tasks and use services to “invoke” human actors who play a particular role in the process. In such a process-driven SOA with human involvement, various concepts and technologies (standard and proprietary) are involved. A typical standards-based solution in the Web services realm is to use REST [1] and SOAP [2,3] for distributed service invocations, WSDL [4] for service descriptions, BPEL [5] for orchestration of services through process models, BPEL4People [6] for human involvement in BPEL processes, and WS-HumanTask [7] to describe service-oriented descriptions of human tasks.

The diversity and constant evolution of these technologies and the underlying concepts hinders the changeability, understandability, and maintainability of process-driven SOAs – and hence makes evolution of process-driven SOAs a costly and error-prone undertaking. This is because systems are realized using specific technology without abstracting or conceptualizing the solutions. Historically, however, none of the technologies mentioned has emerged without having a precursor, and often companies have a legacy investment in one or multiple legacy technologies. For instance, the mentioned

BPEL standard evolved out of WSFL [8] and XLANG [9], which themselves emerged out of other existing workflow languages, and also there are several versions of the BPEL standard. The same is true for all other mentioned technologies and standards.

BPEL4People was proposed in a white paper in June 2005 as a technology for integrating human interaction of people with BPEL processes [10]. Two years later, in July 2007, version 1.0 of BPEL4People and the related WS-HumanTask [7] standards have been published. During this long period naturally solutions for integrating humans into service-oriented infrastructures have been proposed, for instance Human-provided Services [11]. Also BPEL4People concepts based on the white paper have been realized by industry and academia (see for instance [12]). They specified syntax and defined semantics for addressing the concepts as introduced in the white paper. These implementations now must be adapted to comply with the standards. Looking ahead it is clear that with new versions to come the current standards will become obsolete again in only a matter of time.

In order to reduce migration and maintenance costs, adaptation to such – rather typical – technology and (technology standards) life-cycles should be easy to perform. While concepts of a system may not change, new technology may introduce new syntax elements and may modify semantics. Therefore it is desirable to have conceptual representations within a system that have only the necessary dependencies on foundational technology.

The case of modeling human aspects in SOAs shows this aspect very clearly. There is a second, related problem that is also quite apparent in this case: the representation of the conceptual knowledge embedded in the technologies to end-users. While developers may be interested in the low-level, technical standards mentioned above, such technology-dependent views on a process-driven system are hard to communicate to domain experts. Instead a simplified view at a higher level of abstraction is needed.

Model-driven development (MDD) [13] addresses these needs by defining meta-models that express domain concepts. Platform-independent models that conform to these meta-models can subsequently be transformed to platform-specific code. In order to switch to similar but different technology all that needs to be done is to define a different transformation that transforms the conceptual models accordingly. Within this work we apply model-driven development to business process design. Particularly we focus on modeling business processes that involve humans.

However, these meta-models are typically still too detailed and technical to be presented to domain experts. To solve this issue, we propose to present a customized representation of our conceptual models to stakeholders. In particular, we propose to extend the model-driven approach using a view-based modeling framework [14] that realizes separation of concerns [15] for managing development complexity. For instance, while a business expert may be interested in the control-flow of a process, a human resource officer rather deals with assigning people to tasks. As we will introduce the framework, we will also present a concept of realizing such views as an extension of the model-driven approach.

But not only does the framework realize a separation of concern, it also enables developers to place meta-models at defined levels of abstraction. Complexity can be added gradually and while top level views give a simplified overview, refined models

may supply technical details on the given concern. This is needed for instance to model that technical developers need a technology-related view, while the two aforementioned stakeholders rather need less detailed views, abstracting from the technical details.

Within this work we will particularly refine an abstract human meta-model towards a technology-specific one for which a model-to-code transformation will be defined in order to obtain a BPEL4People process. That is, we will extend the view-based modeling framework by new views, dedicated to the concern of how people interact with business processes. While general Human-Process dependencies will be reflected by an abstract, conceptual view, details specific to BPEL4People and WS-HumanTask will be subjoined in an refined BPEL4People view. Finally, we will relate these meta-models to appropriate syntax elements.

This paper is structured as follows: After having defined the problem, Section 2 will introduce and extend a view-based modeling framework [14] with a meta-model for human-process participation and association. This view will then be extended with concepts from BPEL4People. A concrete binding to BPEL4People is presented in Section 3 by mapping these meta-models to the BPEL4People syntax. In Section 4 we will discuss related work and we will conclude with Section 5 by referring to further work.

2 A View-Based Approach

In order to describe human aspects of business processes, we have defined dedicated views for our view-based modeling framework (VbMF, see [14] for more details), which is introduced in this section. The VbMF framework consists of modeling elements such as a meta-meta-model, meta-models, and views. The meta-models are defined using the Eclipse Modeling Framework (EMF) [16]. These EMF models are utilized in openArchitectureWare (oAW) [17], a modular generator for MDD and model-driven architectures (MDA) [18], for integrating the view-based models and model-to-code-transformations.

A *view* is a representation of a process from the perspective of related concerns. In VbMF, new architectural views can be *designed*, existing meta-models can be *extended* by adding new features, views can be *integrated* in order to produce a richer view of a business process and using *transformations*, platform specific code can be generated.

Figure 2 demonstrates the core meta-model of the framework that is derived from the Ecore meta-meta-model [16]. It defines basic elements for the framework and introduces an element view as an representation of a business process concern. Other meta-models make use of this core meta-model, so that relationships between different meta-models can be defined and maintained. In particular model integration is realized by name-based matching amongst name attributes of elements within the core meta-model. Other matching algorithms, such as semantic matching can be plugged into VbMF, if needed.

As mentioned, the framework consists of multiple views, separating the various concerns that in their entirety describe an overall business process and the service integration. The main views defined by VbMF are: The *control-flow view* describes the control-flow of the process. The *collaboration view* specifies the orchestration of external activities. The *information view* contains details on data types and messages.

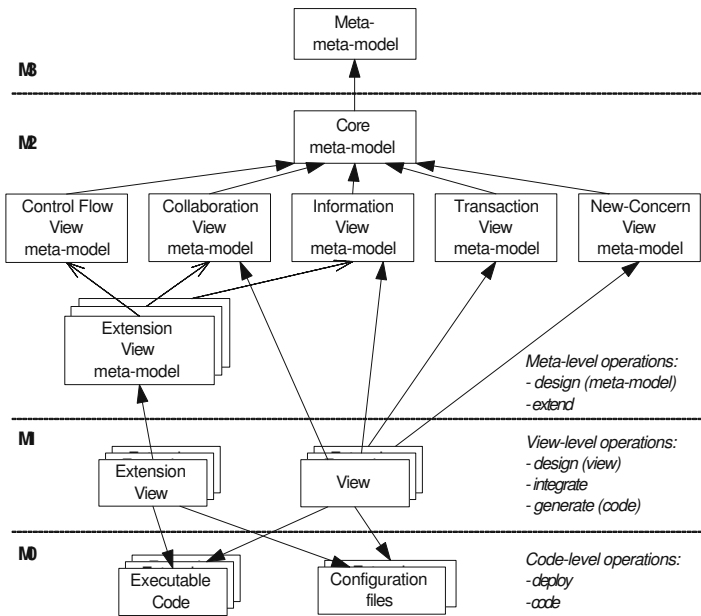


Fig. 1. Overview of the view-based modeling framework

The *transaction view* deals with long-running transactions, as they can be found in process-based systems. Figure 1 gives an overview of the VbMF framework and its main views.

VbMF is generally designed to be extensible in various ways. Firstly, VbMF can be extended with views for other concerns. We will illustrate this extensibility in the next section using a human view. Secondly, VbMF can be extended with views situated at different abstraction levels. For instance, for the above mentioned views, more technical views have been defined for BPEL/WSDL-based control-flow, collaboration, information, and transactions. Below we will illustrate this kind of extension using the BPEL4People view, which extends the human view.

2.1 Extending the Framework with Human Views

Figure 3(a) gives an overview of our extensions to the VbMF framework introduced in this paper: Two views dedicated to describe human aspects of business processes have been defined and a transformation for generating BPEL4People has been implemented (described in the next section).

We have extended the framework with a human view as shown in Figure 3(b) for describing human aspects of a process. Via name-based matching, it introduces the relation of processes and activities to human roles. Roles are abstracting concrete users that may *play* certain roles. The human view thus establishes a role-based abstraction. This role-based abstraction can be used for role-based access control (RBAC). RBAC, in general, is administered through roles and role hierarchies that mirror an enterprise's

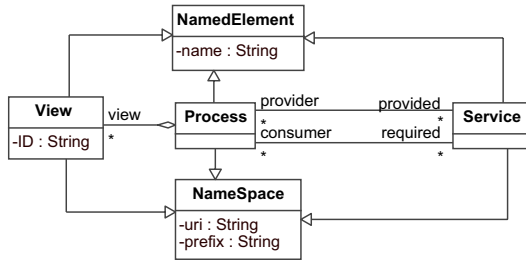


Fig. 2. The *core* meta-model

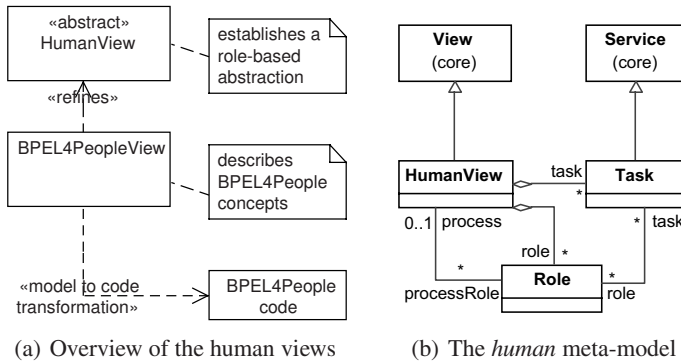


Fig. 3. Introducing the human views

job positions and organizational structure. Users are assigned membership into roles consistent with a user's duties, competency, and responsibility [19].

We can specify an activity as defined within a control-flow view to be a human task that is bound to for instance an owner, the person who performs the task. Likewise process stakeholders can be specified for the process by associating them with the human view that together with other views describes the overall process.

In the meta-models, described so far, there are no restrictions on how processes and tasks may be bound to roles. Particularly this view does not define or propose roles for processes or tasks. BPEL4People as well as WS-HumanTask on the other hand define generic human roles (see also Section 3.2). The specifications describe certain use case scenarios for different roles and therefore dictate access control for the human roles. By working with these roles, BPEL4People technology can help to reduce the complexity and cost of authorization management [19].

2.2 Refining the Meta-Model for BPEL4People

The more specific BPEL4People view extends the human view with a technology-specific perspective on the human aspect. BPEL4People glues BPEL activities and human tasks by introducing `peopleActivity` as a new BPEL extensionActivity.

The human tasks that may be encapsulated or referenced by the people activities are described in the WS-HumanTask specification. In order to hide complexity, these technology-related aspects are not shown in the generic human view, but only in the specific BPEL4People view.

A meta-model for the BPEL4People view is shown in Figure 4. This view inherits from the human view, binds roles to people links (that themselves are bound to concrete people queries) and integrates other *concepts* from BPEL4People. A task for example may hold a description, may be specified to be skipable and can specify scheduled actions that occur when time constraints are violated. Also propagation of ad hoc attachments from and to the process can be defined for a task.

Although for instance descriptions of tasks may already have been defined within a technology neutral meta-model, the optional description – which might be supplied for an arbitrary number of languages – is a specific requirement of BPEL4People. Therefore and in order to avoid polluting the abstract meta-model, the task description is specified – together with other technology-specific concepts – in the BPEL4People view.

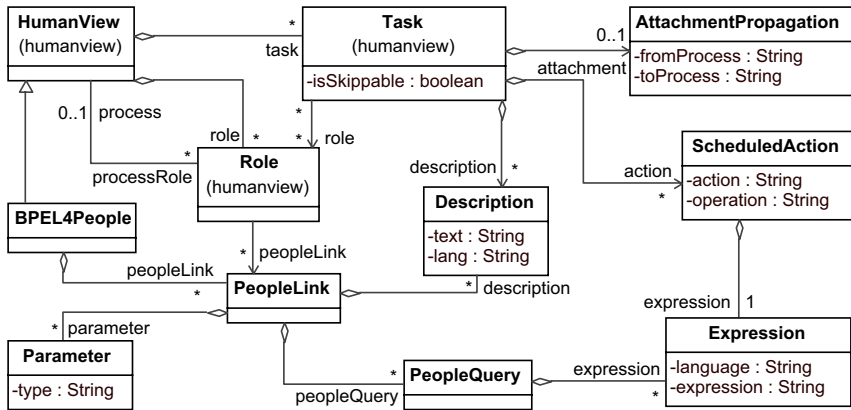


Fig. 4. Meta-model for the BPEL4People view

Roles need to be bound to a set of instances of data that identify persons. Therefore they make use of a *people link* that – when resolved by *people resolution* – results in such a set. People links contain *people queries*, can have *descriptions* and may specify *parameters*. Expressed in a certain language, people queries will be executed during people resolution. By decoupling roles from people links, *reuse* of the latter can take place. Furthermore – and as mentioned above – a role-based abstraction is established from the people link with its more technical aggregations.

3 Application to BPEL4People

To demonstrate the application of the presented models to BPEL4People and platform-specific code, we will elaborate on the mapping to BPEL4People and WS-HumanTask that define concrete syntaxes for human process interactions and relations. We will

conclude this section by demonstrating a use case example. For explaining how the different modeled concepts relate to the standards we will already utilize code of this use case for illustrating purposes within the following paragraphs. Although we will cover important concepts as captured within the introduced human view meta-models, we do not intend to be complete in regard to the specifications [6,7] within this paper.

3.1 Tasks

A human *task* is a process element that is part of the control-flow. It is realized by a *people activity* in BPEL4People as shown in Listing 1.1 that defines or references a *task definition*. Section 4.7 of [6] permits the specification of *scheduled actions* like *defer activation* and *expiration* that can contain *for* or *until* expressions. Moreover the propagation of attachments from and to processes can be specified for a people activity and the attribute `isSkipable` indicates whether the task associated with the activity can be skipped at runtime or not. Section 4.1.1 of [6] lists and describes the semantic of different properties.

```
<bpel:extensionActivity>
  <b4p:peopleActivity name="Acknowledgement"
    inputVariable="ack_input"
    outputVariable="ack_output"
    isSkipable="true">
    <b4p:localTask reference="tns:AcknowledgementTask"/>
    <b4p:scheduledActions>
      <b4p:deferActivation>
        <b4p:for>
          ...
        </b4p:for>
      </b4p:deferActivation>
    </b4p:scheduledActions>
    <b4p:attachmentPropagation fromProcess="all" toProcess="all"/>
  </b4p:peopleActivity>
</bpel:extensionActivity>
```

Listing 1.1. BPEL4People syntax for a human *process element*

In our view-based modeling framework human tasks are modeled as *simple activities* in the control-flow view. Within the human view such activities can be annotated via *name-based matching* to contain human aspects by specifying the name as denoted in the simple activity of the control-flow view.

We translate a task that was modeled in the human view to an appropriate task definition and reference it within a people activity. In contrast to encapsulated tasks, local reuse of task definitions within a process can thus take place which might result in an optimized behavior of the runtime engine that hosts the BPEL4People process.

3.2 Roles

Generic human roles as *process stakeholders*, *process initiators* and *business administrators* for processes have been defined in Section 3.1 of [6]. The human view meta-model contains an association between the human view of the process and roles that may define these generic human roles. Analogically this can be done for the task-role

association. Section 3.1 of [7] defines appropriate roles for tasks such as *task initiator*, *task stakeholders*, *potential owners*, *actual owner*, *excluded owners*, *business administrators* and *notification recipients*.

Process Roles. When mapping to BPEL4People we define the generic human roles for the process within a `peopleAssignments` container by referencing to corresponding `logicalPeopleGroups` as shown in Listing 1.2.

```
<b4p:peopleAssignments>
  <b4p:businessAdministrators>
    <htd:from logicalPeopleGroup="businessAdministratorsLPG" />
  </b4p:businessAdministrators>
  <b4p:processInitiator>
    <htd:from logicalPeopleGroup="processInitiatorLPG" />
  </b4p:processInitiator>
  <b4p:processStakeholder>
    <htd:from logicalPeopleGroup="processStakeholderLPG" />
  </b4p:processStakeholder>
</b4p:peopleAssignments>
```

Listing 1.2. BPEL4People syntax for associating human *roles* to a *process*

Task Roles People assignment for generic human task roles is performed within a task definition. As with process roles we reference corresponding `logicalPeopleGroups` as shown in Listing 1.3.

```
<htd:task name="AcknowledgementTask">
  <htd:interface operation="ack"
    portType="acknowledgementservice:acknowledgePT" />
  <htd:peopleAssignments>
    <htd:taskInitiator>
      <htd:from logicalPeopleGroup="taskInitiatorLPG" />
    </htd:taskInitiator>
    <htd:taskStakeholders>
      <htd:from logicalPeopleGroup="taskStakeholdersLPG" />
    </htd:taskStakeholders>
    <htd:potentialOwners>
      <htd:from logicalPeopleGroup="acknowledgementPotentialOwnersLPG" />
    </htd:potentialOwners>
    <htd:notificationRecipients>
      <htd:from logicalPeopleGroup="notificationRecipientsLPG" />
    </htd:notificationRecipients>
    <htd:excludedOwners>
      <htd:from logicalPeopleGroup="peopleNotAllowed2AcknowledgeLPG" />
    </htd:excludedOwners>
  </htd:peopleAssignments>
</htd:task>
```

Listing 1.3. WS-HumanTask syntax for associating human *roles* to a *task*

The binding of roles to people links within the BPEL4People view is not restricted to a one-to-one mapping. Instead reuse of people links can take place as no composition is specified for the relation.

3.3 People Links

People links are transformed to `logicalPeopleGroup` elements as shown in Listing 1.4. Descriptions are translated to `documentation` elements that – together with

optional parameters, that data can be used for people query evaluation – are placed as sub-elements within the corresponding elements.

```

<htd:logicalPeopleGroups>
  <htd:logicalPeopleGroup name="peopleNotAllowed2AcknowledgeLPG">
    <htd:documentation xml:lang="en">
      These people are not allowed to
      acknowledge the order.
    </htd:documentation>
    <htd:parameter name="name" type="xs:string"/>
    ...
  </htd:logicalPeopleGroup>
  <htd:logicalPeopleGroup name="processStakeholderLPG" >
    ...
  </htd:logicalPeopleGroup>
  ...
</htd:logicalPeopleGroups>

```

Listing 1.4. WS-HumanTask syntax for associating *people links* to *people queries*

We have stated a possible mapping from the presented conceptual meta-models to BPEL4People code. We have seen that elements like people links, together with their aggregated descriptions and parameters, have concrete relations to designated BPEL4People and WS-HumanTask syntax elements.

3.4 A Use Case Scenario

In this section, we have explained the main concepts of human-process relation in regard to the BPEL4People and WS-HumanTask specifications. For demonstrating purposes we already have shown code examples that derive from a use case that we now want to describe in more detail where we will illustrate code generation using model-to-code templates. These templates are written in Xpand, which is a language for model-to-code transformation within the oAW's expression framework.

We illustrated a use case scenario for a shopping ordering process. At a certain stage of the process an acknowledgement from an authorized person is required. Therefore the process makes use of a human task as a special kind of a process activity.

We have modeled this scenario using the VbMF and Figure 5 shows how the human task *Acknowledgment* has been designed in various views. While Figure 5(a) shows the control-flow of the process with a corresponding *simple activity*, Figure 5(c) defines the task in the human view. Invocation of activities as well as human tasks with input and output variables is specified in the collaboration view as shown in Figure 5(b). Besides the definition of the human task, the appropriate generic human roles, that are associated with the process and the tasks, are also defined in the BPEL4People view.

People links are transformed into a set of logicalPeopleGroups as Figure 6 demonstrates. In order to obtain valid BPEL4People code the element names for the generic human roles as defined in [6] and [7] have to be used as the name of the people links when modeling. Figure 7 shows the generation of task definitions with task roles that reference corresponding people links.

The invocation of process activities may be performed by a Web service invocation using the BPEL *invoke* activity or in case of a local task the BPEL4People

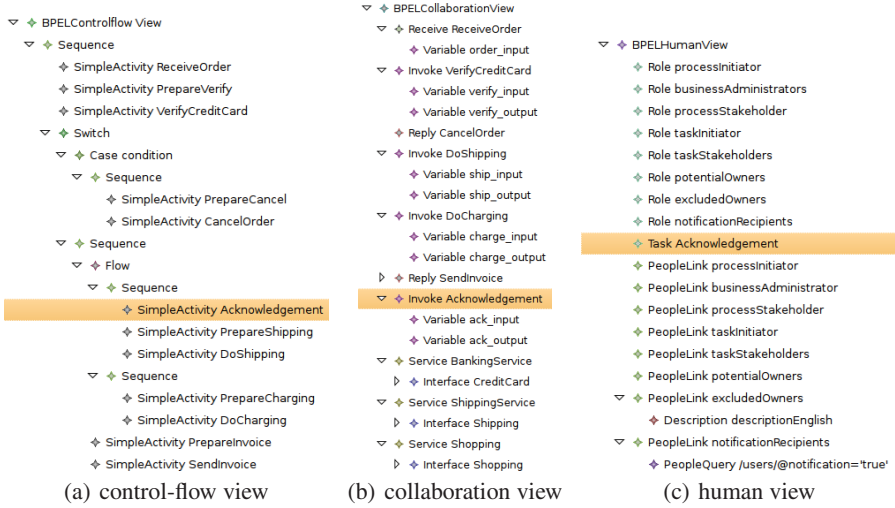


Fig. 5. A human task within the different views

```

«DEFINE LogicalPeopleGroups(bpelinformation::BPELInformationView iv,
    bpelcollaboration::BPELCollaborationView cv,
    bpel4people::BPEL4People_HumanView hv)
  FOR controlflow::ControlFlowView-»
    «IF (getPeopleLinks(hv).size > 0)-»
      <htd:logicalPeopleGroups>
        «FOREACH getPeopleLinks(hv) AS link-»
          <htd:logicalPeopleGroup name="«link.name»LPG">
            «FOREACH link.description AS description-»
              <htd:documentation«IF (description.lang != null)-»
                xml:lang="«description.lang"«ENDIF»>
                «description.text»
              </htd:documentation>
            «ENDFOREACH-»
            «FOREACH link.parameter AS parameter-»
              <htd:parameter name="«parameter.name"»
                type="«parameter.type"» />
            «ENDFOREACH-»
          </htd:logicalPeopleGroup>
        «ENDFOREACH-»
      </htd:logicalPeopleGroups>
    «ENDIF-»
  «ENDEDEFINE»

```

Fig. 6. Model-to-code template for *people links*

peopleActivity. Figure 8 shows the transformation template that generates an appropriate extensionActivity for the peopleActivity in case a task is found in the human view for the activity as specified in the control-flow by name-based matching. The getTaskByName function accomplishes this matching algorithm. This referenced function has been implemented in Xtend, another language of the oAW's expression framework, that provides the possibility to define libraries of independent operations

```

«DEFINE Tasks(bpelinformation::BPELInformationView iv,
    bpelcollaboration::BPELCollaborationView cv,
    bpel4people::BPEL4People_HumanView hv)
    «FOR controlflow::ControlFlowView-»
    «IF (getTasks(hv).size > 0)-»
    <htd:tasks>
    «FOREACH getTasks(hv) AS task-»
    «LET getInvokeByName(task.name, cv) AS invoke-»
    <htd:task name="«task.name»Task">
    <htd:interface
        operation="«invoke.operation.name»"
        portType="«invoke.associatedInterface.service.name.toLowerCase(
            )»:«invoke.associatedInterface.name»PT"
    />
    <htd:peopleAssignments>
    «FOREACH getTaskRoles(task) AS role-»
    <htd:«role.name»>
    <htd:from logicalPeopleGroup="«role.peopleLink.name»LPG" />
    </htd:«role.name»>
    «ENDFOREACH-»
    </htd:peopleAssignments>
    <htd:presentationElements>
    «FOREACH task.documentation AS description-»
    <htd:description«IF (description.lang != null)-»
    xml:lang="«description.lang»"«ENDIF» contentType="text/plain">
    «description.text»
    </htd:description>
    «ENDFOREACH-»
    </htd:presentationElements>
    </htd:task>
    «ENDLET»
    «ENDFOREACH-»
    </htd:tasks>
    «ENDIF-»
«ENDEDEFINE»

```

Fig. 7. Model-to-code template for *tasks*

and no-invasive meta-model extensions [17]. If no task has locally been specified for the process, an external activity will be invoked.

The control-flow view does not distinguish between the invocation of an external activity or the delegation to a local human task. Adopting the notion of transparency [20] we can therefore say that invocation is made transparent for modeled activities of the control-flow from a design point of view. As a consequence, activities can simply be exchanged between human tasks or Web service invocations without the need to alter the control-flow in turn.

4 Related Work

Related Work on Model-Driven Design Our work is based on the model-driven development (MDD) paradigm [21] and extends the MDD paradigm with the notion of architectural views – expressed in terms of meta-models. Two kinds of extensibility are supported: “horizontal” extension with views for additional concerns and “vertical” extension with views at different abstraction levels. Using our view-based approach, we are able to separate the human view – in focus of this paper – from other concerns, and

```

«DEFINE SimpleActivity(bpelinformation::BPELInformationView iv,
    bpelcollaboration::BPELCollaborationView cv,
    bpel4people::BPEL4People_HumanView hv)
FOR bpelcollaboration::Invoke-»
«LET getTaskByName(name, hv) AS task-»
    «IF (task != null)-»
        <bpel:extensionActivity>
            <b4p:peopleActivity name="«name»"
            «IF (in != null)-»
                inputValue="«in.name»"
            «ENDIF-»
            «IF (out != null)-»
                outputVariable="«out.name»"
            «ENDIF-»
            «IF (task.isSkipable != null)-»
                isSkipable="«task.isSkipable»"
            «ENDIF-»
            >
                <b4p:localTask reference="tns:«name»Task"/>
            </b4p:peopleActivity>
        </bpel:extensionActivity>
    «ELSE»
        <bpel:invoke name="«name»"
        «IF (in != null)-»
            inputValue="«in.name»"
        «ENDIF-»
        «IF (out != null)-»
            outputVariable="«out.name»"
        «ENDIF-»
        partnerLink="«partnerLink.name»"
        «LET associatedInterface.service.name.toLowerCase() AS prefix-»
        portType="«prefix»:«associatedInterface.name»"
        «ENDLET-»
        operation="«operation.name»" />
    «ENDIF-»
«ENDLET -»
«ENDDEFINE»

```

Fig. 8. Model-to-code template for activity *invocation*

separate the views for different stakeholders on the human view: high-level views for domain experts and low-level views for technical experts.

List and Korherr [22] compare and evaluate seven conceptual business process modeling languages and propose a generic meta-model that is categorized according to the framework introduced in [23] to four perspectives: organisational, functional, behavioural, and informational. Additionally a perspective for the business context addresses context information like process goals. While it is interesting to capture different conceptual business process modeling languages into a common normalized meta-model we do not need to support these in order to obtain code for process runtime execution. As a matter of fact, a mapping from high level business process modeling languages such as the Business Process Modeling Notation (BPMN) to a certain technology such as BPEL often is missing [24,25,26]. Instead of a comprehensive meta-model we, moreover, want to work with small conceptual models as proposed in [27] that rather represent the least common denominator but can be extended and bound and/or translated to low-level models that support process execution.

A Model Driven Visualization framework is introduced by Bull [28] that provides a mechanism to rapidly prototype new visualizations from meta-models. So called *snap points* define views for domain experts. Considering that the VbMF defines the meta-models for business processes, a direct application of the presented work - that also is based on EMF - would allow customized business process representation to various stakeholders.

Related Work on View-Based Modeling. There are only a few view-based approaches to business process modeling. To the best of our knowledge, none of them integrates a human view. The approach by Mendling et al. [29] is inspired by the idea of schema integration in database design. Process models based on Event-driven Process Chains (EPCs) [30] are investigated, and the pre-defined semantic relationships between model elements such as *equivalent*, *sequence*, and merge operations are performed to integrate two distinct views. In contrast our approach introduces a common core meta-model and well-defined extension points, and utilizes the model-driven paradigm for view integration. That is, our approach is both more flexible and provides more well defined extension points for view integration and extension.

The Amfibia [31,32] approach focuses on formalizing different aspects of business process modeling, and/or develops an open framework to integrate various modeling formalisms through the *interface* concept. Akin to our approach, Amfibia has the main idea of providing a modeling framework that does not depend on a particular existing formalism or methodology. The major contribution in Amfibia is to exploit dynamic interaction of those aspects. Like our approach, Amfibia's framework also has a core model with a small number of important elements, which are referred to, or refined in other models. The distinct point to our framework is that in Amfibia the interaction of different 'aspects' is only performed by event synchronization at run-time when the workflow management system executes the process. Using extension and integration mechanisms in our framework, the integrity and consistency between models can be verified earlier at the model level.

Related Work on Role-Based Abstraction. Working with human labor, the use and administration of roles in regard to business processes is another topic that relates to our work as by modeling the human aspects of processes, relations between processes and tasks to authorized roles are defined.

Johnson and Henderson [33] propose data authorization and access control mechanism for Workflow Management Systems (WfMS) [34]. Similar to our human view the presented comprehensive access control model also establishes - besides defining other relations - a role based abstraction. Predicate-based access control is applied for implementation.

A standard for a functional RBAC model has been proposed by Ferraiolo et al. [19]. While the human view within the VbMF establishes role based abstraction, the actual access control for different roles is defined implicitly and applied by the technology. For instance, it is possible to specify the role of a business administrator in the human view and assign actual users to this role via people queries in the BPEL4People view. The semantics concerning the access controls of this role however are defined in the BPEL4People specification and will only be interpreted by the executing engine. Therefore, there is no need to model access controls within the VbMF as assignment

of people to specific roles suffices in order to obtain a business process with predefined role based access controls for human participants.

5 Summary

We have presented meta-models for expressing human aspects of business processes within a view-based modeling framework that support the specification of processes and tasks containing human aspects. Modeling the human aspects in a process-driven SOA is challenging because technology for human aspects (such as BPEL4People and WS-HumanTask) is constantly evolving, the technology is dependent on many other technologies which also evolve (such as BPEL and Web service technology used in our work), and, finally, different stakeholders, such as domain experts in the field of human resource management, as well as software architects and developers must work with the models – and require different views. Our approach resolves this challenging case by providing a view-based modeling extension to the model-driven paradigm and by providing models that cover the concern of human-process relation. The presented views are split into an abstract, platform independent meta-model as well as an refined one. While the latter can be used for model-to-code transformation, the conceptual model is suitable for being presented to human domain experts. By exchanging the transformations and/or adapting the low-level models, the adaptation of a refined model to a new version of a standard or to another technology can easily be performed. Via model-to-code transformation we have demonstrated a possible mapping to BPEL4People as a specific technology addressing the mentioned concern. While in this work, we have focused on the case of human aspects in process models, the same view-based approach can be generalized and be applied to other cases with similar requirements as well.

In addition to applying our approach to other cases, we plan for the following further work: Specifying the relationship between models by formalizing *integration points* would permit automatic and generic model integration. Therefore we plan to extend the framework with a domain specific language tailoring the merging of views. In order to support more features of BPEL4People like *escalation*, additional refinements to the VbMF may be defined to the presented basic views. Besides the design of business processes, also the monitoring needs to be addressed by conceptual models. Therefore we plan to provide execution views for capturing business processes runtime states.

References

1. Fielding, R.T.: Architectural styles and the design of network-based software architectures. PhD thesis, Chair-Richard N. Taylor (2000)
2. W3C Recommendation: SOAP Version 1.2 Part 1: Messaging Framework (April 2007)
3. W3C Recommendation: SOAP Version 1.2 Part 2: Adjuncts (April 2007)
4. W3C Note: Web Services Description Language (WSDL) Version 1.1
5. Curbera, F., Khalaf, R., Mukhi, N., Tai, S., Weerawarana, S.: The next step in web services. Commun. ACM 46(10), 29–34 (2003)
6. Agrawal, A., Amend, M., Das, M., Ford, M., Keller, C., Kloppmann, M., König, D., Leymann, F., Müller, R., Pfau, G., Plösser, K., Rangaswamy, R., Rickayzen, A., Rowley, M., Schmidt, P., Trickovic, I., Yiu, A., Zeller, M.: WS-BPEL Extension for People (BPEL4People), Version 1.0 (June 2007)

7. Agrawal, A., Amend, M., Das, M., Ford, M., Keller, C., Kloppmann, M., König, D., Leymann, F., Müller, R., Pfau, G., Plösser, K., Rangaswamy, R., Rickayzen, A., Rowley, M., Schmidt, P., Trickovic, I., Yiu, A., Zeller, M.: Web Services Human Task (WS-HumanTask), Version 1.0 (June 2007)
8. IBM Software Group: Web Services Flow Language (WSFL) version 1.0 (2001)
9. Thatte, S.: Web Services for Business Process Design. Microsoft Corporation (2001)
10. IBM-SAP whitepaper: WS-BPEL Extension for People (August 2005)
11. Schall, D., Truong, H.L., Dustdar, S.: On Unifying Human and Software Services for Ad-hoc and Process-centric Collaboration. *IEEE Internet Computing* 12(3) (2008)
12. Holmes, T., Vasko, M., Dustdar, S.: VieBOP: Extending BPEL Engines with BPEL4People. In: PDP 2008. 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing, February 13–15, 2008, pp. 547–555 (2008)
13. Völter, M., Stahl, T.: Model-Driven Software Development: Technology, Engineering, Management. Wiley, Chichester (2006)
14. Tran, H., Zdun, U., Dustdar, S.: View-based and Model-driven Approach for Reducing the Development Complexity in Process-Driven SOA. In: Intl. Working Conf. on Business Process and Services Computing (BPSC 2007). *Lecture Notes in Informatics*, vol. 116, pp. 105–124 (September 2007)
15. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering. Prentice-Hall, Englewood Cliffs (1991)
16. Foundation, T.E.: Eclipse Modeling Framework (2006), <http://www.eclipse.org/modeling/emf/>
17. openArchitectureWare.org: openArchitectureWare (2006), <http://openarchitectureware.org>
18. Open Management Group: MDA Guide Version 1.0.1 (June 2003)
19. Ferraiolo, D.F., Barkley, J.F., Kuhn, D.R.: A role-based access control model and reference implementation within a corporate intranet. *ACM Trans. Inf. Syst. Secur.* 2(1), 34–64 (1999)
20. Carpenter, B.: Internet transparency (2000)
21. OMG: MDA Guide Version 1.0.1. Technical report, Object Management Group (2003)
22. List, B., Korherr, B.: An evaluation of conceptual business process modelling languages. In: SAC 2006 Proceedings of the 2006 ACM symposium on Applied computing, pp. 1532–1539. ACM Press, New York (2006)
23. Curtis, B., Kellner, M.I., Over, J.: Process modeling. *Commun. ACM* 35(9), 75–90 (1992)
24. White, S.A.: Using BPMN to Model a BPEL Process (February 2005)
25. Ouyang, C., Dumas, M., ter Hofstede, A.H.M., van der Aalst, W.M.P.: From BPMN Process Models to BPEL Web Services. In: ICWS 2006: Proceedings of the IEEE International Conference on Web Services (ICWS 2006), Washington, DC, USA, pp. 285–292. IEEE Computer Society, Los Alamitos (2006)
26. Recker, J., Mendling, J.: On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages
27. Johnson, J., Henderson, A.: Conceptual models: begin by designing what to design. *Interactions* 9(1), 25–32 (2002)
28. Bull, R.I.: Integrating dynamic views using model driven development. In: CASCON 2006: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research, p. 17. ACM, New York (2006)
29. Mendling, J., Simon, C.: Business Process Design by View Integration. In: Eder, J., Dustdar, S. (eds.) BPM Workshops 2006. LNCS, vol. 4103, pp. 55–64. Springer, Heidelberg (2006)
30. Keller, G., Nüttgens, M., Scheer, A.W.: Semantische Prozeßmodellierung auf der Grundlage “Ereignisgesteuerter Prozeßketten (EPK)”. *Arbeitsbericht Heft 89*, Institut für Wirtschaftsinformatik Universität Saarbrücken (1992)

31. Axenath, B., Kindler, E., Rubin, V.: An open and formalism independent meta-model for business processes. In: *Proceedings of the Workshop on Business Process Reference Models*, pp. 45–59 (2005)
32. Kindler, E., Axenath, B., Rubin, V.: AMFIBIA: A Meta-Model for the Integration of Business Process Modelling Aspects. In: *The Role of Business Processes in Service Oriented Architectures*. Number 06291 in *Dagstuhl Seminar Proceedings* (2006)
33. Wu, S., Sheth, A.P., Miller, J.A., Luo, Z.: Authorization and access control of application data in workflow systems. *J. Intell. Inf. Syst.* 18(1), 71–94 (2002)
34. Georgakopoulos, D., Hornick, M., Sheth, A.: An overview of workflow management: from process modeling to workflow automation infrastructure. *Distrib. Parallel Databases* 3(2), 119–153 (1995)

A Semantics-Based Aspect Language for Interactions with the Arbitrary Events Symbol

Roy Grønmo^{1,2}, Fredrik Sørensen¹, Birger Møller-Pedersen¹,
and Stein Krogdahl¹

¹ Univ. of Oslo, Dept. of Informatics, Norway

² SINTEF ICT, Norway

o o b t o o

Abstract. In this paper we introduce an aspect language that can define cross-cutting effects on a set of UML 2.0 sequence diagrams. The aspects and sequence diagrams are woven at the model level. By basing the weaving upon a formal trace model for sequence diagrams, we ensure that the weaving is semantics-based. We propose the *arbitrary events symbol* as a wildcard mechanism to express zero or more events on a sequence diagram lifeline. The approach is explained by a real-life example, and a weaving tool partially implements the approach.

1 Introduction

Aspect-orientation for programming has emerged as a promising way to separately define cross-cutting parts of programs, in order to achieve separation of concern. We believe that the same potential is there also for modeling. This paper explores aspect-oriented modeling for UML 2 sequence diagrams [14].

In aspect-oriented programming the *base program* is the main program upon which one or more aspects may define some cross-cutting code as additions or changes. An aspect is defined by a pair (*pointcut* and *advice*), where the *pointcut* defines where to affect the base program and the corresponding *advice* defines what to do in the places identified by the pointcut. Analogously we term our set of sequence diagrams as the *base model*, and we define an aspect diagram to consist of a pointcut diagram and an advice diagram, both based upon the concrete syntax of sequence diagrams.

In this paper we assume that the sequence diagrams are used to automatically produce executable test code, e.g. to test if a sequence diagram is a correct refinement of another sequence diagram [12], or to test if a system specified by UML statecharts, class diagrams and object diagrams is consistent with sequence diagram specifications [15]. Thus, we need to weave the aspect and the base model before generation of test code. The aspect diagram defines the cross-cutting model elements to influence the base model, so that an aspect weaver can produce a new model which is the base model woven with the advice.

The woven model is not intended to be viewed (except for debugging) or further updated by the modeler. This means that the structure of the result is not a primary focus. It suffices that the woven model is correct with respect to our formal model.

Many aspect-oriented approaches suffer because they rely on a pure syntactic pointcut matching and weaving. Syntactic-based matching has a problem because it does not

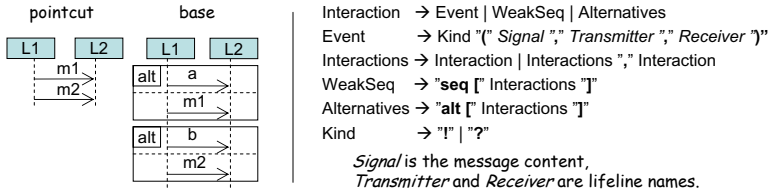


Fig. 1. Left: Syntactic-based matching problem, Right: Syntax for sequence diagrams

capture all the matches it conceptually should do. The pointcut of Figure 1 expresses that the message m from the lifeline L_1 to the lifeline L_2 is followed by the message m from L_1 to L_2 . The base model has two consecutive `alt` operators. An `alt` operator defines a choice of different alternatives, where the alternatives are given as operands separated by a dashed line. If we try to find matches of the pointcut within the base model with pure syntactic matching, then we do not find any matches. However, one possible execution trace chooses the second operands of the two `alt` operators, which then should result in a match of the specified pointcut.

Our main contribution consists of: 1) an aspect language which is sufficient to handle our case study, and 2) semantics-based matching and weaving. We devote much attention to the *arbitrary events symbol* that makes it possible to define pointcuts with an arbitrary number of events in specific positions. This symbol enables us 1) to make the pointcut more robust with respect to some kinds of changes in the base model, and 2) to define more relaxed matching criteria leading to additional matches in cases with irrelevant base model differences.

The paper is organized as follows; Section 2 introduces the STAIRS formal model for sequence diagrams in relation to our case study; Section 3 presents the aspect diagram language including the arbitrary events symbol; Section 4 defines the semantics-based matching; Section 5 explains how the weaving algorithm works; Section 6 compares our approach with related work; and finally Section 7 provides the conclusions and suggests future work.

2 Sequence Diagrams and STAIRS

STAIRS defines a formal model where the semantics of a sequence diagram is understood as a set of execution traces. The syntax of a UML sequence diagram, called *interaction*, follows the EBNF in the right part of Figure 1 [16]. We focus on the operators `seq` and `alt`. These two operators are chosen because they are the basic operators from which we also may define several other operators.

Each message is represented by two events, a *transmission* event (!) and a *reception* event () (the transmitter and receiver lifelines are omitted for readability when this information is unambiguously defined by associated diagrams). An event takes place on a lifeline L if it is a transmission event on L , e.g. $!s \quad L \quad L$, or a reception event on L , i.e. $s \quad L \quad L$. We require that the messages are complete (i.e. contain both events ! and) within each operand, pointcut, advice and each sequence diagram in the base model.

The weak sequence operator, s , of sequence diagrams imposes a partial order of events given by: 1) the transmission event must come before the reception event of the same message, and 2) all events are ordered for each lifeline in isolation. An intuitive idea behind this partial order is that messages are sent asynchronously and that they may happen in any order on different lifelines, but sequentially on the same lifeline. The o operator defines alternative interactions.

Sequence diagrams do allow *crossing messages*, i.e. two messages and are crossing only when they have events on the same two lifelines, and has an event before on one lifeline, while has an event before on the other lifeline, e.g. $s [!(a \ L1 \ L2) !(b \ L1 \ L2) ?(b \ L1 \ L2) ?(a \ L1 \ L2)]$.

Figure 2 shows an extract of a base model sequence diagram for the ICU (I see you) buddy system. Prototype implementations have been used as test cases within computer science courses at the University of Oslo. ICU is an SMS-based service where the users have access to different positioning services including the positioning of users who have accepted to be on the buddy list. The services have been partially specified by sequence diagrams. The sequence diagrams have been manually interpreted to produce conforming state machines for which Java code has been produced by the JavaFrame code generation framework [5]. Future versions may automate the production of state machines from sequence diagrams [10].

There are four diagram levels due to lifeline decomposition of which we will concentrate on the first two levels as this will be sufficient to explain our approach. The number of decomposition levels for a particular scenario is chosen by the modeler.

In the position user service (Figure 2), an SMS message is sent from Us to T , at level 1, to request the positioning of a buddy. The message is forwarded from T to the decomposed (indicated by the keyword s) lifeline sUs . In the decomposed diagram, at level 2, we see that the same message is received by the s lifeline. The s lifeline translates the message into an internal message, sUs , which is sent to the decomposed lifeline C . Its internal lifelines and the messages at level 3 and 4 are omitted from the illustration. The sUs message will finally reach the proper session object at level 4. The session object will initiate a message, s s , to the positioning service T via s s , to position the buddy. The fourth lifeline at level 2 without any messages in the figure, s , stores all the persistent data and provides querying services upon these data. The syntax representation for the position user service is shown in the middle part of Figure 2.

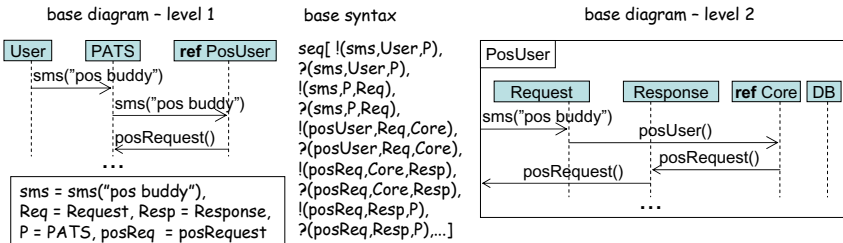


Fig. 2. Base model extract of the position user service

We briefly explain the semantics operator, \sqcup , while a precise definition is given in [16]. The semantics of an interaction i is $i \sqcup (p \sqcup n)$, where p is the set of *positive traces* and n is the set of *negative traces*. *Positive traces* define valid behavior and *negative traces* define invalid behavior, while all other traces are defined as inconclusive. In this paper we concentrate on aspects that only affect positive traces, and we therefore use a simplified model without negative traces: that is $i \sqcup p$. A *trace* is a sequence of events which we display as $\langle e_1 \dots e_n \rangle$, where e_i are events for all $i = 1 \dots n$.

The \sqcup operator produces one trace for each valid permutation of events that satisfy the two partial order requirements as explained for the seq operator above. The \sqcup operator produces the union of the traces for each operand of the \sqcup operator. Each message in the trace is dynamically given a unique identifier, which is shared between the transmission and reception events of the message.

We define one trace to be *partial order equivalent (POE)* to another trace if they are both permutations of the same set of events with the same order on each lifeline. The \sqcup operator is used to define POE since it is defined to produce all such permutations:

Definition 1. We say that two traces $t^A = \langle t_1^A \dots t_n^A \rangle$ and $t^B = \langle t_1^B \dots t_n^B \rangle$ are **partial order equivalent (POE)** if and only if:

$$t_1^A \leq t_n^A \quad \text{and} \quad t_1^B \leq t_n^B$$

We let the function, $\text{POE} : \mathcal{T} \times \mathcal{T} \rightarrow \mathcal{P}(\mathcal{T})$, calculate all the POE traces of a given trace, $\text{POE}(\langle t_1^A \dots t_n^A \rangle) = \{ \langle t_1^A \dots t_n^A \rangle, \langle t_1^B \dots t_n^B \rangle \}$. The pointcut diagram in Figure 1 has two traces which are POE: $\langle !m1 \ ?m1 \ !m2 \ ?m2 \rangle$ and $\langle !m1 \ !m2 \ ?m1 \ ?m2 \rangle$. The function POE of either of these two traces returns the set of both traces.

3 Aspect Diagram

The aspect diagrams are inspired by graph transformation [3] where the left part, the pointcut diagram, defines a pattern for which we are looking for matches or morphisms in the base model. The right part, the advice diagram, defines a replacement of the matches within the base model. This implies that messages present only in the pointcut and not in the advice, will be deleted, while messages present only in the advice and not in the pointcut, will be added. Both the pointcut and advice diagrams are based upon the graphical elements of sequence diagrams so that the modeler can think in terms of an already familiar notation.

Property names (lifelines, messages or message parameters) to be matched in the pointcut may be defined by its full name or by a mixture of fixed characters and the wildcard symbol $*$. Identifier variables may be shown explicitly in the diagram as part of the property name: d m . The d will be bound to the actual identifier of a matching element. Ids that are repeated in the advice must be placed on an identical kind of element, so it is not allowed to repeat the id from a pointcut message in the advice and to change either the message name, transmission or reception lifeline. Repeated elements in the advice may use the full form d m , where all parts need to be identical from the pointcut, or we may as a shorthand omit either the

id part or the property name. We allow to use messages with signatures equivalent to those defined in AspectJ [7] where \ast means an arbitrary number of parameters.

The pointcut diagram is restricted so that it can only use events and the `s` operator. We avoid the `and` operator in the pointcut since it is not obvious how to combine this with the advice. To avoid possible confusion for the aspect modeler, we require that such alternatives are modeled explicitly by several aspect diagrams instead of a single, more compact aspect diagram.

We continue the presentation of the aspect diagram language in two subsections. First we show an aspect diagram example for our case study, and then we introduce the *arbitrary events symbol*.

3.1 Example

We will now investigate the aspect model example in Figure 3 which shall ensure that the user is registered in the ICU system before using the available services. This is a cross-cutting functionality since it shall be applied to all the ICU services (including the position user service in Figure 2), except the register user service. So instead of augmenting all but one of the sequence diagrams at four levels for each of the services (about 10 services have been used in the ICU system), we would like to apply a single aspect definition for easier specification and maintenance.

The aspect diagram will follow the same decomposition principles of ordinary sequence diagrams, so that we may define a pair of pointcut and advice for each level in the base model. In the figure we only show the first two levels of aspect diagrams (as we did with the base model).

The pointcut at level 1 defines that we are looking for matches of all incoming SMS messages with any content (wildcard \ast) except for the service to register the user. The exception is defined as a *negative pointcut* (called Negative Application Condition (NAC) in graph transformation) where the `sms` identifier of the pointcut is reused from the ordinary pointcut to restrict the possible matches. There may be an arbitrary number of negative pointcut diagrams associated with each level of pointcut diagrams. Pointcut matches are excluded if at least one negative pointcut also provides a match.

The two messages identified by the pointcut (`sms` and `sms`) are repeated in the advice and will thus remain in the result. The advice contains an `and` operator which shall be added to the base model.

We propose a new operator, *insertRest*, which in the example is placed in the `s` operand of the `and` operator. The *insertRest* operator defines that the remaining part of the base model shall go into the `and` operand, overriding the default behavior of being placed after the full `and` operator. An *insertRest* operator must span over all lifelines and may be placed anywhere in the advice. Multiple *insertRest* operators are allowed in the advice.

Notice that there are two messages to be added by the advice diagram at level 2, `sUs` and its reply, which were not visible at level 1 since they are only between internal lifelines. Wildcard matching symbols \ast are used in all places where the base sequence diagrams have different values to be matched: the content of the incoming SMS messages (`sms` and `sms`), the name of the decomposed lifeline (\ast), and the internal message (`s v`) initiated by the SMS message.

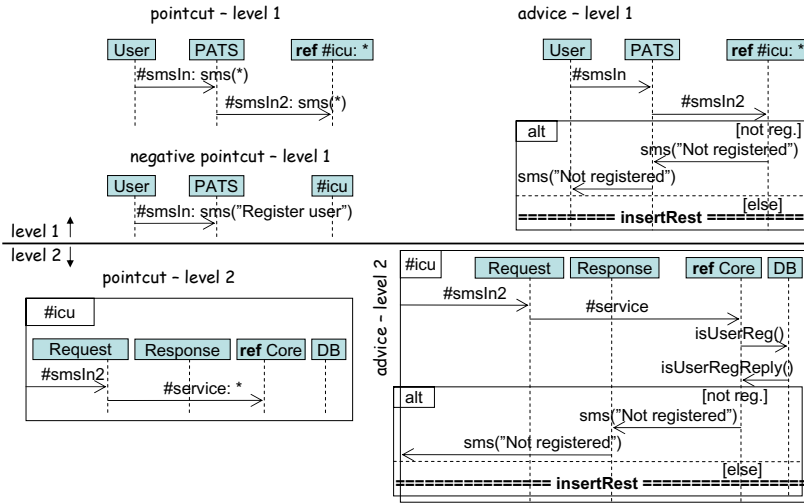


Fig. 3. Aspect: Check User Registration

Our aspect diagrams follow ordinary UML decomposition rules. The of the advice introduced at level 1 that spans over the decomposed lifeline, must be repeated at all the lower decomposition levels. A reception event on a decomposed lifeline has the decomposed diagram frame, in the next level, as its source, and one of its internal lifelines as its target, e.g. `! sms T)` at level 1 corresponds with `! sms s)` at level 2. A transmission event on a decomposed lifeline has the decomposed diagram frame, in the next level, as its target, and one of its internal lifelines as its source, e.g. `! sms s d) T)` at level 1 corresponds with `! sms s d) s s)` at level 2. The ordering of events on the decomposed lifeline must be maintained within the decomposed diagram, e.g. `! sms` must come before the `operator` both at level 1 and 2.

3.2 Arbitrary Events Symbol

Our pointcut in Figure 3 defines that there are some events that have to happen directly after each other: `! sms)` followed by `! sms)` on the `T` lifeline at level 1, and `! sms)` followed by `! s v)` on the `s` lifeline at level 2. These positions in the pointcut are fragile with respect to base model evolutions. Consider that we decide to send a request confirmation back to the user directly after each incoming SMS message on the `T` lifeline. Such a base model change may be registered directly in the base model or as another aspect. In either case there will be an intermediate event in between `! sms)` and `! sms)` which unintentionally prevents our aspect to be applied.

The two aspects (and) in the left part of Figure 4 share the same pointcut, and application of one aspect will prevent application of the other aspect at the same position.

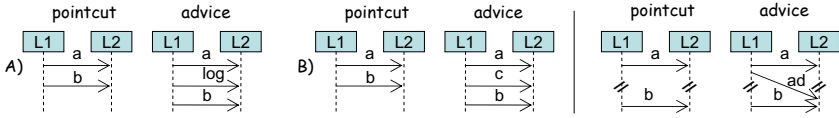


Fig. 4. left: Mutually exclusive aspects, right: The arbitrary events symbol

In many cases it is desirable to allow pointcut matches even with intermediate events, while in other cases such intermediate events should not be allowed.

Klein et al. [8] suggest that the matching strategy of allowing intermediate events or not should be a user-configurable property of the matching tool, and will thus not be visible in the aspect diagrams. We propose on the other hand that the matching strategy is explicitly defined as part of the pointcut with the *arbitrary events symbol* (!). This provides the benefit that we may easily define how to merge the intermediate events with the advice inserted events. We may also use different matching strategies within the same aspect. In between two events we may forbid other events, while for other event pairs we may allow intermediate events.

The arbitrary events symbol is placed on the lifeline of a pointcut or advice diagram to indicate the presence of an arbitrary number of events (including zero events). An arbitrary events symbol used in the pointcut has to be preserved from the pointcut, stay on the same lifeline and remain in the same order relative to the other arbitrary events symbols in the advice diagram. Due to this restriction we do not need identifiers for the arbitrary events symbols, but we will use the symbol !_i^L to denote the i 'th arbitrary events symbol on a lifeline L numbered from top to bottom. An arbitrary events symbol belongs to a specific lifeline, and it cannot be placed on a decomposed lifeline.

The pointcut in the right part of Figure 4 defines that we are looking for matches of an $\text{!}_1^{L_1}$ message followed by a $\text{!}_2^{L_2}$ message, and the arbitrary events symbol used on both the lifelines indicate that there may be arbitrary events in between $\text{!}_1^{L_1}$ and $\text{!}_2^{L_2}$ on lifeline L_1 and between $\text{!}_1^{L_2}$ and $\text{!}_2^{L_2}$ on lifeline L_2 . The corresponding advice adds an ad message with an explicit position relative to the arbitrary events. The transmission event of ad , $\text{!}_1^{L_1} \text{ad}$, shall be inserted directly after the $\text{!}_1^{L_1}$ event (and before all the arbitrary events) on lifeline L_1 , and the reception event, $\text{!}_2^{L_2} \text{ad}$, shall be inserted directly before the $\text{!}_2^{L_2}$ event (and after all the arbitrary events) on lifeline L_2 .

Figure 5A shows an illegal aspect since the arbitrary events symbols in the pointcut are not preserved in the advice. We do not allow to delete the arbitrary events since this may be harmful, and it may also produce illegal sequence diagrams where messages do not have both a transmission and reception event. We do not allow directly consecutive arbitrary events symbols in the pointcut such as in Figure 5B since this would be redundant. This disallowed redundancy case for pointcut diagrams of Figure 5B may however occur in the advice and still be allowed when explicit pointcut messages are deleted. Figure 5C illustrates that the arbitrary events symbol represents events and not messages and it is thus meaningful to specify that $\text{!}_1^{L_1}$ and $\text{!}_2^{L_2}$ may have arbitrary events in between, while $\text{!}_1^{L_2}$ and $\text{!}_2^{L_2}$ cannot have any events between them.

Since we have introduced the arbitrary events symbol and the `insertRest` operator, we need to extend the syntax for sequence diagrams. Both the pointcut and advice diagrams get additional EBNF clauses (the `insertRest` operator is only allowed for advice

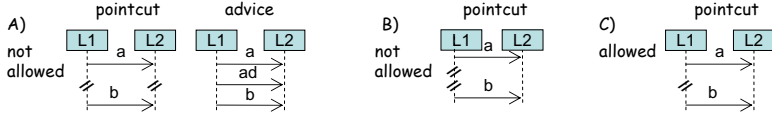


Fig. 5. Rules for the arbitrary events symbol

diagrams):

$$s \quad s \quad [^L a \ ?a] \ !ad \quad L^1 \ s \quad L^2 \ ?ad \quad !b \ ?b$$
The arbitrary events symbol has a superscripted L to indicate its owner lifeline. A possible syntax representation for the advice diagram in the right part of Figure 4 is:

$$s \quad s \quad [^L a \ ?a] \ !ad \quad L^1 \ s \quad L^2 \ ?ad \quad !b \ ?b \quad (1)$$

Notice that there are multiple alternatives for the syntax representation, since all permutations that follow the partial orders of each lifeline are valid (and the s operator may be nested arbitrarily).

4 Semantics-Based Matching

In order to make the matching semantics-based, we define matches directly on the base model traces. Remember that all messages are given unique identifiers which are shared between the transmission and reception events of the message, meaning that all the events in the pointcut trace have different identifiers from the events in the base trace. We need an injective mapping function, $\gamma : v \rightarrow v$, which maps from pointcut events to base events. For each event, γ only maps the identifier, while it preserves all the other event properties (kind, signal, transmitter, receiver). The γ mapping function will be one-to-one between the match part of the base trace and the pointcut trace.

Definition 2. For a pointcut without any arbitrary events symbols we have a **match** if and only if a base trace contains a pointcut trace, where each event in the pointcut trace is mapped by

In theory we may calculate all the pointcut and base traces to find matches. In practice this is an intractable problem since the number of traces may have an exponential growth relative to the number of events in the diagram. In our first test implementation we were not able to handle a relatively small base model, consisting of eleven consecutive messages in the same direction between the same two lifelines, since there are as much as 58 786 traces.

In an optimized weave algorithm we avoid calculating all the traces by instead working on the POE (last paragraph of Section 2) equivalence classes (abbreviated as *POE classes*) instead. This has a large impact on the performance since a POE class may represent thousands of actual traces, e.g. all the 58 786 traces in the base model mentioned above belong to the same POE class. The maximum number of POE classes for an interaction is equal to the total number of operands. For each POE class, we only represent the event orders per lifeline, so that each event occurs only once. An interaction may be defined as a collection of POE classes. The next lemma states that a

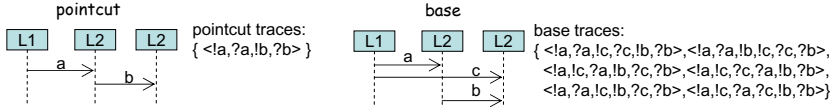


Fig. 6. is a match blocking message

lifeline-based matching wrt. to each POE class is sufficient to identify all the possible matches:

Lemma 1. (Lifeline-based matching) *For a pointcut without any arbitrary events symbols and a base trace : There exists a match in one of its POE traces () if and only if*

1. *l i in : the event order on l of contains the event order on l of the pointcut (where each event in the pointcut is mapped by) AND*
2. *there are no messages in having the reception event before the contained pointcut on one lifeline and the transmission event after the contained pointcut on another lifeline (match blocking messages).*

A proof of Lemma 1 is given in [4]. Lemma 1 needs to exclude match blocking messages. Otherwise the if-direction of the lemma does not hold as we can see from Figure 6. The pointcut has a single trace: $\langle !a, ?a, !b, ?b \rangle$. None of the six shown base traces have a contained pointcut trace, and thus there are no matches (Def. 2). This is because the match blocking message will always get its two events between the first and last events of the matched pointcut trace.

The decomposed lifelines and diagrams at different levels, as we had within the position user service (Figure 2), do not need any special treatment when we work with traces. This is because the trace events represent a flattened structure where the decomposed lifelines are not present, only lifelines with transmission and reception events. For the $\langle !s, s \rangle$ event in Figure 2, the event is shown at both level 1 and level 2, but there will only be a single event represented in the trace, i.e. $\langle !s, s \rangle$ (s s T).

4.1 Matching with the Arbitrary Events Symbol

As opposed to decomposition of base models and aspect models, the arbitrary events symbol needs special treatment in the matching process. Since the arbitrary events symbol is a lifeline-based mechanism, it fits nicely with the lifeline-based matching. The event order per lifeline of the pointcut can simply be extended to include the arbitrary events symbol. Then the arbitrary events symbol represents a wildcard of an event list with zero or more arbitrary events, and the lifeline-based matching will work fine also for arbitrary events symbols.

Figure 7 shows a pointcut expressing that we are looking for an message followed by a message, where there may be an arbitrary number of events in between the $\langle !$ and $\langle !$ events and between the and events. Base model 1 will have a match, where the arbitrary events symbols have the matches: $L1 \ ?d \rangle$ and $L2 \ ?c \ !d \rangle$.

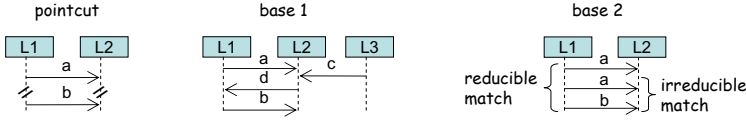


Fig. 7. Matching

The base model 2 in Figure 7 shows a base model where we have two overlapping matches, one with both messages (*reducible match*) and the other one with only a single message (*irreducible match*). In the reducible match, the second message matches arbitrary events symbols. It seems more appropriate and intuitive to choose the match with only a single message.

Definition 3. A match $m_{min} = m_1 \dots m_n$ is an *irreducible match* if there is no subsequence of m_{min} which is a match.

The irreducible match definition can be easily translated also to the lifeline-based matching by saying that there shall not exist a subsequence on any of the lifelines, which also constitutes a match in combination with the other lifelines. Reducible matches will only occur in cases where the arbitrary events symbol is used. In our matching we require that the matches are irreducible.

5 Weaving

The previous section showed that a lifeline-based matching of the POE classes is equivalent to a semantics-based matching on the traces. This section continues by defining a lifeline-based weaving.

We calculate the POE classes of the base model, the single POE class of the pointcut and the POE classes of the advice. Since the pointcut is restricted to use only `s` and events, it has always only one POE class. The POE classes are derived from an interaction by configuring all combinations of operands into (potentially) different POE classes.

The weave algorithm repeats the following three steps as long as there are unhandled matches in the base POE classes: 1) Identify a match in a base POE class (lifeline-based matching), 2) Perform lifeline-based weaving, according to Def. 4 below, for each of the advice POE classes. Add the results, a new POE class for each advice POE class, to the set of base POE classes, and 3) Remove the matched base POE class and repeat the three steps if there are more matches.

Definition 4. *Lifeline-based weaving* for a matched base POE class $(\langle \text{basePLls} : m(l) \rangle)$ with match match and an advice POE class $(\langle \text{advPLls} : res \rangle)$. The resulting POE class, $\langle \text{weavePLls} : res \rangle$, gets the initial value: $\langle \text{basePLls} : m(l) \rangle$. Then the lifelines of weavePLls are updated according to three rules:

- (1) $l \in \text{basePLls} : m(l) \rightarrow res \text{ replaceEvts}(l \text{ match} \text{ advPLls}(l))$
- (2) $l \in \text{advPLls} : res \rightarrow res \text{ addLL}(l \text{ advPLls}(l))$
- (3) $l \in \text{basePLls} : (m(l) \rightarrow res \text{ ins}(l \text{ advPLls}(l)))$

Prerequisites of Def. 4: A POE class contains the following methods; $LLs()$ retrieves the set of (non-empty) lifelines; $vs(m)dv(vs)$ replaces the match events by the advice events on lifeline; $dv(vs)$ retrieves the list of events of the advice on lifeline; $ddLL(dv, vs)$ adds as a new lifeline with the advice events on as the content; $s(dv, vs)$ inserts the advice event list on lifeline into an appropriate position on lifeline (the details are given below). $m()$ retrieves the event list of the match on the lifeline, and \rangle denotes an empty event list.

Explanation of Def. 4: Each lifeline can be woven separately, as defined by three mutually exclusive rules. When a lifeline has matched events, **rule (1)**, then the matched events on this lifeline are simply replaced by the corresponding advice events (in some cases an empty list). When a lifeline has events in the advice and not in the base, **rule (2)**, then all of this advice lifeline is inserted as a new base lifeline.

The most difficult rule, **rule (3)**, is when a lifeline has no matched events, but have events in both the base and advice, e.g. the dv event in the advice of Figure 8 occurs on lifeline L with no events in the pointcut (the match part), and there is a $!$ event on the L lifeline in the base model. *Should the new event be placed before or after the event?* Choosing to place dv before $!$ will produce the undesired woven diagram (Figure 8) which has no possible traces because there is a deadlock.

In many cases, a proper placement can be found by exploring the partial order relationships. Let $e_1 e_2$ denote a partial order where the event e_1 must happen before the event e_2 . We will produce the union of the partial orders of the advice POE class and the matched base POE class:

$$po(!a ?a) po(!adv ?adv) po(!b ?b) po(?a !adv) po(?b ?a)$$

Since partial order is a transitive relation, we may calculate the transitive closure, which will produce the pair $! dv$. This defines a unique and proper position for dv on the base L lifeline in Figure 8. There are however cases, where there may be several position choices fulfilling the partial order requirements, e.g. add another event $! L$ after $!$ on L . In such cases we choose an arbitrary position among the choices except that we will avoid or minimize the number of crossing messages, and provide a warning message to the modeler.

In our advice at level 2 in Figure 3 there are four events on the a and s lifelines. The pointcut has no events on these lifelines, but the base model does. However, all these four events get unique positions when calculating the transitive closure of the partial order relation (due to limited space the base model details to show this is not included in the paper).

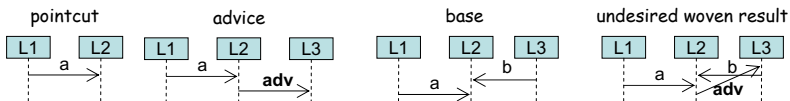


Fig. 8. Placement of a new event on a lifeline with no events in the pointcut

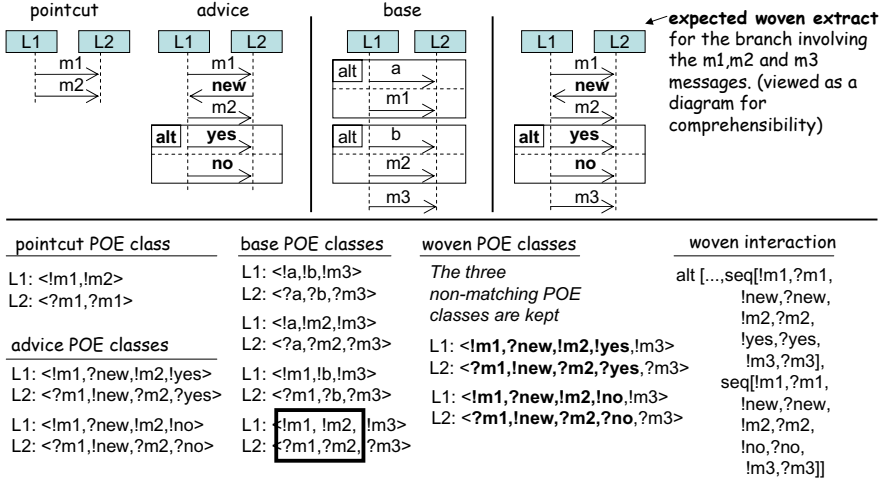


Fig. 9. Trace-based weaving on POE classes

When there are no more unhandled matches, the woven result is a set of POE classes. Finally, we need to go from POE classes to a woven interaction. Each POE class is represented by a single s operator with the lifeline events as operands in one of the legal orders (the choice is insignificant). Then all these s operators are used as operands inside an outermost alt operator to represent the woven interaction.

Figure 9 shows the weaving of an aspect and a base model. The aspect defines a replacement of all occurrences of two consecutive messages m_1 and m_2 , by an advice which adds the message new and an alt operator. Each POE class is represented by its event order on each of the two lifelines L_1 and L_2 . In addition to the single pointcut POE class, there are two advice POE classes and four base POE classes. The only base POE class with a match (marked by rectangles) is woven for each of the two advice POE classes, resulting in two new POE classes which adds up to a total of five POE classes. The weaving terminates since there are no more matches. The final woven interaction is shown in the bottom right part of the figure.

5.1 Weaving of the Arbitrary Events Symbol and the Insertrest Operator

To handle the arbitrary events symbols in the weaving, we need to bind these symbols to actual events relative to the match. L_i will then hold an event sequence for the i 'th arbitrary events symbol on the L lifeline, and these event sequences will replace the corresponding arbitrary events symbols in the advice so that the advice will have ordinary event sequences for all its lifelines.

Each insertRest operator in the advice will be replaced by an event order per lifeline. For each lifeline the event order will be the remaining subsequence of the base model events after the match part. Alternatively we may describe this relative to the general match definition and for a given match $m_1 \dots m_n$ within a base trace, the insertRest events will be:

$$b \quad \langle \quad m_1 \quad m_n \quad \rangle$$

t t

These insertRest events are distributed to event orders per lifeline for the lifeline-based implementation.

We have a tool implementation of the basic matching and weaving approach described in this paper. The tool uses an Eclipse-based SeDi sequence diagram editor v.1 [11] to define base, pointcut and advice diagrams. The weaving has been verified to behave correctly on our test examples, by manually investigating the woven textual interactions. We are currently implementing a translation from textual interactions to graphical diagrams for easier manual validation purposes. Future work is to extend the tool to also support the arbitrary events symbol, decomposition (introduced in SeDi v.2) and the insertRest operator.

5.2 Discussion

Consider the aspect and base model example in Figure 10. The aspect defines that two consecutive messages should be replaced by a message, and the base model contains four consecutive messages. Without using ids and the injective mapping function in the match, we could mistakenly choose a match which does not pair the correct transmission and reception events (shown as black boxes in the figure). By matching the last two events on the L lifeline and the first two events on the L lifeline, we get a final woven result with a crossing message (notice that the message is a match blocking message for the two remaining messages). Crossing messages are allowed in general, but it is unexpected and undesired in this case.

We have described the matching strategy as a *random matching*. Find any match, perform weaving and repeat the process. If our weaving terminates, then we are guaranteed that there will not exist any matches in the woven model. With the aspect and base models in Figure 10, a random matching strategy gives one of the following three alternative derivations with two different end results: 1) , 2) , and 3) . Klein et al. [9] suggest a *left-most matching* strategy leading to the unique derivation alternative 3. Our weaving also supports the left-most matching by ensuring that we always choose the top-most matches of each lifeline.

We define a *plain additive aspect* to be an aspect that does not delete events. For such aspects we will mark all the events in a treated match and exclude them from

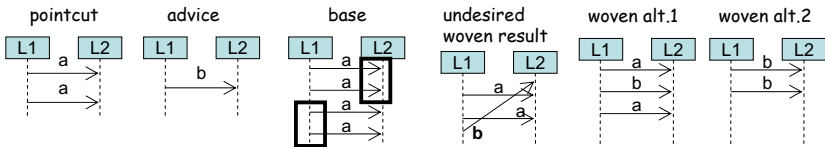


Fig. 10. Incorrect match leads to undesired weaving / Alternative woven results

possible future matches. This ensures a terminating weaving process for a lot of aspects that would otherwise never terminate, e.g. (shorthand notation for an aspect: dv).

6 Related Work

In this paper we have restricted the base model to use only the s and op operators. However, the results are directly applicable to other operators that can be defined with s and op , e.g. opt (optional), par (parallel), and the $loop$ operator for loops with an upper bound. The seq operator is not supported. It represents a strict sequence of events also across lifelines, which is in strong contrast to our approach.

We have reported complementary work in [4], where we perform a static weaving on a finite structure even for many typical loops without upper bounds. This paper goes beyond [4] by providing details of the aspect language including property matching, identifiers, negative pointcuts, decomposition, the s s operator and the arbitrary events symbol.

The pointcut model in AspectJ [7] cannot express matching based on a sequence of events, which is necessary to encounter the problem of syntactic-based matching described in this paper. QVT [13] is a model-to-model transformation language which supports general source and target MOF-languages. Since we address transformations where the source and target is the same language, UML sequence diagrams, we benefit from making tailored constructs and enabling the user to work on the more intuitive concrete syntax.

The identifiers, property name matching, and the arbitrary events symbol are inspired by Join Point Designation Diagrams (JPDD) [18] proposed by Stein et al. We have modified the notation slightly and introduced advice diagrams since JPDD only covers pointcut diagrams. JPDD is intended for mapping to aspect-oriented programming languages such as AspectJ, as opposed to our model matching and weaving.

Deubler et al. [2], Solberg et al. [17], and Jayaraman et al. [6] all define syntactic-based approaches for sequence diagrams. Deubler et al. match single events only and provide no model weaving or mapping to a concrete aspect language. Solberg et al. rely on binding models instead of a generic matching pattern (as in our approach), to identify the base model elements to be affected by the aspects.

Klein et al. [8,9] perform a semantics-based weaving of sequence diagrams by using automata representations. They present four different matching choices which is defined to be a tool-specific configuration, and where a single matching strategy applies to the entire pointcut diagram. We support the *general part* (by using the arbitrary events symbol) and *enclosed part*.

Klein et al. [8] support wildcard matching of events. The important difference between their approach and our proposed arbitrary events symbol, is that they have no explicit graphical element for this, but define a matching strategy outside of the aspect diagram. Their approach have two drawbacks compared to our arbitrary events symbol: 1) their wildcard matching applies in all or no positions within the entire diagram, 2) it is a tool choice (or a global choice) how to merge the wildcard events with the aspect added events.

Decomposed lifelines and aspect diagram definition over multiple levels is something we have not seen in any related work. Decomposition for aspect diagrams allows the modeler to work with smaller units in isolation. Furthermore, decomposition does not introduce any added complexity since this syntactic arrangement is not visible at the trace level where our matching and weaving is performed.

Avgustinov et al. [1] have a trace-based run-time matching of events to execute some extra code when a match occurs. Since this happens during run-time and not statically as in our approach, the aspects are restricted to additive parts that are inserted entirely after the already executed match part. While performance is a major issue in run-time weaving, our weaving is static and termination within reasonable time is sufficient.

7 Conclusions

We have proposed an aspect language for UML 2.0 sequence diagrams. Aspect diagrams in terms of pointcut and advice diagrams use the same graphical elements as sequence diagrams, with only a few extensions, thus providing a familiar notation to sequence diagram users. The advice diagram replaces matches of the pointcut diagram, which can simulate traditional aspect-oriented mechanisms like *before*, *after*, *replace* and *around*.

Syntactic-based matching will fail to match all the intended base model joinpoints even for simple pointcuts as two consecutive messages. Our aspect language is therefore based upon a formal trace model (STAIRS) for sequence diagrams, and thereby on semantics-based matching. Matching is defined on traces of the base model traces. For performance reasons we have established a semantically equivalent implementation to plain trace-based matching, which works on partial order equivalent classes representing sets of traces.

The arbitrary events symbol is a powerful extension that allows to match an arbitrary number of events on a lifeline. This symbol allows to define flexible and robust pointcut definitions, and also to define how the additive parts of an aspect shall be positioned in relation to these arbitrary events. This mechanism may be useful in the context of multiple aspects or base model evolution which could otherwise unintentionally prevent matches of a pointcut.

As future work we plan to investigate more real-life scenarios to see if the expressiveness of the proposed language is enough, or if additional constructs are needed. We are also investigating confluence and termination properties of the proposed aspect language.

Acknowledgment. The work reported in this paper has been funded by The Research Council of Norway, grant no. 167172 V30 (the SWAT project).

References

1. Avgustinov, P., Tibble, J., de Moor, O.: Making trace monitors feasible. In: The 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA (2007)
2. Deubler, M., Meisinger, M., Rittmann, S., Krüger, I.: Modeling Crosscutting Services with UML Sequence Diagrams. In: Bruehl, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844. Springer, Heidelberg (2006)

3. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: *Fundamentals of Algebraic Graph Transformation*. Springer, New York (2006)
4. Grønmo, R., Sørensen, F., Møller-Pedersen, B., Krogdahl, S.: *Semantics-based Weaving of UML Sequence Diagrams*. In: *International Conference on Model Transformation (ICMT)* (in press, 2008)
5. Haugen, Ø., Møller-Pedersen, B.: *JavaFrame: Framework for Java-enabled modelling*. In: *Ericsson Conference on software Engineering (ECSE)* (2000)
6. Jayaraman, P., Whittle, J., Elkhodary, A.M., Gomaa, H.: *Model Composition in Product Lines and Feature Interaction Detection Using Critical Pair Analysis*. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) *MODELS 2007*. LNCS, vol. 4735, Springer, Heidelberg (2007)
7. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: *An Overview of Aspect*. In: *The 15th European Conference on Object-Oriented Programming* (2001)
8. Klein, J., Fleurey, F., Jézéquel, J.-M.: *Weaving multiple aspects in sequence diagrams*. *Trans. on Aspect Oriented Software Development III* (2007)
9. Klein, J., Héliouët, L., Jézéquel, J.-M.: *Semantic-based weaving of scenarios*. In: *The 5th International Conference on Aspect-Oriented Software Development* (2006)
10. Krüger, I., Grosu, R., Scholz, P., Broy, M.: *From MSCs to Statecharts*. In: *International Workshop on Distributed and Parallel Embedded Systems (DIPES 1998)* (1999)
11. Limyr, A.: *Graphical editor for UML 2.0 sequence diagrams*. Master's thesis, Department of Informatics, University of Oslo (2005)
12. Lund, M.S.: *Operational analysis of sequence diagram specifications*. PhD thesis, Department of Informatics, University of Oslo, Norway (2008)
13. OMG. *MOF QVT Final Adopted Specification*, OMG Document: ptc/05-11-01 (November 2005)
14. OMG. *UML 2.0 Superstructure Spec.*, OMG Adopted Spec. ptc/03-08-02 (August 2003)
15. Pickin, S., Jard, C., Jéron, T., Jézéquel, J.-M., Traon, Y.L.: *Test Synthesis from UML Models of Distributed Software*. *IEEE Trans. Software Eng.* 33(4) (2007)
16. Runde, R.K., Haugen, Ø., Stølen, K.: *Refining UML interactions with underspecification and nondeterminism*. *Nordic Journal of Computing* 2(12) (2005)
17. Solberg, A., Simmonds, D., Reddy, R., Ghosh, S., France, R.B.: *Using Aspect Oriented Techniques to Support Separation of Concerns in Model Driven Development*. In: *29th Annual International Computer Software and Applications Conference (COMPSAC)*, Edinburgh, Scotland (2005)
18. Stein, D., Hanenberg, S., Unland, R.: *Join Point Designation Diagrams: a Graphical Representation of Join Point Selections*. *International Journal of Software Engineering and Knowledge Engineering* 16(3), 317–346 (2006)

Model-Driven Platform-Specific Testing through Configurable Simulations

T. Kuhn and R. Gotzhein

Networked Systems Group, University of Kaiserslautern
{kuhn, gotzhein}@informatik.uni-kl.de

Abstract. The increasing size and complexity of software systems requires sophisticated testing methodologies. Since platform limitations could void the results of test suites, especially the embedded systems domain requires testing methodologies that also consider hardware resources. Current approaches for specifying and executing test cases, e.g. approaches based on the UML testing model, do not support the concept of platform testing very well. Deployment descriptions are only used for documentation. In this work, we address this problem by providing an extension to the UML testing profile that covers the modeling of platform testing models, which include realistic deployments, and by *C-PartsSim*, our simulation tool that is capable of executing these platform testing models. With *C-PartsSim*, it is possible to connect specialized simulators at runtime, thereby providing a configurable, platform-specific system simulation for testing. We describe the automatic transformation from scenario models to a tailored simulator instance for a concrete testing scenario and present case studies to demonstrate the flexibility and accuracy of our approach.

1 Introduction

Test-driven development methodologies, which propagate testing through the whole software development process, have shown their beneficial impact on software quality in several research studies [1]. Especially in model-driven development processes, the UML testing profile [2] is used to create testing models together with the specification of software systems. This way, early testing of specifications is supported, which results in earlier defect detection and therefore lower development costs [3]. However, modern embedded software systems are more and more deployed on hybrid hardware platforms. From real-world experiments, it is evident that platform resources can have a significant impact on the behavior of a software system after deployment [4] [5]. Especially when high-level software models are deployed on embedded sensor nodes, there is a risk that the modeled behavior is altered significantly due to platform constraints. Therefore, we propose platform-specific testing already in early stages of the development process. We distinguish between *platform testing*, which tests the correctness of a design model on a hardware platform with respect to defined test cases, and *performance evaluation*, which evaluates the performance of a design model together with (simulated) hardware devices in defined scenarios.

Current testing methods including the UML testing profile provide only limited support for platform testing. Only generic UML deployment diagrams are supported by the UML testing profile; these diagrams are not sufficient to provide a deployment specification that would be usable for a wider set of testing tools. Therefore, current simulation tools must be configured using their own scripting language; for example, the network simulator 2 (ns-2) [6] is configured using the OTCL scripting language.

Although there exist specialized simulators for certain system aspects that can be used for platform testing, only some hardware resources are simulated accurately by these simulators. Our first contributions to address and improve this situation were the simulation tools *ns+SDL* [5] and *PartsSim* [4]. Both tools connect a runtime environment, which is capable to execute models of software systems by specialized simulators, thus supporting early platform testing. The simulator *ns+SDL* was developed to support performance evaluation of software systems (e.g. communication protocols) that were specified with the SDL [17] specification language, by connecting an SDL runtime environment to the network simulator *ns-2* [6]. *PartsSim* adds *simulator components* for realistic platforms, which have been extracted from existing platform simulators (e.g. from the Avroa [9] simulator).

A drawback of both *ns+SDL* and *PartsSim* is that the interfaces and connections between simulator components are hard-coded [4]. Consequently, the integration of new simulator components requires simulator modifications. For a simulation run, the design models of the System under Test and of the test components are loaded by the simulators, as they would be loaded into existing devices in a physical deployment. In this way, it is possible to simulate system deployment on a specific target hardware quite accurately. Both tools have shown their applicability in various simulation studies [4] [5]. However, due to the hard-coded simulator interfaces and connections, their usability is limited.

To improve this situation, we have devised a simulation framework that incorporates *simulator components* into the testing process dynamically. Our solution consists of two parts, the UML platform testing profile i.e., a UML profile extending the UML testing profile, and a new, configurable version of *PartsSim* called *C-PartsSim*. The UML platform testing profile provides a generic approach to model testing scenarios and deployments. The specialization of the UML testing profile ensures conformance to a standard methodology for modeling testing scenarios. A front-end to *C-PartsSim* reads conforming test configurations from test contexts and dynamically creates a simulation script for every test case. The generated simulation script couples simulator components according to the model and generates timed events, thus creating a tailored, scenario-specific simulator instance.

The remaining part of this paper is structured as follows: Section 2 surveys related work. Section 3 documents our UML platform testing profile. Section 4 describes *C-PartsSim*. Section 5 identifies possible ways to extend *C-PartsSim* through the definition of new testing components. Section 6 reports on experiences of applying our framework, demonstrating the feasibility of our approach as well as the accuracy of the simulation results. Section 7 draws conclusions and lays out future work.

2 Related Work

There is a large body of work on simulators for platform testing or for performance evaluation. In this section, we focus on approaches that test SDL or UML models in combination with simulated hardware.

The work presented in [19] proposes a set of stereotypes as extensions to UML models of software systems. These stereotypes enable modeling of resource usage and temporal constraints. The stereotyped models are then used as input for a transformation that automatically generates models for the network simulator OpNet [7], which is capable of simulating wired and wireless networks. It expects its input, software systems, and simulated devices as set of finite state machine models. The approach presented in [19] enables performance testing of UML models to the extent that is possible with the OpNet simulator. It is not necessary to re-specify the tested system specifically for the simulation. The drawback of this approach is that only UML models that map to the finite state machines of OpNet are supported. In particular, no platform resources can be simulated with this approach, and no network simulators other than OpNet can be used.

The Fraunhofer Institute for integrated circuits has devised a simulator coupling infrastructure [10], which is capable of coupling Matlab/Simulink, ModelSim, User Mode Linux and VHDL simulations together. The *ns-2* network simulator controls the simulation. This simulator coupling infrastructure supports the direct execution of software systems without raising the need to re-specify its behavior for a specific simulator. The structure of the simulated system is fixed, every node is simulated by one simulator, and all nodes are connected to a simulated network. While this approach is powerful enough for simulating a number of common networking scenarios, the fixed structure limits its applicability for scenarios with multiple networks, and for scenarios that require the simulation of non-networking devices. Furthermore, no platform simulation is provided by the simulation framework presented in [10].

UMLSim [11] utilizes User Mode Linux to load and execute simulated applications. UMLSim does not take UML models as input, but regular compiled Linux applications, which could be generated from UML models using available tools. Therefore, the same executable can be used for simulation and for deployment to real hardware. This raises the credibility of simulations, with the drawback that only Linux applications can be simulated. The only supported communication between applications is a TCP/IP network. No accurate platform timing is supported. This limits the applicability of this approach with regard to platform testing and the embedded systems domain.

SDL2SPEETCL [22] provides transformations of SDL models to the performance evaluation library SPEETCL [21]. This enables performance evaluation of SDL models to the extent that is possible with SPEETCL. SPEETCL simulates no specific network hardware; networks are simulated with the help of stochastic simulation components that introduce errors according to a defined distribution. Traffic generators are supplied to simulate typical traffic streams like web-, speech-, and ftp-traffic.

The Network Simulator for SDL systems *ns+SDL* [5] links SDL systems with the well-known network simulator *ns-2* [6]. The *ns-2* provides accurate simulation of a broad range of network hardware. The same compilers are used to transform the SDL

system into an executable for both simulation and deployment, which raises the credibility of simulation results. The *ns+SDL* does not support the simulation of platform resources, therefore, this approach is also limited in its applicability with respect to platform testing of embedded systems that run on platforms with low computational and memory resources.

In summary, there have been several efforts towards platform testing of SDL or UML models. However, all approaches are limited in their applicability to specific scenarios and resources of embedded systems. While there exist approaches that automatically map software system models to network simulators, these simulators only consider topologies, movements, and networks. None of the currently available network simulators provides realistic simulation of platform timing or memory resources, which have a significant impact on the performance of the software system (this impact is further outlined in Section 6 of this work). Additionally, none of the aforementioned simulators supports non-networking devices and user interactions.

On the other hand, simulators such as XEEMU [8] and Avrora [9] only provide platform simulation, with no support for the simulation of other resources and only very limited support of devices. Especially the simulation of realistic networks or of realistic environments consisting of sensors and actuators is not supported. This limits the applicability of these simulators to systems without communication requirements.

To overcome these shortcomings, our simulation framework combines available, specialized simulators into a tailored simulator instance that provides a holistic simulation of all required system artifacts. The simulator integration is controlled by a simulation script. By using AndroMDA [13], *C-PartsSim* generates this simulation script automatically when it is started out of a UML model describing the testing scenario.

3 The UML Platform Testing Profile

Based on the UML testing profile, we have devised a UML platform testing profile that supports the modeling of platform testing scenarios as well as performance evaluation scenarios. The UML testing profile [2] was developed to define a language for modeling the structure of test systems. The rationale was to define a profile that supports model-driven testing, supplementing model-driven development of software systems. The following core concepts were defined to achieve this goal:

- *Test contexts* are a collection of test cases and a test configuration.
- A *test case* is the complete specification of one test, describing on how the System under Test (SUT) should be tested for a given test objective. It specifies the interaction of test components with an SUT to realize a test objective. Test objectives specify the objective of a test case or of a test context.
- A *test configuration* defines a set of test components including the System under Test and the connections between them. *Test components* realize the behavior of a test case, they interact with other test components and with the System under Test. Test components are active parts of a test context. They execute sequences of stimuli and coordinate with other test components.

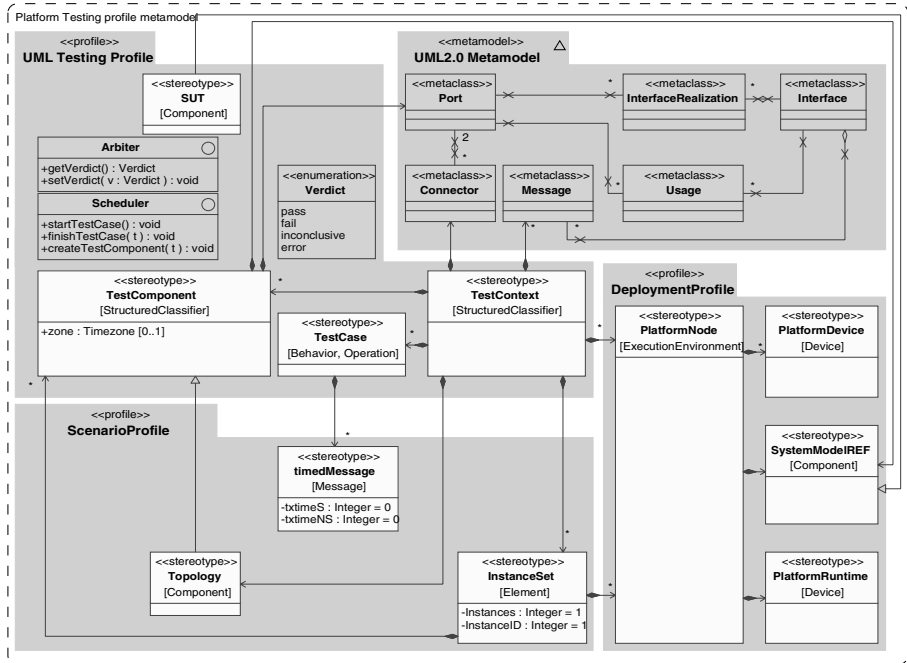


Fig. 1. Framework components

- The result of a test is judged by the *arbiter* component, and can be positive, negative or inconclusive. Arbiters support the automatic execution of testing scenarios and the assessment of the test results.

Our platform testing profile extends the UML testing profile with concepts that support the modeling of realistic deployments. Therefore, we introduce platform nodes, which model the physical entity of an execution environment, platform runtimes that model the computational resources of a platform, and devices. Every node is assigned to one execution platform, which controls the platform resources of that node. Every node executes one software system model on that platform. Devices model hardware components, or user interfaces. Devices and platforms are connected to test components, which implement their behavior.

As shown in Figure 1, the platform testing profile extends the UML testing profile, and can be subdivided into two parts. The deployment profile models realistic deployments, the scenario profile contains additional stereotypes that are used to model platform testing configurations. The additional stereotypes of the scenario profile specify node topologies, instance sets and timed messages. Timed messages are used to specify the behavior of a test case; messages are generated and transmitted to test components at defined points of time. Instance sets specify deployments of similar nodes, and therefore support the creation of large deployments. The example test context in Section 6 contains an example for an instance set. A platform testing configuration must contain exactly one topology and exactly one scheduler. It may

contain any number of nodes, which may also be of different type. The topology component controls the placement of every simulated node. This placement can be accessed by all test components, i.e. network simulators or simulated devices. The scheduler controls the simulation by recording and ordering all simulation events. By replacing the scheduler component, the time base of a simulation can be changed from simulation time to real-time, which supports the integration of a simulator into a running system.

Platform nodes represent simulated physical entities. They consist of a platform runtime, of zero to many devices, and of an associated behavior. The associated behavior is either implemented natively, or it is a software model, e.g. the System under Test. Platform runtimes represent characteristics of an execution environment. An associated test component implements the platform simulation and is responsible for the level of detail the platform runtime is simulated with. Currently, we have implemented two test components, simulating two different platforms: A generic platform component, modeling no constraints at all, as well as the Avrora component, which is based on the Avrora simulator [9], accurately modeling the performance and energy consumption of an Atmel ATmega128L processor. This way, the evaluation of platform-independent as well as the evaluation of platform-specific scenarios is possible. Devices are used to model existing or virtual devices. Devices are always connected to exactly one node; the device behavior is also implemented by a test component.

Depending on the simulated deployment, multiple levels of abstraction are supported. Figure 2 shows an example from the embedded systems domain on three abstraction levels. Figure 2a presents a basic scenario, which has been modeled with stereotypes defined by the UML testing profile only. Two instances of the System under Test (*ProtocolSUT*) are executed, and connected through the virtual channel “SimpleMedium”, which is implemented by a test component. The medium is an abstract SDL model, which simulates a perfect medium. Since no execution environment was assigned to the test components, the generic platform without resource constraints is used. Using this abstraction level, the validation of the functional behavior of the System is supported.

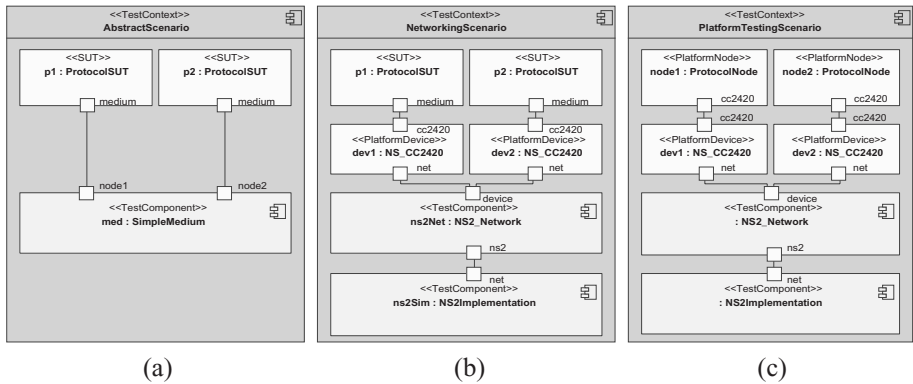


Fig. 2. Example scenario with multiple levels of abstraction

Figure 2b shows an abstraction level that is used after successful validation of the functional behavior. This abstraction level links the network simulator *ns-2* as simulation component to the System under Test. The System under Test communicates with the network simulator through simulated devices of type *NS_CC2420*, which simulate an interface to the ChipCon CC2420 wireless transceiver chip. The behavior of the transceiver chip and radio propagation is simulated by the *ns-2*; the test component *NS2_Network* configures a simulated wireless network, the component *NS2Implementation* provides the simulator backend. This abstraction level is suitable for platform testing after a communication technology was selected. The network simulator provides an accurate simulation of network parameters like bandwidth, delay, and jitter. This way, it can be evaluated whether a selected communication technology fulfills the requirements. Again, the System under Test is executed on the generic platform, which has no resource constraints.

The scenario shown in Figure 2c adds platform simulator components. Protocol-Node extends the generic MicaZ platform, which provides an Atmel ATmega128L microcontroller. The platform is simulated by the Avrora simulator, which is integrated as simulation component into *PartsSim*. This way, a cycle-accurate simulation of the ATmega128L microcontroller is provided.

4 C-PartsSim

To simulate UML platform testing models, we have devised *C-PartsSim*, a simulator that integrates specialized platform and native simulators dynamically. *C-PartsSim* includes a front-end that automatically transforms a model of a testing scenario into a simulation script. The simulation script is a TCL file that creates a tailored instance of *C-PartsSim* for a specific scenario. It instantiates necessary simulation components and sets up links between them.

A tailored *C-PartsSim* instance is created by selecting and connecting simulation components. *C-PartsSim* processes Nodes and test components. Platform nodes are mapped to nodes, all other component types, i.e. devices, and platform runtimes, are mapped to generic test components. This is done during the AndroMDA transformation (see Figure 3). Test components have, either a native implementation, or they have a behavior model assigned to them, which can be specified in SDL or UML. Nodes link test components together.

Figure 3 shows the transformation of an instance set. The platform node cyclist and all connected platform devices and test components are mapped to *C-PartsSim* test components. AndroMDA generates an array into the TCL script for the instance set, and first instantiates all test components. In a second step, connections are transformed. A third step (not shown) transforms the sequence diagram specified behavior of the associated test case into events that are generated at the appropriate times.

Services provided by test components are connected to each other using synchronous, message-based communication for interaction. The extensibility of the simulation framework is given due to the ability to define or to specialize new service interfaces, message types, and services anytime – even while a simulation scenario is executed. This way, a broad variety of system structures is supported by the simulator. New components can easily be integrated into the simulation framework (see Section 5).

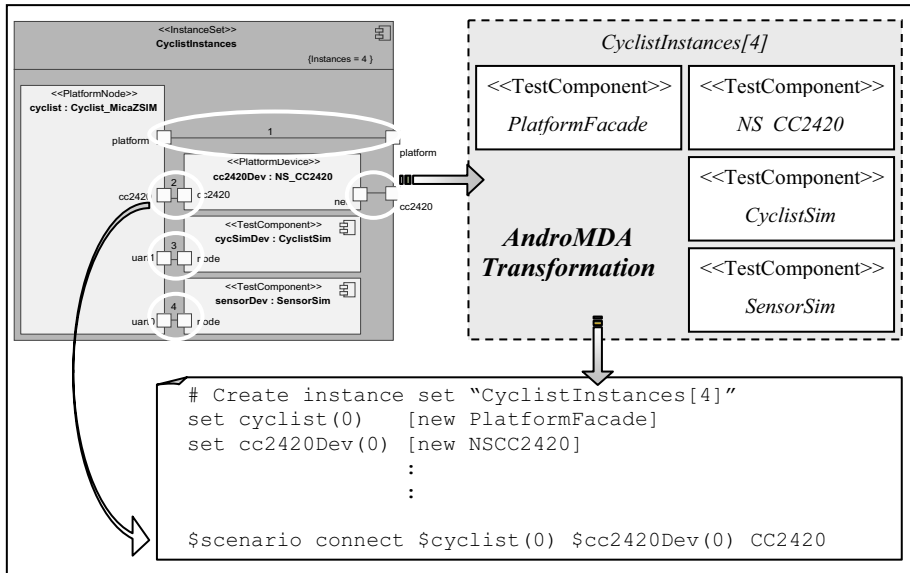


Fig. 3. Transformation of platform testing model into C-PartsSim configuration

The creation of a simulation framework requires clear definitions and semantics for all connected simulator components. One important definition for all simulators is the simulation time. Simulation time is the relevant time for all simulated devices and models. A simulated operation, i.e. the transmission of a frame over a wireless network ideally requires the same amount of simulation time that a real transmission would require. On the other hand, the simulation of this transmission may require any amount of real time, which depends on the resources of the host system, on which the simulation is executed on.

Our simulation framework splits the whole simulation into events, which trigger simulation operations in multiple simulation components. The scheduler orders all events according to their scheduled firing time in simulation time, and therefore serializes the execution of simulation operations. Simulation time is eventually advanced when the next event in the queue is processed. Currently, a token passing architecture is implemented. Only one simulation component can acquire the token and therefore can be active at any point of time in real time.

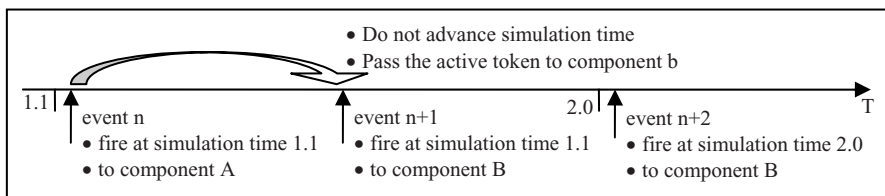


Fig. 4. Token passing

In Figure 4, two simulation components A and B schedule events at simulation time 1.1. The scheduler serializes the processing of these events in real-time. Event n of component A is processed first. Afterwards, event $n+1$ of component B is processed. Since event $n+1$ is scheduled at the same simulation time as event n , the simulation time is not advanced. After completion of event $n+1$, event $n+2$ is scheduled. The token remains at component B. The simulation time is advanced to time 2.0 before the event is dispatched. This way, the execution of all simulation components is serialized. Nevertheless, all simulation components can schedule and execute events at the same point of simulation time.

A simulation operation cannot be interrupted by another simulation event, and can have any duration in simulation time. The duration of a simulation operation defines the maximum offset between the simulation time of the scheduler, and the simulation time of a simulation component. The acceptable synchronization offset constrains the accuracy of the simulation. Operations that require larger amounts of simulation time should be separated into multiple, shorter operations to decrease synchronization offset.

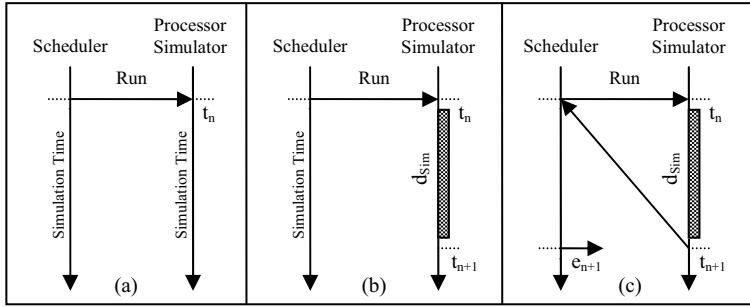


Fig. 5. Processor simulator

In Figure 5, the processor simulation component receives a “Run” message (Figure 5.a), which triggers a simulation operation that simulates the execution of n_{cycles} processor cycles. The simulation time of the processor simulation component and of the main scheduler is t_n . The duration of the simulation operation d_{sim} is defined to be $n_{cycles} * d_{SimCycle}$, the number of cycles that are executed multiplied with the simulated duration of one processing cycle. The simulation time of the processor simulation component is $t_{n+1} = t_n + d_{sim}$, after the atomic event is completed (Figure 5.b). Therefore, it schedules the next “Run” event e_{n+1} to catch up with the other simulation components. The scheduling request is received by the scheduler at simulation time t_n , because its simulation time was not advanced yet. The event e_{n+1} is scheduled at simulation time t_{n+1} (see Figure 5.c). At time t_{n+1} , the simulation scheduler and the processor simulation component are synchronized again. In the example above, the synchronization accuracy is constrained by the $n_{cycles} * d_{SimCycle}$, which is the duration of the simulation operation.

The problem of synchronizing multiple simulation components becomes evident in scenarios that are simulated by a combination of multiple simulators. Simulation components interact by message-based communication, i.e. by scheduling events of

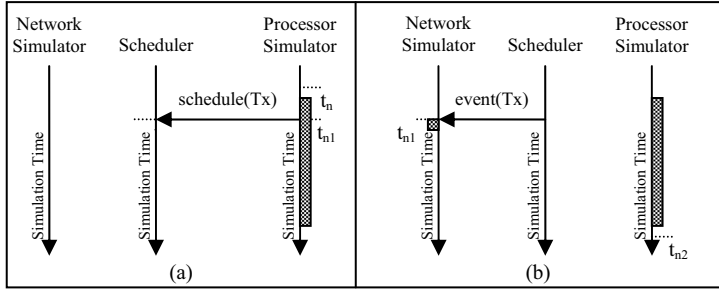


Fig. 6. Synchronization of multiple simulation components

each other. The basic interfaces that are used for communication are defined by the simulation framework. All events that are generated by the active simulation component for another component are queued in the scheduler.

In the example shown in Figure 6, a Tx request is transmitted from the processor simulation component to the ns-2 simulation component, which implements the network simulator. The event is transmitted at simulation time $t_{n1} = t_n + d_{\text{SimCycle}} * 100$, i.e. one hundred simulated processor cycles in simulation time after the run request was received. This message triggers the transmission of a Wireless LAN frame over a simulated network. Instead of transmitting the message directly to the network simulator, the message is scheduled for delivery to the network simulator component at simulation time t_{n1} (see Figure 6.a). The simulation time of the network simulator is still t_n , which is earlier than t_{n1} , so it will process the event precisely at time t_{n1} . After the processor simulator completes its simulation operation, the scheduler forwards the event to the network simulator, and the Tx request for the simulated network is processed at simulation time t_{n1} (Figure 6.b). The simulation time of the processor simulator is already at $t_{n2} = t_n + (n_{\text{cycles}} * d_{\text{SimCycle}})$, $n_{\text{cycles}} > 100$. Since it is not possible to go backwards in simulation time, a response from the network simulator will be received by the processor simulation component, and therefore by the System under Test that is executed on it, at time t_{n2} at earliest.

Therefore, when multiple simulator components are coupled, all simulator components are responsible for retaining synchronization with the main simulation time. A longer, uninterrupted simulation yields shorter execution times, resulting from less overhead. Shorter simulation operations yield more accurate simulations, but consume more time due to increased synchronization overhead. The accuracy of simulations depends on the duration of the longest simulation operation that any of the simulator components utilize.

5 Extending *C-PartsSim*

Our simulation framework is modular; it can be extended with new test components as well as with new interfaces, and event types. Test components are the central point for extending *C-PartsSim*. They serve as simulation components, implementing

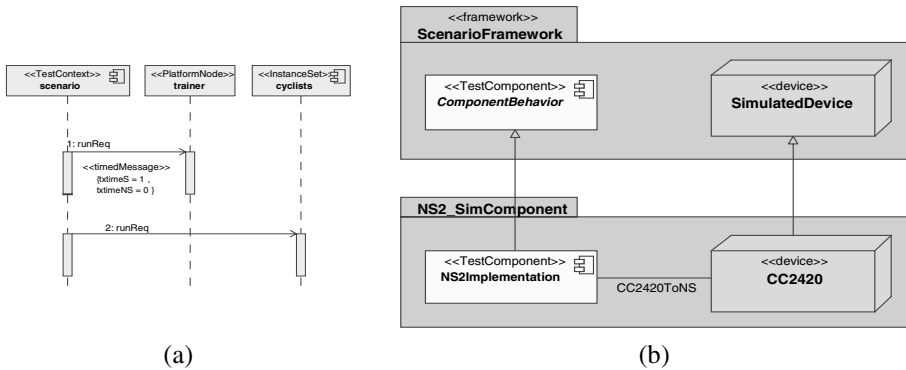


Fig. 7. Implementation of *C-PartsSim* components

backends for the simulation of devices and platforms. They also simulate scenario-specific parts, like user behavior or inputs to the System under Test. Test components are created either out of UML sequence diagrams, out of a model, or using native C++ or Java code.

Test components created out of sequence diagrams support a very simple behavior. Such a behavior only consists of timed messages that generate signals, which are transmitted at defined points of time (see Figure 7.a). This limitation originates from the transformation that is currently used by the front-end to *C-PartsSim* for handling sequence diagrams. The transformation embeds these sequence diagrams directly into the simulation script. Test components based on sequence diagrams can be used to trigger the System under Test using well defined sequences of messages, running predefined test cases.

The creation of test components out of models requires a compatible model runtime environment for the used transformation. We currently support the Cmicro and Cadvanced SDL-to-C compilers from the Telelogic SDL suite [14], as well as experimental support of the UML Tool TAU Generation 2 [15] and the SDL compiler ConTraST [20] through our SENF2 [12] runtime environment. This enables us to create test components, as well as Systems under Test from SDL and UML specifications. Scenarios consisting of both types of test components, whose behavior is specified in SDL and UML, are supported, too.

The creation of a simulation component using native C++ or Java code is used for the integration of existing simulators into *C-PartsSim*. Natively implemented test components must refine the basic *C-PartsSim* framework. The framework defines a message-based interface, which covers the basic *C-PartsSim* messages. Turning an existing simulator into a simulation component for *C-PartsSim* is done as follows:

- The generic signals of *C-PartsSim* must be specialized to resemble the specific needs of the added simulator. Eventually, multiple specializations are to be created to define a hierarchy of simulator-specific signals.
- Depending on the type of the added simulator, which is either a device or a platform simulator, the interface service to *C-PartsSim* must be implemented. This is supported by an existing framework, which can be extended.

- All internal simulation components of the added simulator, i.e. the scheduler component and the component that loads the scenario, must be replaced by a component that interfaces with the *C-PartsSim* scheduler service.
- All parts of the added simulator that are simulated with other simulator components from *C-PartsSim* must be replaced with interfaces to *C-PartsSim*. In ns-2, these were the topology simulator and the implementation of the System under Test. For the Avrora simulator, the network simulation was replaced.
- Simulator specific test components, eventually also node devices and runtime environments must be created. Figure 7.b shows an example integration of the CC2420 wireless transceiver chip, which is simulated by the ns-2 as simulator backend.

Depending on the complexity of the added simulator, this task can be completed within several days if the internal structure and the semantics of the added simulator are known. Using this methodology, it is possible to add new simulation backends for the simulation of existing devices as well as completely new devices that interact with each other.

6 Experiences

In this section, we present two case studies showing the feasibility of our simulator on multiple abstraction levels. The context of both evaluation scenarios is our Assisted Bicycle Trainer [16], a distributed system supporting the group training of cyclists. The distributed software system that is deployed to the cyclist and to the trainer nodes was specified with SDL. Sensor data is collected by every bicycle and transmitted over a wireless ad-hoc network over multiple hops to the trainer node. The processor is an Atmel ATmega 128L microcontroller, with very scarce computational and memory resources.

First, we show a test situation, where the model runtime environment is evaluated. The ability of evaluating the model runtime environment through simulations shows the strength of our approach. This scenario shows the capabilities of *C-PartsSim* to integrate scenario specific test components. The second case study is a performance prediction scenario, which we originally performed using *PartsSim*, the predecessor of *C-PartsSim* [4].

6.1 Evaluating a Model-Runtime Environment

The sensors of every bicycle periodically transmit data to a local node, which accumulates this data and forwards the accumulated data through the network once per second. Since sensor events may also happen spontaneously, the number of generated events per time unit is not predictable. If the number of events waiting in the signal queue exceeds the queue size, the behavior of the generated software system is no longer conforming to the SDL standard, which assumes unbounded queues. Therefore, we have devised an SDL language extension for the specification of queue bounds [18]. This language extension was tested with *C-PartsSim*, using a basic scenario together with a generic platform simulation.

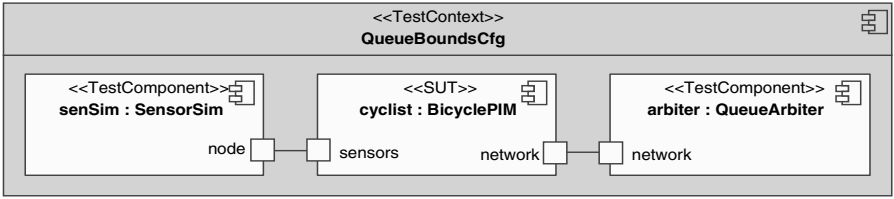
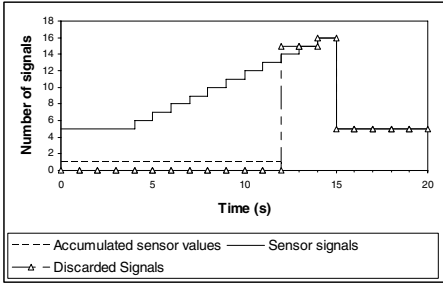


Fig. 8. Test context for testing an SDL language extension

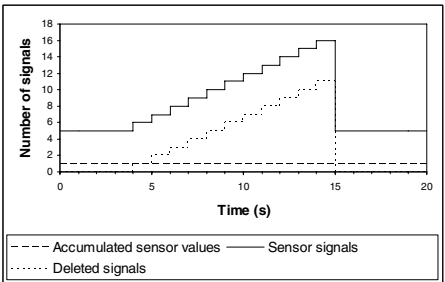
The simulation was performed using the test context shown in Figure 8. Internal simulator components, i.e. the scheduler and the topology were not modeled; therefore, default values are inserted by the transformation front-end. The test component SensorSim serves as a driver for the System under Test named BicycleSUT. The component SensorArbiter receives the accumulated sensor values. The behavior of all test components (SensorSim, BicycleSUT, ArbiterINST) is specified in SDL and loaded into the simulation at runtime. The SDL models are represented by artifacts. No platforms and devices are instantiated; therefore, test components are connected directly using message-based communication, and the simulation is performed on a generic platform, i.e. on platform without hardware constraints.

The SensorSim test component increases the number of generated signals by one per second. For this scenario, we have limited the SDL signal queue to 20 signals, such that the effect of producing results that are non-conforming to the specification becomes visible. Figure 9.a shows the results of the simulation using a runtime environment without our modifications.

After 12 seconds of simulation time, the number of sensor messages together with internal messages exceeds the queue size. The accumulated sensor values are not produced anymore, as the timer signal triggering the accumulation periodically can not be appended to the queue and is therefore discarded. In addition, all further incoming sensor values are discarded. The results of this test context, using our modified runtime environment as well as our language extensions, are shown in Figure 9.b. As indicated, the test context with queue bounds shows a correct behavior, the accumulated sensor values are transmitted once every second. This shows that our



(a)



(b)

Fig. 9. Test results with and without queue bounds

simulation framework is not only capable of evaluating the System under Test, but also the used runtime environments.

6.2 Testing a Communication System

The assisted bicycle trainer is placed on top of a communication system that is able to utilize contention-free communication. Therefore, a time-slotted, virtual medium is established, and slots are assigned to individual nodes. In [4], we have presented simulation results of this scenario. Here, we elaborate on modeling the scenario for *C-PartsSim*, and on the differences to *PartsSim*.

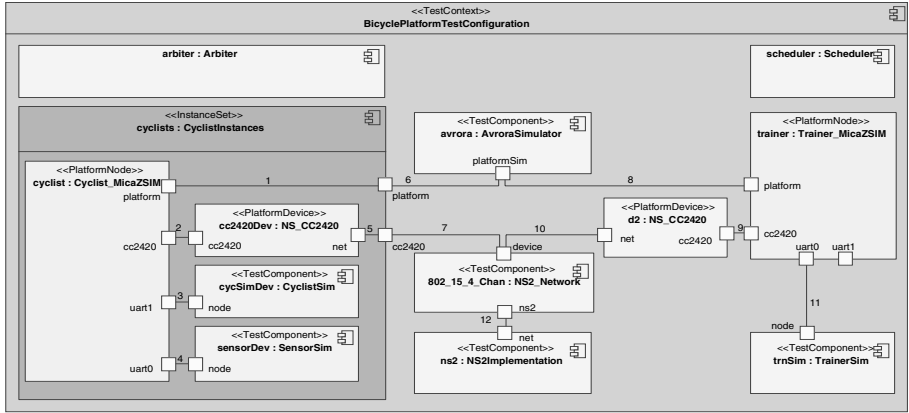


Fig. 10. Assisted Bicycle Trainer test context

Figure 10 shows the complete specification of the test context of Figure 3c, which is specified with our UML profile (see Figure 1). In contrast to the simulator *PartsSim* [4], all connections between devices and test components can be altered for a specific scenario. This provides much more flexibility than the *PartsSim* solution. *C-PartsSim* retains the high simulation accuracy that is provided by *PartsSim*. Scenarios simulated with *PartsSim* yield the same simulation results if the same random seeds are provided. Therefore, existing simulation scenarios can iteratively be ported from *PartsSim* to *C-PartsSim*.

7 Conclusions and Future Work

In this work, we have presented our approach to support model-driven platform testing and performance evaluation. This is supported by our simulation framework, which consists of a UML platform testing profile and a new, configurable simulator called *C-PartsSim*. Our framework provides defined semantics for interconnecting specialized simulators, which are turned into simulator components. A front-end to *C-PartsSim* instantiates a tailored platform simulator based on a given UML platform testing model. This way, *C-PartsSim* is capable of simulating the deployment of a

software system on multiple levels of platform abstraction. We have also documented a methodology for extending *C-PartsSim*, either with pre-existing, specialized simulator components or with test components, modeled with SDL or UML 2.0.

Our simulation results show the accuracy of the instances of our simulation framework. Scenarios can be tested with the help of a tailored system simulator at multiple levels of platform abstraction. The tailored system simulator is instantiated based on a UML model of the testing scenario. Future work in this area includes the integration of continuous simulators with event-based simulators

Acknowledgements

The work presented in this paper was carried out in the μ Pros project (funded by DFG under the project number Go503/5-1), and the BelAmI project (funded by BMBF, Fraunhofer-Gesellschaft, and MWFFK of Rheinland-Pfalz).

References

- [1] Janzen, D., Saiedian, H.: Test-driven Development: Concepts, Taxonomy and Future Directions. *IEEE Computer* 38(9), 43–50 (2005)
- [2] Object Management Group (OMG). UML 2.0 Testing Profile Specification, Version 1.0 (2004), <http://www.omg.org>
- [3] Shull, F., Basili, V., Boehm, B., Brown, W., Costa, P., Lindwall, M., Port, D., Rus, I., Tesoriero, R., Zelkowitz, M., M.: What We Have Learned About Fighting Defects. In: *Proceedings of the Eight IEEE Symposium on Software Metrics*, pp. 249–258. IEEE, Los Alamitos (2002)
- [4] Kuhn, T., Becker, P.: A Simulator Interconnection Framework for the Accurate Performance Simulation of SDL Models. In: Gotzhein, R., Reed, R. (eds.) *System Analysis and Modeling: Language Profiles*. LNCS, vol. 4320, pp. 216–228. Springer, Heidelberg (2006)
- [5] Kuhn, T., Gerald, A., Gotzhein, R., Rothländer, F.: ns+SDL – The Network Simulator for SDL Systems. In: Prinz, A., Reed, R., Reed, J. (eds.) *SDL 2005*. LNCS, vol. 3530, pp. 103–116. Springer, Heidelberg (2005)
- [6] Information Sciences Institute, University of Southern California: The Network Simulator ns-2 (valid in 2007), <http://www.isi.edu/nsnam/ns/>
- [7] Chang, X.: Network Simulations with OPNET. In: Farrington, P.A., Nembhard, H.B., Sturrock, D.T., Evans, G.W. (eds.) *Proc. of WSC 1999*, Piscataway, New Jersey (U.S.A.), vol. 1, pp. 307–314. IEEE, Los Alamitos (1999)
- [8] Herczeg, Z., Kiss, Á., Schmidt, D., Wehn, N., Gyimóthy, T.: XEEMU: An Improved XScale Power Simulator. In: Azémard, N., Svensson, L. (eds.) *PATMOS 2007*. LNCS, vol. 4644, Springer, Heidelberg (2007)
- [9] Titzer, B.L., Lee, D.K., Palsberg, J.: Avrora: Scalable Sensor Network Simulation with Precise Timing. In: *Proceedings of the Fourth International Symposium on Information Processing in Sensor Networks*, IPSN 2005, UCLA, Los Angeles, California, USA, April 25–27 (2005)
- [10] Hatnik, U., Altmann, S.: Using ModelSim, Matlab/Simulink and NS for Simulation of Distributed Systems. In: *IEEE PARELEC 2004*, Dresden, September 7–10, 2004, pp. 114–119 (2004) ISBN 0-7695-2080-4

- [11] Almesberger, W.: Umlsim - A UML-based simulator. In: Proceedings of the 10th International Linux System Technology Conference (Linux-Kongress 2003), pp. 202–213 (October 2003)
- [12] Kuhn, T., Gotzhein, R., Webel, C.: Model-Driven Development with SDL – Process, Tools and Experiences. In: Kühne, T. (ed.) MoDELS 2006. LNCS, vol. 4364, Springer, Heidelberg (2007)
- [13] AndromDA (valid in, 2007), <http://www.andromda.org>
- [14] Telelogic AB: Telelogic Tau Generation 1, <http://www.telelogic.com/products/tau/index.cfm>
- [15] Telelogic AB, Telelogic TAU Generation 2, <http://www.telelogic.com/products/tau/g2>
- [16] Fliege, I., Gerald, A., Gotzhein, R., Jaitner, T., Kuhn, T., Webel, C.: An Ambient Intelligence System to Assist Team Training and Competition in Cycling. In: Moritz, E.F., Haake, S. (eds.) The Engineering of Sport 6. Developments for Sports, vol. I, pp. 103–108. Springer Science, Business Media, New York (2006)
- [17] International Telecommunications Union: Specification and Description Language (SDL). ITU-T Recommendation Z.100 (August 2002)
- [18] Gotzhein, R., Grammes, R., Kuhn, T.: Specifying Input Port Bounds in SDL. 13th International SDL Forum. In: Gaudin, E., Najm, E., Reed, R. (eds.) SDL 2007. LNCS, vol. 4745, Springer, Heidelberg (2007)
- [19] De Miguel, M., Lambolais, T., Hannouz, M., Betgé-Brezetz, S., Piekarec, S.: UML Extensions for the Specification and Evaluation of Latency Constraints in Architectural Models. In: ACM Proc. of WOSP 2000, Ottawa, Canada, pp. 83–88 (2000)
- [20] Fliege, I., Grammes, R., Weber, C.: ConTraST – A Configurable SDL Transpiler And Runtime Environment. In: Gotzhein, R., Reed, R. (eds.) SAM 2006. LNCS, vol. 4320, Springer, Heidelberg (2006)
- [21] Stepler, M.: SPEETCL, SDL Performance Evaluation Tool Class Library, AixCom GmbH (valid in, January 2008), <http://www.aixcom.com/>
- [22] Stepler, M.: SDL2SPEETCL, SDL Performance Evaluation Tool Class Library, AixCom GmbH (valid in, January 2008) <http://www.aixcom.com/>

Testing Metamodels

Daniel A. Sadilek and Stephan Weißleder

Humboldt-Universität zu Berlin
Unter den Linden 6
10099 Berlin, Germany
{sadilek, weissled}@informatik.hu-berlin.de

Abstract. In this paper, we deal with errors in metamodels. Metamodels define the abstract syntax of modeling languages. They play a central role in the Model-Driven Architecture. Other artifacts like models or tools are based on them and have to be changed if the metamodel is changed. Consequently, correcting errors in a metamodel can be quite expensive as dependent artifacts have to be adapted to the corrected metamodel. We argue that metamodels should be tested systematically with automated tests. We present a corresponding approach that allows automated metamodel testing based on a test specification. From a test specification, multiple test models can be derived. Each test model defines a potential instance of the metamodel under test. A positive test model defines a potential instance that should be an actual instance of the metamodel; a negative test model defines one that should not. We exemplify our approach with a metamodel for defining a company's structure. Finally, we present *MMUnit*, an implementation of our approach that builds on the Eclipse platform and integrates the JUnit framework. MMUnit allows to test EMF-based metamodels, which can contain additional constraints, e.g. constraints expressed in OCL.

1 Introduction

Metamodels describe the structure of modeling languages. They play a central role in all model-driven MDx technologies like MDA, MDD, and MDSE [1]. With these technologies becoming ever increasing popular, metamodels are getting more and more important.

Metamodels are artifacts of the software engineering process just like ordinary models, program code, or documentation. As such, they *contain errors*: For instance, they may specify classes with attributes of wrong type; associations between classes may be missing or superfluous or have wrong multiplicities; additional constraints (e.g. expressed in OCL) may contain errors.

Model transformations and tools for a modeling language like editors, interpreters, or debuggers base on a metamodel. When errors in the metamodel are detected and removed after modeling or after tool development, already created models and dependent tools must be adapted.¹ This requires additional effort

¹ This applies both to manually implemented tools and to declarative tool descriptions that tools can be automatically generated from.

and causes higher costs. Hence, the early detection of errors in a metamodel can save time and money.

In software engineering, *testing* is the primary means to detect errors. To our knowledge, metamodels are not tested systematically, yet. In this paper, we advocate *testing metamodels* and present an approach for automated testing based on a test specification for multiple metamodel test cases.

In the following section, we define the terms used in this paper. In Sec. 3, we develop an approach for automated, systematic metamodel testing. We clarify this approach by examples in Sec. 4 and by a detailed explanation of the corresponding test execution process in Sec. 5. We sketch our prototype implementation in Sec. 6, discuss related work in Sec. 7, and conclude in Sec. 8.

2 Terminology

In this section, we give brief definitions of the terms used in this paper and we introduce necessary mathematical notation.

Language, Syntax, Semantics. A *language* consists of syntax and semantics. The *syntax* of a language is a possibly infinite set of language utterances. A grammar is a common means of defining the syntax of a textual language. For graphical languages, graph-grammars or metamodels can be used. Often, *abstract syntax* and *concrete syntax* are distinguished: The abstract syntax of a language defines the mere structure of the language utterances; the concrete syntax of a language additionally contains elements that are necessary for displaying language utterances to a user, e.g. brackets in a mathematical expression. The *semantics* of a language provides a meaning for the language utterances.

Metamodel, Model. A *metamodel* in our sense is an object-oriented specification of the abstract syntax of a modeling language. It does not specify other aspects of a modeling language like concrete syntax or semantics. A *metamodel instance* or a *model* is a language utterance of a language whose abstract syntax is defined with a metamodel. Given a metamodel mm , we denote the abstract syntax specified with this metamodel with Θ_{mm} . This means, Θ_{mm} is a set of models that contains all instances of the metamodel mm .

We consider metamodels that are instances of a MOF-like meta-metamodel [2]. Those metamodels are elements of the set Θ_{MOF}^2 .

3 Specifying Metamodel Tests

In this section, we describe the artifacts we use for metamodel testing. While this section provides a rather static view on our approach, Sec. 5 describes how these artifacts are used for metamodel test execution.

² Due to the recursive definition of MOF, MOF itself is an element of the set of models it specifies: $MOF \in \Theta_{MOF}$.

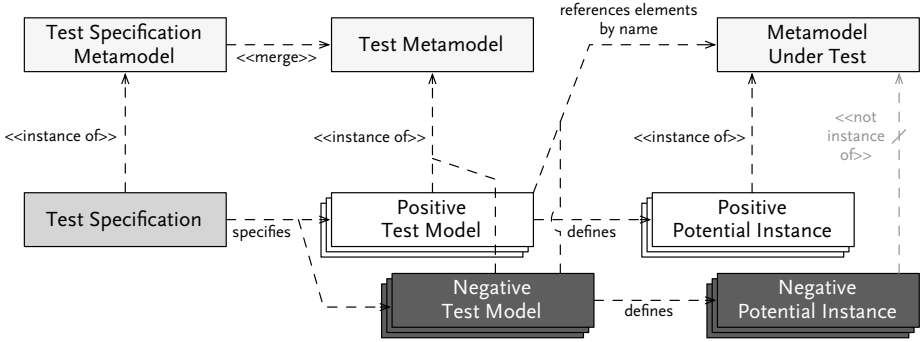


Fig. 1. Relations between different artifacts for metamodel testing

Figure 1 shows the relations between the different artifacts of our approach. A user creates a *test specification*, which is an instance of the *test specification metamodel* (TSM). A test specification specifies one or multiple *test models*, which are instances of the *test metamodel* (TMM). Each test model defines a *potential instance* of the *metamodel under test* (MMUT). For this, it references elements of the MMUT by name. Furthermore, a test model defines whether the potential instance should or should not be an actual instance of the MMUT. In the first case, we speak of a *positive* test model with a *positive* potential instance; in the second case, we speak of a *negative* test model with a *negative* potential instance. Detailed definitions of the artifacts follow in the rest of this section.

3.1 Testing Metamodels with Potential Instances

What is an error in a metamodel? In general, an error in a metamodel MMUT manifests in the specified set of models Θ_{MMUT} . If the MMUT is erroneous, Θ_{MMUT} contains a model that is undesired or it lacks a model that is desired. For instance, an MMUT can define classes with attributes of wrong type; associations between classes can have wrong multiplicities or can be missing or superfluous.

One possibility to test a set specification is to check if each element of the set is desired and each desired element is part of the set. Since metamodels generally specify an infinite set of models, this is impossible. Instead, representative elements can be given that lie either inside or outside this set. For metamodels, we call these representative elements *potential instances* of the MMUT.

Let M be the set of all models specifiable with MOF-compliant metamodels:

$$M = \bigcup_{x \in \Theta_{MOF}} \Theta_x.$$

Definition 1 (Potential instance). A potential instance pi is an arbitrary model from the set of all possible models: $pi \in M$. If a potential instance is an instance of the MMUT, which means $pi \in \Theta_{MMUT}$, we call it a positive potential instance (Fig. 2a); otherwise, we call it a negative potential instance (Fig. 2b).

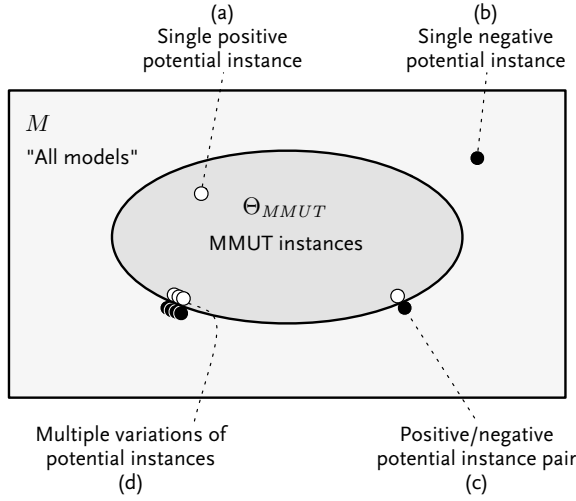


Fig. 2. Metamodel as a set specification; models as elements

Figure 2 shows an Euler diagram visualizing this idea. The rectangle represents M and the oval represents $\Theta_{MMUT} \subset M$.

3.2 Defining Potential Instances with Test Models

In our approach, the user defines potential instances and annotates if they are positive or negative ones. Technically, he needs a way to describe and to store potential instances. As potential instances are models, it seems as if we would just need a corresponding metamodel. Can we use the MMUT? No, we cannot for two reasons: First and foremost, a potential instance may just not be an instance of the MMUT. This may either be because it is a negative potential instance or because the MMUT contains an error so that the potential instance, albeit a positive one, is not an instance of the MMUT. Second, the user may want to specify instances *before* the MMUT even exists. In fact, not only the MMUT is not a fitting metamodel for potential instances but any fixed metamodel is not fitting.

Theorem 1. *No fixed metamodel fmm can serve the purpose to store potential instances.*

Proof (No fixed metamodel for potential instances (indirect)). Suppose a fixed metamodel fmm exists that can store any potential instance $pi \in \Theta_{fmm}$. As a consequence, all classes instantiated in pi must be defined in fmm . As each (meta)model is finite, fmm contains a finite number of classes. Therefore, there is a class name cn that is not already used for a class in fmm . We construct a potential instance \hat{pi} that contains an instance of the class with the name cn . Since Θ_{fmm} does not contain a class called cn , $\hat{pi} \notin \Theta_{fmm}$. \square

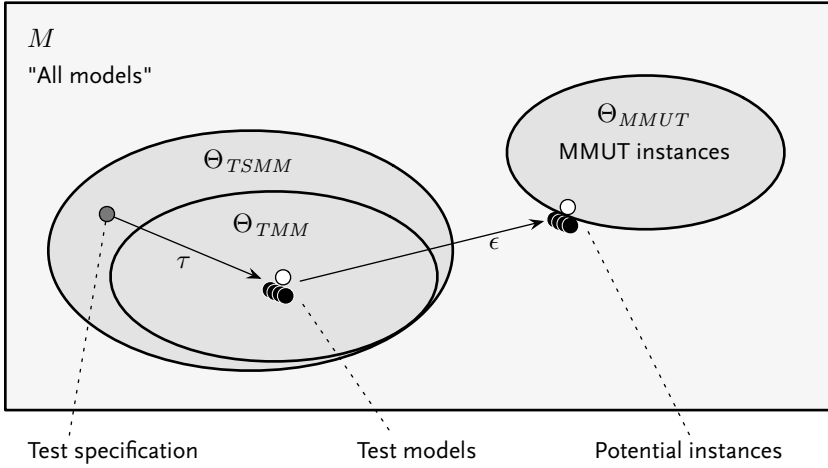


Fig. 3. A test specification specifies multiple test models. Each test model defines a potential instance of the MMUT. White = positive, black = negative test model/potential instance.

Therefore, potential instances used to test an MMUT cannot be stored as models. Nevertheless, they can be *defined* by models, which we call *test models*.

Definition 2 (Test model). A test model is a model that defines a potential instance. It contains the information whether the defined potential instance is a positive or a negative one. If a test model defines a positive potential instance, we call it a positive test model; otherwise, we call it a negative test model.

Definition 3 (Test metamodel). The test metamodel (TMM) is the meta-model for test models.

Formally, we define a function $\epsilon : \Theta_{TMM} \rightarrow M$ that maps a test model to a potential instance. ϵ is realized in the test execution (Sec. 5). Figure 3 shows an Euler diagram visualizing Θ_{TMM} , Θ_{MMUT} , and ϵ . Also shown is Θ_{TSM} , which we will describe later on.

3.3 Structure of the Test Metamodel

The TMM (Fig. 4) is similar to the metamodel for UML object diagrams with two differences: (1) it references elements (classes, attributes, associations) from the MMUT by name and (2) it contains an additional attribute `TestModel.positive` to express whether the defined potential instance is a positive or a negative one.

A `TestModel` consists of `Instances` that can have `Attributes` and are connected by `References`. An `Instance` defines an instance of a class from the MMUT with the name from the `Instances`'s attribute `className`. For documentation, an `Instance` can be given an optional name in the attribute `objectName`. Each `Attribute` has a name and a value, which is given generically as a string

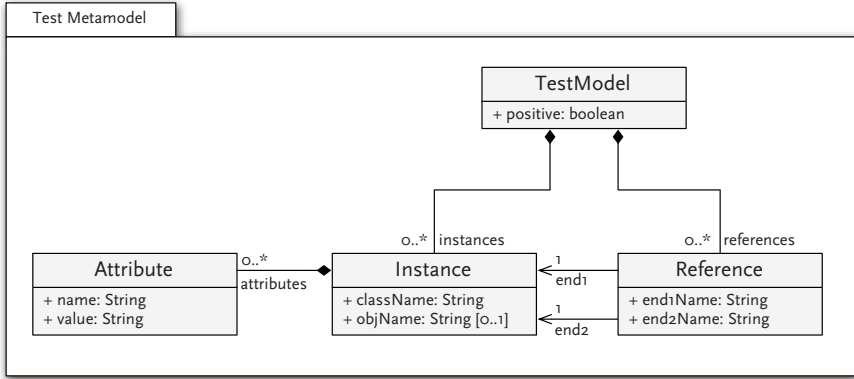


Fig. 4. The test metamodel

literal. The name of an **Attribute** together with the name of its containing **Instance** identify the attribute of the potential instance of the MMUT which the value should be assigned to. For the same purpose, the ends of a **Reference** can be named.

Parallels in Textual Languages. We found some interesting parallels between our TMM and textual languages. One of the reasons why we couldn't store potential instances as instances of the MMUT was that a user may want to define potential instance before the MMUT exists. In test-driven development [3] this approach is called *test first*. In traditional programming, “test first” means to write a test before writing the piece of software that is tested.

In a textual language, writing test code that refers to not yet existing code is possible due to the genericness of the textual representation of the test. This genericness relies on two properties of text: (i) Text can be written and saved without a specific grammar. For instance, Java code can be written in an ordinary text editor that knows nothing about Java. Even with an editor made particularly for Java editing, Java files are saved as plain text. (ii) Text that refers to elements like classes, variables, or functions can be written even when these elements are not (yet) defined³. This is possible because references in source code are expressed by writing names whose *resolution is deferred*.

The TMM possesses the same properties: (i) It is a “universal” metamodel in the sense that it allows to define arbitrary potential instances. (ii) It allows to reference elements of the MMUT by name, so that reference resolution is deferred (cf. Sec. 5).

3.4 Specifying Multiple Test Models with a Test Specification

With the test models presented so far, a user can define potential instances and can specify whether they lie inside or outside Θ_{MMUT} . But which test models

³ To be more precise: when no text has been written that defines these elements.

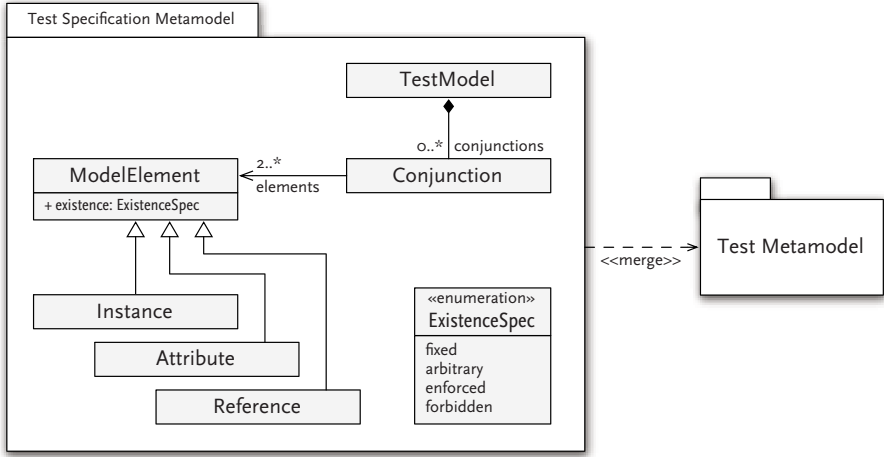


Fig. 5. The test specification metamodel

make good tests? Test models can define potential instances that just lie anywhere inside or outside of Θ_{MMUT} , like those shown in Figs. 2a and 2b. But these probably don't make good tests because they lie far from the boundary of Θ_{MMUT} . Therefore, they are not sensitive to changes in the metamodel.

What does it mean for a potential instance to be “near the boundary” of Θ_{MMUT} ? The nearer a potential instance is to the boundary of Θ_{MMUT} , the less has to be changed in the defining test model in order to cross the boundary. With pairs of a positive and a negative test model that differ only slightly, one can demarcate the boundary of Θ_{MMUT} (Fig. 2c). Such pairs should be as similar as possible in order to keep the demarcated boundary as thin as possible. For instance, they may differ in an attribute value or by the existence of some object or reference. This resembles the common testing technique of boundary testing [4,5].

Also, a user may want to specify multiple variations of a test model (Fig. 2d). We want to enable a user to specify such pairs and multiple variations with little redundancy. Little redundancy means that he has to specify the similar parts of the test models only once and has to specify only the differences, in addition. For this, we define a new type of models: *test specifications*.

Definition 4 (Test specification). A test specification is an extended version of a test model. It contains additional existence specifications for the model elements. The existence specifications specify how multiple (ordinary) test models can be derived from one test specification.

Definition 5 (Test specification metamodel). The test specification metamodel TSM is the metamodel for test specifications. It extends the TMM with the possibility to give existence specifications for model elements.

Formally, we define a function $\tau : \Theta_{TSM} \rightarrow \mathcal{P}(\Theta_{TMM})$ that maps a test specification to a set of test models. τ is realized in the test execution (Sec. 5). The Euler diagram in Fig. 3 shows Θ_{TSM} and a test specification that specifies one positive and four negative test models.

The TSM extends the TMM by package merge (Fig. 5). It introduces the common superclass `ModelElement` for `Instance`, `Attribute`, and `Reference`. A `ModelElement`'s existence specification can be set in its attribute `existence`. Possible existence specifications are *arbitrary*, *enforced*, and *forbidden*. The default is arbitrary.

Assuming a test specification with `TestModel.positive = true`, from a user's perspective the existence specifications mean the following: *Arbitrary*: the element has no special role. *Enforced*: keeping the element results in a positive test model and removing it in a negative one. *Forbidden*: keeping the element results in a negative test model and removing it in a positive one. In Sec. 5, this rather intuitive definition is defined more formally.

Multiple elements can have an existence specification different from *arbitrary*. In this case, the test specification specifies not only a pair of test models but multiple test models. For instance, in a test specification with `TestModel.positive = true`, four elements could be forbidden. Then, this test specification would specify one positive and four negative test models (Fig. 3). The positive test model would contain no forbidden element and each of the four negative test models would contain one of the four forbidden elements.

If the four elements should be `forbidden` conjunctionally, i.e. they should describe just one negative test model, then they can be connected by a `Conjunction`.

4 Example: An Erroneous Metamodel

In this section, we give the reader an intuition of how metamodel test specification and execution may look like from the perspective of a metamodel engineer. The test specifications in this section are shown in screenshots of the test specification editor that is part of our implementation (Sec. 6). The task of the metamodel engineer is to develop tools that support several functions connected to the hierarchy of a company, e.g. book keeping or responsibility assignment. As she is a determined modeler, she starts with developing a metamodel. In order to prevent high adaptation costs, she tests the metamodel with scenarios of known company structures.

Figure 6 shows an exemplary metamodel. Its purpose is the definition of a company's structure. The metamodel contains classes for the projects of a company, its employees and their insurances, and for the company itself. In our example, each company has to contain at least one employee and each employee can have at most one boss. The OCL invariant of `Employee` is satisfied iff the insured sum for each employee is at least 1.000.000 or the employee has no boss. However, the engineer wanted to express that the insured sum for each boss has to be at least 1.000.000. Since these two expressions are not equal, the specification of the metamodel contains an error.

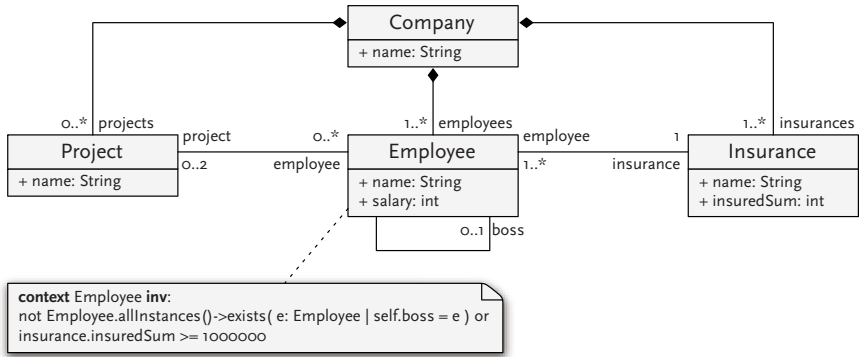


Fig. 6. An erroneous metamodel for a company’s structure

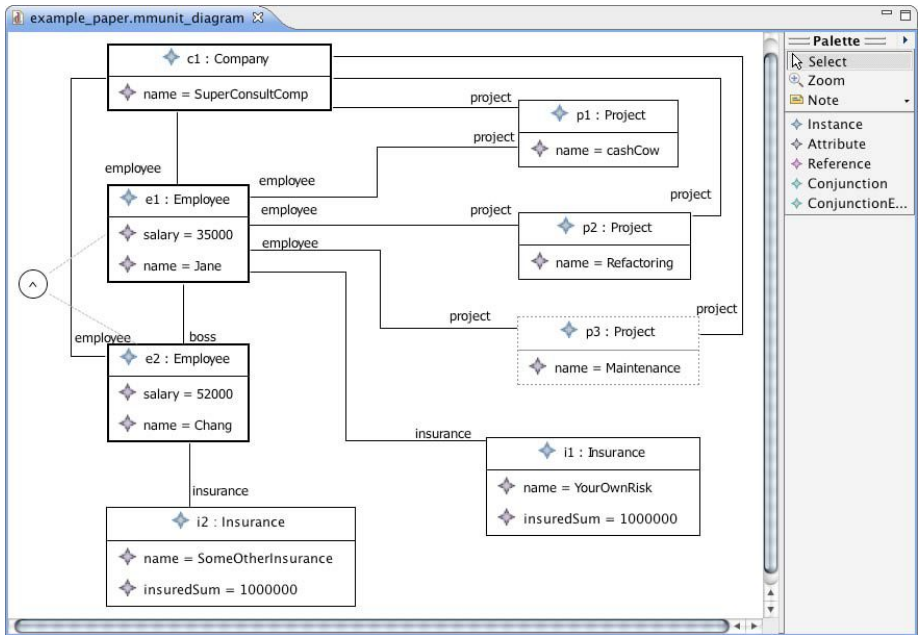


Fig. 7. Screenshot of a test specification that successfully validates the MMUT

In the following, we present the two test specifications available to the meta-model engineer, which are depicted in Fig. 7 and Fig. 8.

We start with Fig. 7: The object **c1:Company** is enforced, which is depicted by the thick, black border. Removing it results in a negative test model, for which the defined potential instance must not be an instance of the company metamodel. This test is passed for the company metamodel because it specifies that each employee must belong to one company.

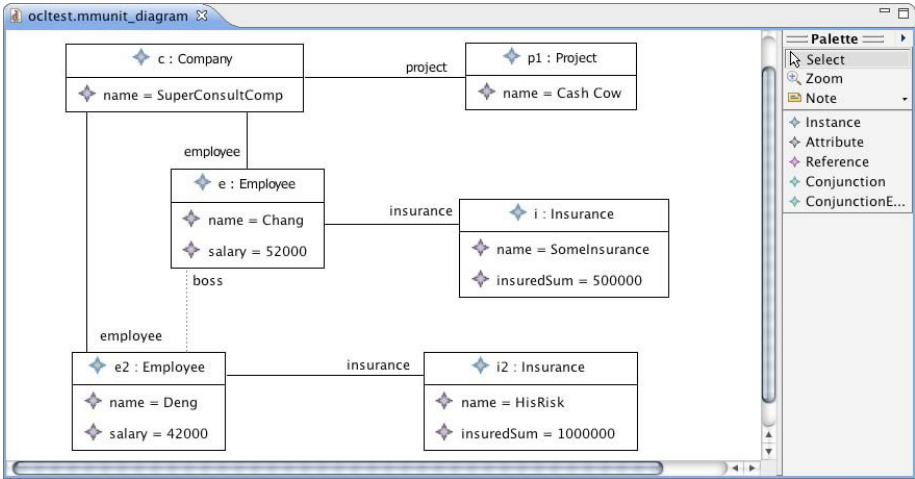


Fig. 8. Screenshot of a test specification that reveals the error in the MMUT

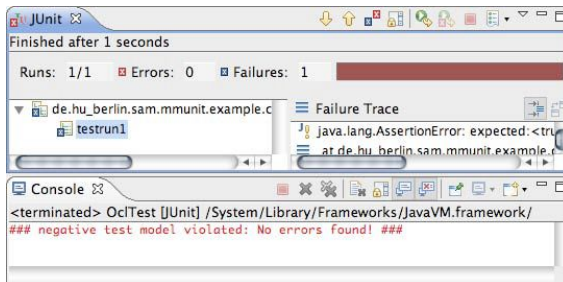


Fig. 9. The test specified in Fig. 8 revealed an error in the metamodel

The object `p3:Project` is forbidden, which is depicted by the dashed, gray border. Consequently, the test specification represents a positive test model without `p3:Project`. Adding it to the test model results in a negative test model. Again, this test is passed for the company metamodel because it specifies that each employee should be assigned to at most two projects.⁴

Both employees of the test specification are enforced and they are connected by a conjunction, which is depicted by a “^” in a circle. Removing *both* employees leads to a negative test model. Again, the company metamodel fulfills this test because it specifies that each company must have at least one employee.

In Fig. 8, a different test specification is shown. The goal of this test specification is to validate the OCL expression of the MMUT: The intention of the metamodel engineer was to express that the insured sum of each boss in the company is at least 1.000.000. Consequently, the test in Fig. 8 contains a forbidden

⁴ Alternatively, the reference between the objects `e1:Employee` and `p3:Project` could have been set to forbidden. In this case, `p3:Project` would be part of the positive test model but it would not be referenced by the Employee.

relationship between the employees Deng and Chang: If Chang was the boss of Deng, then Chang’s insured sum would have to be at least 1.000.000. However, the insured sum of Chang is only 500.000. Consequently, the addition of this boss-relationship has to cause an error and, therefore, it has to be forbidden. All references to the MMUT can be resolved, all multiplicities of the test model are valid. However, the corresponding test case fails (shown in Fig. 9) because the corresponding negative test model causes no error. The reason is the described error in the OCL expression of the MMUT. The metamodel engineer recognizes the error and corrects the OCL expression, which should contain “`e.boss = self`” instead of “`self.boss = e`”. After correction, all metamodel tests pass.

5 Metamodel Test Execution

In this section, we describe how metamodel tests specified in a test specification are executed. The test process consists of 5 steps:

1. *Derive test models from the test specification.*

The metamodel tests are specified in test specifications. For testing, test models have to be derived from the test specification according to the function τ . Figure 10 shows a pseudo-code definition of this function.

We presented our approach as describing positive and negative potential instances and checking if they are elements of Θ_{MMUT} . Each test specification can specify corresponding positive and negative test models. The value of the attribute `TestModel.positive` strongly influences the way these test models are derived. For instance, `TestModel.positive=true` and the exclusion of all forbidden elements specifies a positive test model. The same test model with `TestModel.positive=false` is a negative test model. Consequently, the type of the test models specified by the test specification depends on `TestModel.positive`. For reasons of conciseness, we describe the test execution for `TestModel.positive=true`. The case `TestModel.positive=false` is handled likewise.

The positive test model is derived from the test specification by leaving out all forbidden elements. A separate negative test model is derived for each conjunction in the test specification and for each enforced or forbidden element that is not connected to a conjunction: Let c be a conjunction of elements and e a single element for each of which a negative test model is to be derived. Then, a negative test model for e or c is created as follows: All elements of the test specification except conjunctions are copied to a new instance of TMM, which is the new negative test model. Furthermore, all forbidden elements except e or except the referenced elements of c are removed from the new negative test model. If e or a referenced element of c is enforced, then it is also removed from the negative test model.

2. *For each test model: Resolve references of the test models to the MMUT.*

Each test model references the elements of the MMUT by name. In this step, it is checked if all references can be resolved to elements of the MMUT. If a reference of a positive test model cannot be resolved, the test fails. On the other hand, an unresolved reference is sufficient for a test with a negative test model to pass.

```

τ(model) {
  testmodels = Set();

  — Create positive test model.
  positiveTestmodel = model.clone();
  — Remove all forbidden elements from positive test model.
  positiveTestmodel->remove(positiveTestmodel->allElements()
                           ->select( e | e.existence = forbidden ));
  testmodels += positiveTestmodel;

  — Create negative test models for all elements not part of a conjunction.
  foreach element in (model.elements - model.conjunctions.elements)
    ->select( e | e.existence <> arbitrary ) {
    testmodels += buildTestmodel(model, Set(element));
  }

  — Create negative test models for all conjunctions.
  foreach conjunction in model.conjunctions {
    testmodels += buildTestmodel(model, conjunctions.elements
                                ->select( e | e.existence <> arbitrary ));
  }

  return testmodels;
}

— Helper function that creates a negative test model according to
— the given elements with existence specifications.
buildTestmodel(model, elementsWithExistenceSpec) {
  result = model.clone();
  result.positive = not result.positive;
  — Remove all enforced elements from negative test model.
  result->remove(elementsWithExistenceSpec
                ->select( e | e.existence = enforced ));
  return result;
}

```

Fig. 10. Pseudo-code definition of τ

3. *For each test model: Create a corresponding instance of the MMUT.*

After successfully passing the second step for a test model without detecting an error, it is possible to create a corresponding instance of the MMUT. For that, all elements of the MMUT referenced by the test model are instantiated and all attributes and references are set. This step discovers if, e.g., a default value of an attribute is an instance of the attribute type and can be assigned to the attribute.

4. *For each test model: Check multiplicities and constraints of the created instance of the MMUT.*

Other than the resolved classes, attributes, and references, the MMUT contains additional information like multiplicities and OCL constraints. The created instance from Step 3 is now used to validate these information. For the concrete prototype implementation, several supporting validation frameworks can be used (cf. Sec. 6).

5. For each test model and each test specification: Decide test outcome.

The results of the preceding steps are now combined: If the test for a negative test model fails any of the preceding steps or the test for a positive test model succeeds in all of them, the corresponding test is passed. Otherwise, the test fails. Since a test specification specifies a set of test models, the test outcome for the test specification is composed of the outcomes for the test models. Likewise, if the tests for all specified test models are passed, then the test for corresponding test specification is passed. Otherwise, it fails.

6 MMUnit: Implementation

We implemented our approach based on the Eclipse Modeling Framework (EMF). The implemented metamodels are based on Ecore, EMF's meta-metamodel, which is similar to Essential MOF (EMOF). Since $\Theta_{TMM} \subset \Theta_{TSMM}$, we restricted the implementation to the use of TSMM: test models are implemented as instances of TSMM without conjunctions. The resulting prototype *MMUnit*⁵ provides an editor for test specifications (Fig. 7 and Fig. 8) and can be used to generate a JUnit [6] test for each test specification. The generated JUnit tests (Fig. 11) use a library of MMUnit containing the class *MMTester*. This class implements the test process almost as described in Sec. 5. Since these attributes just influence the numbers of generated positive and negative test models, this means no restriction to our approach. The important features like the specification of test models in one test specification, the definition of potential instances, and the validation of OCL expressions are implemented. To validate the OCL expressions of the MMUT, the Eclipse Validator Framework is used. MMUnit also allows to combine the execution of several test specifications in one JUnit class: Corresponding entries in the context menu of the test specifications and a dialog provide possibilities to choose a set of test models.

```
@Test
public void testrun1() {
    MMTester oMMTester = new MMTester();
    oMMTester.setTestModel("model/tests/example_paper.mmunit");
    oMMTester.setMetaModelFile("model/companymm.ecore");
    assertTrue(oMMTester.runTest());
}
```

Fig. 11. The generated JUnit Code

7 Related Work

Many approaches in model-based testing use models as specifications to generate test cases for a system under test (SUT) [7,8,9,10]. The tests validate if the SUT

⁵ Available for download at <http://mmunit.sf.net>

satisfies all constraints of the model. The models themselves are assumed to be correct. In contrast to this, we want to test the correctness of metamodels.

Küster [11] validates model transformations. Wang et al. [12] verify metamodel coverage of model transformations. Brottier et al. [13] generate metamodel based tests for model transformations. They all assume that the used metamodels are correct and they focus on testing the transformation process between them. Our approach is complementary to their approaches, as it tests the metamodels they assume to be correct. Especially for model transformations, a combination of both approaches seems to be promising.

In grammar testing [14], character sequences are used to test a developed grammar [15]. This generic approach permits to define both words that conform to the grammar and words that do not. Similar to that, our test metamodel allows to generically describe instances of metamodels.

In the field of software engineering, many frameworks arose that support development and testing. Development is supported by, e.g., tools for the Eclipse platform [16,17]. Unit tests are supported by the frameworks JUnit [6] and NUnit [18]. In our work, we created a prototype that is based on Eclipse and that supports JUnit. The integration of MMUnit into existing and widely used frameworks simplifies the integration into existing development projects.

8 Conclusion

Contribution. In all model-driven development processes, metamodels play an important role. In this paper, we considered the correctness of metamodels and identified metamodel testing as a method to reduce the effort for correcting erroneous metamodels and dependent tools and models.

Our contribution is an approach for automated and systematic metamodel testing. Our approach is founded on regarding metamodels as set specifications. We described an algorithm for the test process, implemented a corresponding prototype, and demonstrated our approach with a metamodel for a company.

By the integration with JUnit, metamodel tests can be executed automatically and test execution can be integrated into the existing software development process (e.g. metamodel tests can be executed as part of a continuous integration build). Metamodel tests are not restricted to validate new metamodels. Rather, they can be used as regression tests for evolving metamodels [19]. This would help MDA engineers to understand the effects of changing a metamodel. Thus, metamodel tests make it less likely for MDA engineers to introduce new errors.

The main benefit of our approach is a way to test a metamodel independently of the models and tools that depend on it. This allows for a test-driven development with “test first” and, therefore, helps reducing development and maintenance effort. To our knowledge, there is no tool support for testing metamodels besides MMUnit.

Future Work. Textual and graphical editors can be built for metamodels. An editor should allow a user to create arbitrary instances of the metamodel but

should warn him of—or even prevent him from—creating models that are not instances of the metamodel. Our approach could be extended to test editors, as well. For this, test models could be transformed to a series of creation commands for the editor under test.

In this paper, we did not deal with the quality of test specifications. The quality of tests in general can be measured with coverage criteria. We plan to adapt coverage criteria for metamodel testing from other fields. For instance, the work about adequacy criteria for UML design models by Andrews et al. [20] may be a starting point.

Our implementation MUnit should be improved regarding error reporting. Currently errors are only reported on the console. Instead, they could be visualized in the test specification editor or in the metamodel editor.

Finally, systematic case studies are necessary to answer these questions: How can metamodel testing be integrated into an overall MDA process? Is creating metamodel tests profitable—in terms of quality and in economic terms?

Acknowledgments. We would like to thank Sebastian Schuster for comments on a preliminary version of this paper and the anonymous reviewers for valuable comments. This work was supported by grants from the DFG (German Research Foundation, research training group METRIK).

References

1. Miller, J., Mukerji, J.: MDA Guide Version 1.0.1. Technical report, Object Management Group (OMG) (2003)
2. Object Management Group: Meta Object Facility (MOF) 2.0 Core Specification (2006)
3. Beck, K.: Test Driven Development: By Example. Addison-Wesley Professional, Reading (2002)
4. Beizer, B.: Software Testing Techniques. John Wiley & Sons, Inc, Chichester (1990)
5. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann Publishers Inc., San Francisco (2006)
6. Sourceforge: JUnit (2008), <http://junit.sourceforge.net>
7. Abdurazik, A., Offutt, J.: Using UML Collaboration Diagrams for Static Checking and Test Generation. In: Evans, A., Kent, S., Selic, B. (eds.) UML 2000. LNCS, vol. 1939. Springer, Heidelberg (2000)
8. Nebut, C., Fleurey, F.: Automatic test generation: A use case driven approach. IEEE Trans. Softw. Eng. 32(3), 140–155 (2006)
9. Offutt, J., Abdurazik, A.: Generating Tests from UML Specifications. In: France, R.B., Rumpe, B. (eds.) UML 1999. LNCS, vol. 1723. Springer, Heidelberg (1999)
10. Prenninger, W., Pretschner, A.: Abstractions for Model-Based Testing. Electr. Notes Theor. Comput. Sci. 116, 59–71 (2005)
11. Küster, J.M.: Definition and validation of model transformations. Software and Systems Modeling V5(3), 233–259 (2006)
12. Wang, J., Kim, S.K., Carrington, D.: Verifying metamodel coverage of model transformations. In: ASWEC 2006 (2006)

13. Brottier, E., Fleurey, F., Steel, J., Baudry, B., Traon, Y.L.: Metamodel-based test generation for model transformations: an algorithm and a tool. In: ISSRE 2006: Proceedings of the 17th International Symposium on Software Reliability Engineering, Washington, DC, USA, pp. 85–94. IEEE Computer Society, Los Alamitos (2006)
14. Purdom, P.: A sentence generator for testing parsers. *bit* 12(3), 366–375 (1972)
15. Lämmel, R.: Grammar adaptation. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 550–570. Springer, Heidelberg (2001)
16. Eclipse Foundation: Eclipse (2008), <http://www.eclipse.org>
17. Eichler, H., Sadilek, D.A., Scheidgen, M., Soden, M., Wachsmuth, G., Weißleder, S.: Frameworks to create language definitions and tools on top of the eclipse modelling project (poster). In: EclipseCon 2008 (2008)
18. NUnit.org: NUnit (2008), <http://www.nunit.org>
19. Favre, J.M.: Meta-model and model co-evolution within the 3D software space. In: ELISA 2003, pp. 98–109 (2003)
20. Andrews, A.A., France, R.B., Ghosh, S., Craig, G.: Test adequacy criteria for uml design models. *Softw. Test., Verif. Reliab.* (2003)

A Metamodeling Approach for Reasoning about Requirements

Arda Goknil, Ivan Kurtev, and Klaas van den Berg

Software Engineering Group, University of Twente, 7500 AE Enschede, the Netherlands
{a.goknil, kurtev, k.g.van.den.berg}@ewi.utwente.nl

Abstract. In requirements engineering, there are several approaches for requirements modeling such as goal-oriented, aspect-driven, and system requirements modeling. In practice, companies often customize a given approach to their specific needs. Thus, we seek a solution that allows customization in a systematic way. In this paper, we propose a metamodel for requirements models (called *core metamodel*) and an approach for customizing this metamodel in order to support various requirements modeling approaches. The core metamodel represents the common concepts extracted from some prevalent approaches. We define the semantics of the concepts and the relations in the core metamodel. Based on this formalization, we can perform reasoning on requirements that may detect implicit relations and inconsistencies. Our approach for customization keeps the semantics of the core concepts intact and thus allows reuse of tools and reasoning over the customized metamodel. We illustrate the customization of our core metamodel with SysML concepts. As a case study, we apply the reasoning on requirements of an industrial mobile service application based on this customized core requirements metamodel.

Keywords: requirements metamodels, reasoning, model customization.

1 Introduction

Model Driven Engineering (MDE) considers models as primary engineering artifacts throughout the software development [11]. A software system is specified as a set of models that are repetitively refined until a model with enough details to implement the system is obtained.

Software development has different phases (requirement analysis, architectural design, detailed design, implementation and testing) which result in different artifacts. Currently, there exist standard modeling languages for expressing architecture, detailed design, and implementation of systems. Requirements descriptions, however, are considered mostly as textual artifacts with structure often not explicitly specified. Requirements descriptions are one of the earliest models of a system. In order to keep the continuum of models in MDE by treating every artifact as a model we need to represent requirements descriptions as models as well. To achieve this, developers need to employ a metamodel for requirements. However, it is difficult to propose a single and eventually standardized metamodel for requirements. There are several

commonly used approaches to represent requirements: goal-oriented [27] [16], aspect-driven [20], variability management [16], use case [3], domain-specific [12], and reuse-driven techniques [13]. Goal-oriented approach [27] defines a model for decomposing system goal into requirements with goal trees and offers some decision methods based on this decomposition. Aspect-oriented approach [20] gives a requirements model for separation of crosscutting functional and non-functional properties in requirements analysis phase.

A possible approach is to extract the common concepts from the existing techniques into a single metamodel. The current state of the requirements engineering practice shows that companies often adapt and customize a given approach to the company's specific needs. Thus, we need a solution that will allow us to achieve generality by using a set of common concepts and to allow customization in a systematic way.

In this paper, we propose a metamodel for requirements models (called *core* metamodel) and suggest an approach for customizing this metamodel in order to support different requirements specification techniques. We define the semantics of the concepts and the relations in the core metamodel. On the basis of the semantics we can perform reasoning on requirements that may detect implicit relations and inconsistencies. Furthermore, our approach for customization keeps the semantics of the core concepts intact and thus allows reuse of tools and reasoning over the customized metamodel.

The core metamodel represents the common concepts extracted from some existing requirements modeling approaches [27] [15] [16] [18] [20] [28]. The customization of the core metamodel is based on set-theoretic operations. This ensures the validity of the results inferred from the customized requirements models by using the reasoning rules defined for the core metamodel. In the core metamodel we give the building blocks of a requirements specification. We are not interested in giving the details of requirements such as dynamic properties of target systems. Requirements engineer can always come up with his/her domain specific language for different types of requirements such as real-time specifications of embedded systems.

We illustrate our approach by customizing the core metamodel with SysML constructs. As a case study we model the requirements of an industrial mobile service application based on the customized metamodel.

The paper is structured as follows. In Section 2, we describe the customization approach for the requirements metamodels. Section 3 gives the details of the core requirements metamodel with the inference rules. Section 4 gives the details of SysML requirements metamodel. In Section 5 we describe the mappings between the two requirements metamodels. We also give the customized core requirements metamodel for SysML. In Section 6, we give a case study to illustrate the customization. Section 7 presents the related work. Section 8 summarizes the paper and describes future work.

2 Overview of the Customization Approach

In our approach the requirements engineer starts with the core requirements metamodel (See Fig. 1) and identifies the concepts that need specialization and concepts that has to be added. The result of the customization is a new requirements

metamodel. In Fig. 1, we use SysML as an example metamodel that specializes the core metamodel. The plus operator denotes the specification of the relations between the elements in the metamodels. These relations are based on set operations. An example is given in Section 5. Other metamodels for different approaches such as goal-oriented and aspectual requirements can be composed with the core requirements metamodel.

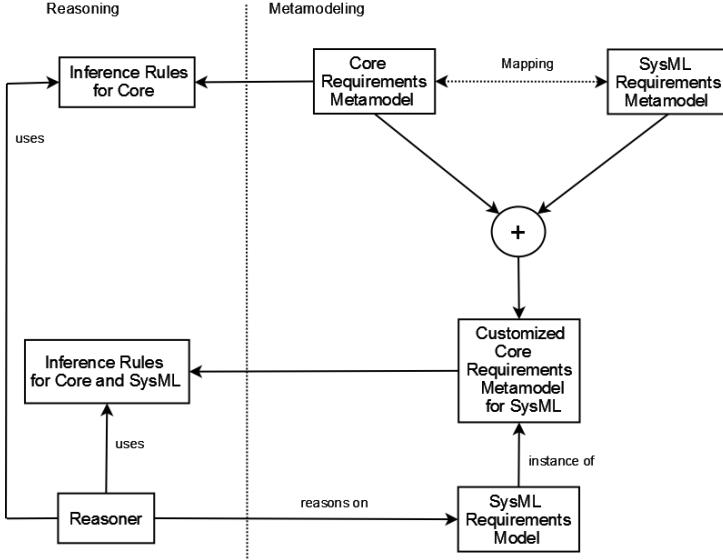


Fig. 1. Customization of Requirements Metamodels

In this paper we express the metamodels as OWL [4] ontologies. The composition operator is also expressed in OWL since this language allows direct mapping from set operations to the language constructs. By using OWL we can use the reasoning capabilities of the ontology tools. The aim of the approach is to specify generic inference rules for the core metamodel and to apply them for the customized metamodels (see left part of **Fig. 1**). Additional inference rules, specific for a given metamodel, may be added if needed.

3 Core Requirements Metamodel

The core requirements metamodel contains common concepts identified in existing requirements modeling approaches [27] [15] [16] [18] [20] [28]. The core metamodel in Fig. 2 includes entities such as *Requirement*, *Stakeholder* and *Relationship* in order to model general characteristics of requirements artifacts. They serve as extension points for possible customizations of the core metamodel. In this metamodel, all requirements are captured in a requirements model (*RequirementModel*). A requirements model is characterized by a name property and contains requirements instances of the *Requirement* entity. All requirements have a unique identifier (ID

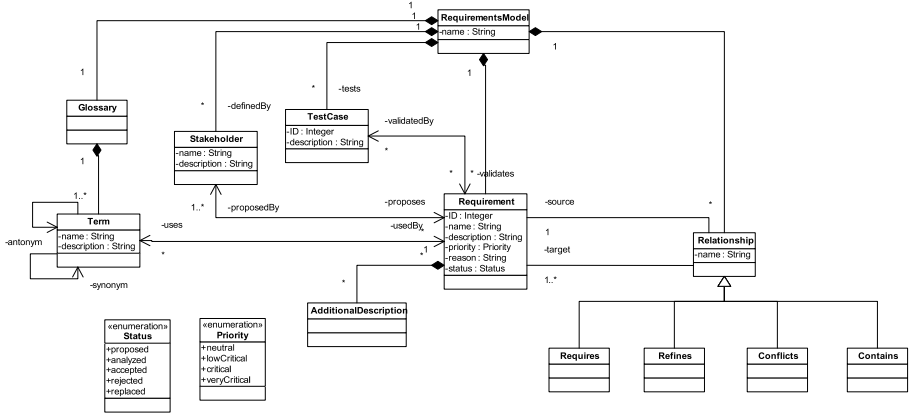


Fig. 2. Core Requirements Metamodel

property), a name, a textual description (description property), a priority, a rationale (reason property), and a status. Requirements may have additional descriptions (*AdditionalDescription* entity) such as a use case or any other formalization.

Usually, requirements are classified as functional and non-functional requirements. Since there might be different classifications of requirements for different approaches, we decided not to give any further specialization of the *Requirement* concept in the core metamodel: this can be added in the customization. Requirements can be related with each other. We recognize four types of relations: *Refines*, *Requires*, *Conflicts*, and *Contains*. These core relations can be specialized and new relations may be added as specializations of the *Relationship* concept. The metamodel includes the entities *Stakeholder*, *TestCase*, *Glossary* and *Term*. Test cases are not always considered as parts of requirements specifications. However, they are important to validate or verify requirements. Some metamodels [18] [28] consider test cases as a part of the requirements specification.

In order to specify relations between core and other requirements metamodels we give a set-theoretic interpretation of the core entities.

Let Core Requirements Metamodel (CRM) = {R, RS, RF, RQ, CF, CT, SH, TC, GS, T, AD} where the following abbreviations for the entities are used:

AD: AdditionalDescription	R: Requirement	SH: Stakeholder
CF: Conflicts	RF: Refines	T: Term
CT: Contains	RQ: Requires	TC: TestCase
GS: Glossary	RS: Relationship	

We assume that (a): all relations between requirements are the subset of relationship and (b): the intersection of these four relations is an empty set and the *Refines* relation is a subset of the *Requires* relation.

$$a : (RF \subseteq RS) \wedge (RQ \subseteq RS) \wedge (CF \subseteq RS) \wedge (CT \subseteq RS)$$

$$b : RF \cap RQ \cap CF \cap CT \equiv \emptyset \quad \wedge \quad RF \subseteq RQ$$

The relations in the core metamodel are defined and formalized as follows.

- *Definition 1. Requires relation:* A requirement R_1 *requires* a requirement R_2 if R_1 is fulfilled only when R_2 is fulfilled. R_2 can be treated as a pre-condition for R_1 [28].
- *Definition 2. Refines relation:* A requirement R_1 *refines* a requirement R_2 if R_1 is derived from R_2 by adding more details to it [27].
- *Definition 3. Contains relation:* A requirement R_1 *contains* requirements $R_2..R_n$ if R_1 is the conjunction of the contained requirements $R_2..R_n$. This relation enables a complex requirement to be decomposed into child requirements [18].
- *Definition 4. Conflicts relation:* A requirement R_1 *conflicts with* a requirement R_2 if the fulfillment of R_1 excludes the fulfillment of R_2 and vice versa [26].

The definitions given above are intuitive and informal. In the remaining part of this section we give a formal definition of requirements and relations among them in order to ensure sound inference rules.

We assume the general notion of requirement being “a property which must be exhibited by a system” [7]. We define a requirement R as a tuple $\langle P, S \rangle$ where P is a predicate (the property) and S is a set of systems that satisfy P , i.e. $\forall s \in S : P(s)$.

▪ Formalization of Requires

Let R_1 and R_2 are requirements such that $R_1 = \langle P_1, S_1 \rangle$ and $R_2 = \langle P_2, S_2 \rangle$. R_1 *requires* R_2 iff for every $s_1 \in S_1$ then $s_1 \in S_2$.

From this definition we conclude that $S_1 \subset S_2$. The subset relation between the systems S_1 and S_2 gives us the properties of *non-reflexive*, *non-symmetric*, and *transitive* for the *requires* relation.

▪ Formalization of Refines

Let R_1 and R_2 are requirements such that $R_1 = \langle P_1, S_1 \rangle$ and $R_2 = \langle P_2, S_2 \rangle$. We assume that P_1 and P_2 are formulas in first order logic (there may be formalizations of requirements in other types of logics such as modal and deontic logic [14]) and P_2 can be represented in a conjunctive normal form in the following way:

$$P_2 = p_1 \wedge p_2 \wedge \dots \wedge p_{n-1} \wedge p_n \wedge q_1 \wedge q_2 \wedge \dots \wedge q_{m-1} \wedge q_m$$

Let $q_1^1, q_2^1, \dots, q_{m-1}^1, q_m^1$ are the predicates such that $q_i^1 \rightarrow q_i$ for $i \in 1..m$

R_1 *refines* R_2 iff P_1 is derived from P_2 by replacing every q_i in P_2 with q_i^1 $i \in 1..m$ such that the following two statements hold:

- (a) $P_1 = p_1 \wedge p_2 \wedge \dots \wedge p_{n-1} \wedge p_n \wedge q_1^1 \wedge q_2^1 \wedge \dots \wedge q_{m-1}^1 \wedge q_m^1$
- (b) $\exists s \in S_2 : s \notin S_1$

From the definition we conclude that if P_1 holds for a given system s then P_2 also holds for s . Therefore $S_1 \subset S_2$. Similarly to the previous relation we have the properties *non-reflexive*, *non-symmetric*, *transitive* for the *refines* relation. Obviously, if R_1 *refines* R_2 then R_1 *requires* R_2 .

▪ Formalization of Contains

Let R_1 , R_2 and R_3 are requirements such that $R_1 = \langle P_1, S_1 \rangle$, $R_2 = \langle P_2, S_2 \rangle$, and $R_3 = \langle P_3, S_3 \rangle$. We assume that P_2 and P_3 are formulas in first order logic and can be represented in a conjunctive normal form in the following way:

$$P_2 = p_1 \wedge p_2 \wedge \dots \wedge p_{m-1} \wedge p_m$$

$$P_3 = p_{m+1} \wedge p_{m+2} \wedge \dots \wedge p_{n-1} \wedge p_n$$

R_1 contains R_2 and R_3 iff P_1 is derived from P_2 and P_3 as follows:

$P_1 = P_2 \wedge P_3 \wedge P'$ where P' denotes properties that are not captured in P_2 and P_3 (i.e. we do not assume completeness of the decomposition [27])

From the definition we conclude that if P_1 holds then P_2 and P_3 also hold. Therefore, $S_1 \subset S_2$ and $S_1 \subset S_3$. Obviously, the *contains* relation is *non-reflexive*, *non-symmetric*, and *transitive*.

▪ Formalization of Conflicts

Let R_1 and R_2 are requirements such that $R_1 = \langle P_1, S_1 \rangle$ and $R_2 = \langle P_2, S_2 \rangle$. Then, R_1 *conflicts with* R_2 iff $\neg \exists s : s \in S_1 \wedge s \in S_2 : P_1(s) \wedge P_2(s)$. The *conflicts* relation is *symmetric*.

It should be noted that the definition of *requires* is given in extensional terms as a subset relation between the systems that satisfy the requirements. The definitions of *refines* and *contains* are given in intensional terms, that is, they take into account the form of the requirement specification as a predicate. If we would interpret *refines* in an extensional way then we will conclude that *requires* and *refines* are both interpreted as a subset relation and therefore are equivalent. Apparently in our formalization, *refines* and *requires* are different.

From the given definitions we may infer several rules that show how these three relations can be combined. We explore all combinations of requirements relations in the core metamodel in order to derive inference rules for requirements. Due to space limitation we do not give all combinations and inference rules for the relations. The rules are expressed in Semantic Web Rule Language (SWRL) [9] since OWL is not expressive enough in this case. The following example illustrates some of the rules on the basis of a concrete requirements specification document given in the WASP framework [21]. The example requirements (see Case Study in Section 6) are:

- *REQ_BDS_007*: When changes are discovered in the status and/or location of a user's body, the WASP platform must sent out notifications according to the alerts set by the user.
- *REQ_NOT_006*: The WASP platform must notify the end-user about the occurrence of an event for which an alert was set, as soon as the event occurs.
- *REQ_NOT_009*: The WASP platform must actively monitor all events.

In the requirements document, the following relations are given: *refines* (*REQ_BDS_007*, *REQ_NOT_006*) and *requires* (*REQ_NOT_006*, *REQ_NOT_009*). When we apply the inference rules to the given requirements, we have inferred that *REQ_BDS_007* also requires *REQ_NOT_009* (dashed line in Fig. 3).

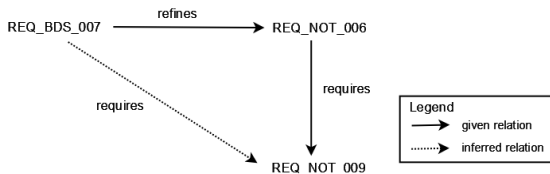


Fig. 3. Example with Given and Inferred Relations

We can formalize and proof these rules as follows:

Rule 1: $\text{refines}(R_1, R_2) \wedge \text{requires}(R_2, R_3) \rightarrow \text{requires}(R_1, R_3)$

Proof: Let $R_2 = \langle P_2, S_2 \rangle$ where $P_2 = p_1 \wedge p_2 \wedge \dots \wedge p_{n-1} \wedge p_n$ and $R_1 = \langle P_1, S_1 \rangle$. Since R_1 *refines* R_2 , from the definition we have that $P_1 = p_1 \wedge p_2 \wedge \dots \wedge p_{n-1} \wedge p_n \wedge q'_1 \wedge q'_2 \wedge \dots \wedge q'_{m-1} \wedge q'_m$ and $q'_i \rightarrow q_i$ $i \in 1..m$. Again from the definition we have that if P_1 holds then P_2 also holds. From the *requires* relation between R_2 and R_3 we have that $S_2 \subset S_3$. Therefore if P_2 holds then P_3 also holds. Now we may conclude that if P_1 holds then P_3 also holds. This gives the subset relation $S_1 \subset S_3$ which proves that R_1 *requires* R_3 .

Rule 2: $\text{contains}(R_1, R_2) \wedge \text{requires}(R_2, R_3) \rightarrow \text{requires}(R_1, R_3)$

Proof: Let $R_1 = \langle P_1, S_1 \rangle$, $R_2 = \langle P_2, S_2 \rangle$, and $R_3 = \langle P_3, S_3 \rangle$

Since R_1 *contains* R_2 we have $S_1 \subset S_2$. From R_2 *requires* R_3 it follows that $S_2 \subset S_3$. Consequently $S_1 \subset S_3$. Similarly to the previous proof, we conclude that R_1 *requires* R_3 .

We can have implications for more combinations (e.g. three relations for four requirements and two conjunction operators) by using these inference rules.

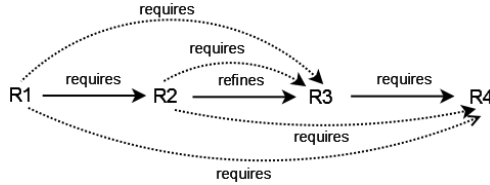


Fig. 4. Example with Inferred Relations by Combining Inference Rules

The relations shown with dash lines in Fig. 4 are inferred by using Rule 1, the transitivity of the relations, and the fact that *refines* implies *requires*. By combining these rules we have the following indirect relations:

$$\begin{aligned} & \text{requires}(R_1, R_2) \wedge \text{refines}(R_2, R_3) \wedge \text{requires}(R_3, R_4) \rightarrow \\ & \text{requires}(R_2, R_3) \wedge \text{requires}(R_2, R_4) \wedge \text{requires}(R_1, R_3) \wedge \text{requires}(R_1, R_4) \end{aligned}$$

Several rules for consistency checking are derived from the basic combinations where there is only one relation between two requirements. These inconsistencies are different from *conflicts* relation between requirements. Inconsistencies here indicate that relations between requirements are violating their constraints. Some of the consistency rules are given below:

- $\text{refines}(x_1, x_2) \rightarrow \neg \text{refines}(x_2, x_1)$
- $\text{refines}(x_1, x_2) \rightarrow \neg \text{requires}(x_2, x_1)$
- $\text{refines}(x_1, x_2) \rightarrow \neg \text{contains}(x_2, x_1)$

We specified OWL [4] ontologies for each metamodel with Protégé [6] environment. Inference rules were expressed in SWRL [9]. The rules to check the consistency of relations were implemented as SPARQL [24] queries. The inference rules are executed by Jess rule engine [10] available as a plug-in in Protégé. To reason

We assume that (a): Requirements types in SysML are subsets of *ExtendedReq* which is a subset of *Requirement*, (b): The intersection of all these requirements types is an empty set (they are disjoint), (c): Relations between requirements are the subset of relationship *Trace*, (d): the intersection of these relations is an empty set, (e): *UseCase* is a subset of *AdditionalDescription*, (f): Relations between use cases are the subset of relationship *UseCaseRelation*, and (g): the intersection of the relations between use cases is an empty set.

$$a : (IR \subseteq ER) \wedge (PR \subseteq ER) \wedge (FR \subseteq ER) \wedge (PSR \subseteq ER) \wedge (DC \subseteq ER) \wedge (ER \subseteq R)$$

$$b : IR \cap PR \cap FR \cap PSR \cap DC \equiv \emptyset$$

$$c : (DV \subseteq T) \wedge (CP \subseteq T) \wedge (CT \subseteq T)$$

$$d : DV \cap CP \cap CT \equiv \emptyset$$

$$e : UC \subseteq AD$$

$$f : (US \subseteq UCR) \wedge (SC \subseteq UCR) \wedge (EX \subseteq UCR)$$

$$g : US \cap SC \cap EX \equiv \emptyset$$

We introduce the following inference rules specific for SysML and not defined for the core metamodel. The relations *uses*, *extends* and *specializes* are transitive. Transitivity is captured in Rule 1:

$$\text{Rule 1. } \text{uses}(uc_1, uc_2) \wedge \text{uses}(uc_2, uc_3) \rightarrow \text{uses}(uc_1, uc_3)$$

If two use-cases are related, derived requirements from these use-cases are also related. Rule 2 specifies this:

$$\text{Rule 2. } \text{uses}(uc_1, uc_2) \wedge \text{hasAdditionalDescription}(req_1, uc_1) \wedge \text{hasAdditionalDescription}(req_2, uc_2) \rightarrow \text{requires}(req_1, req_2)$$

Since the concepts of use case and *uses* relation are not precisely defined in SysML, the inference rules 1 and 2 express the intuitive meaning we assigned to them. The formalization of SysML concepts needs further investigation.

5 Mappings between the Core and SysML Metamodels

In order to customize the core metamodel with SysML constructs we establish mappings between the elements in these metamodels. Mappings are specified as relations on sets. Some elements like *Requirement* in the core and *Requirement* in SysML are mapped directly. However, some elements e.g. *Derive* from SysML has no corresponding element in the core metamodel. Table 1 shows the mappings between core and SysML metamodels.

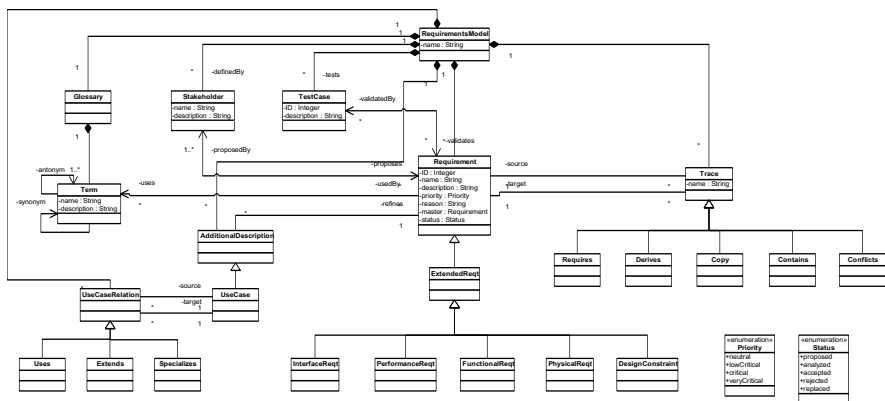
The semantically equivalent entities are related with set equality (e.g. rows 1, 5, 6). All specialized requirements in SysML are specializations of *Requirement* in the core metamodel (row 2). *Requires* (RQ) and *Conflicts* (CF) have no corresponding relation in SysML metamodel. All relations that have no corresponding relations in SysML metamodel are specializations of *Trace* (T) relation (rows 9 and 10). The relation

Table 1. Mapping between Core and SysML Requirements Metamodels

	Core Metamodel	Relation	SysML Requirements MM
1.	R	\equiv	R
2.	R	\supseteq	$IR \cup PR \cup FR \cup PSR \cup DC$
3.	$RF \cup RQ \cup CF \cup CT$	\subseteq	T
4.	RS	\supseteq	$DV \cup CP \cup CT$
5.	RS	\equiv	T
6.	RF	\equiv	DV
7.	CT	\equiv	CT
8.	RS	\supseteq	CP
9.	RQ	\subseteq	T
10.	CF	\subseteq	T
11.	AD	\equiv	AD
12.	TC	\equiv	TC

Copy (CP) in SysML is mapped to a specialization of Relationship (RS) in core metamodel (row 8).

Customization operators are derived from the mappings given in Table 1. Two required operators are “equivalent class” and “sub-class”. They may be expressed in different ways depending on the technology. In OWL environment these operators correspond to *rdfs:equivalentClass*, *rdfs:subClassOf*, *rdfs:equivalentProperty*, and *rdfs:subPropertyOf*. Fig. 6 gives customized core requirements metamodel for SysML. This metamodel is the output of the customization process given in Fig. 1.


Fig. 6. Customized Core Requirements Metamodel for SysML

6 Case Study WASP Application Framework

In this section we apply the proposed approach in a case study. An existing requirements specification document is represented as a model instance of the customized metamodel from Section 5. The case study is about the requirements for

WASP (Web Architectures for Services Platforms), a framework for context-aware mobile services [21]. The requirements are identified using a three-step process of defining scenarios, use cases and requirements (see [21] for concrete details). There are 2 scenarios, 32 use cases and 81 requirements (70 functional and 2 non-functional;; three of these requirements are decomposed into 9 sub-requirements).

We compared the reasoning facilities available in our approach with the similar support provided by IBM Rational RequisitePro. RequisitePro provides only two relations between requirements: *traceFrom* and *traceTo*. The relations in the customized metamodel (e.g., the uses relation) must all be mapped to one of those two relations. For example, links from requirements to use cases are mapped to *traceTo* links in RequisitePro and to *hasAdditional-Description* in our framework.

There is no explicit indication in the WASP requirements document for requirements relations. However, there are some keywords in the document to reference to other requirements. These keywords are “see also”, “implies”, “implied by” and “extension of”. We mapped them to the available relations in RequisitePro and our framework (see Table 2). In the 4th column, we indicate our choice for the directionality, e.g. for “implies” and “implied-by”.

Table 2. Mapping of Requirements Relations in Case Study

Document	RequisitePro	Our Framework	Directionality w.r.t. document
R_1 see (also) R_2	R_1 traceTo R_2	R_1 requires R_2	both the same
R_1 implies R_2	R_1 traceTo R_2	R_1 requires R_2	both the same
R_1 implied by R_2	R_2 traceTo R_1	R_2 requires R_1	both reversed
R_1 extension of R_2	R_1 traceTo R_2	R_1 refines R_2	both the same
R_1 example in R_2	R_1 traceTo R_2	R_2 refines R_1	ours reversed

Individual requirements in the document were represented as individuals in the OWL ontology in Protégé. The execution of the inference rules with the Jess rule engine inferred the implicit relations between requirements in the document. We also executed consistency rules to check the requirements relations (both given and inferred). The Jess rule engine was executed in two steps: a) with inference rules written for only the core requirements metamodel, b) with inference rules written for the customized metamodel for SysML. Table 3 shows given and inferred facts for requirements document of the WASP application.

Table 3. Given and Inferred Facts for the WASP Application Requirements

Facts	# R	# UC	# Relations R x R	# Relations R x UC	# Relations UC x UC
Given	81	32	20	103	24
Inferred in Step a	0	0	5	0	0
Inferred in Step b	0	0	735	0	4

Reasoning on the core metamodel (step a) resulted in 5 inferred relations between requirements. Since we do not have any inference rule for use cases in the core metamodel, we do not have any inferred relations between use cases and use cases & requirements. We executed the rules to check the consistency of the given and inferred requirements relations. We did not detect any inconsistency for these relations. The result reflects the accuracy of relations given in the document regarding the relation definitions we use for the core metamodel. We also checked the inferred relations manually if they correspond to a relation that can be identified by analyzing the textual requirements document. We found one inferred relation that is not true. When we traced from the inferred relation back to the given relations, we found that one given relation in the ontology has not a correct mapping to the requirement relations in the document. This is due to the assumption that links “see (also)” represent “requires” relations. However, we found that one of these links actually corresponds to “refines” relation. Our conclusion is that often the requirements engineers use links with ambiguous meaning or the links are not applied systematically.

The execution of the inference rules added by the SysML requirements meta-models (step b) resulted in 735 inferred relations between requirements and 4 inferred relations between use cases. The consistency check detected 16 inconsistent relations. The analysis of these inconsistencies revealed that they are caused by Rule 2 in Section 4. Rule 2 implies that if two requirements are related to two different use cases and one of these use cases uses another one, then there should be a “requires” relation between these requirements. When we checked the given relations in the requirements document, we realized that the interpretation of the requirements engineer for “uses” relation is different. There are given “requires” relations between requirements whose use cases are not related each other with “uses cases”. Therefore, Rule 2 does not capture the document structure properly and does not reflect the understanding of the requirements engineer. In order to apply the rules in practice, we should give the precise definition for each relation to requirements engineer and offer a guideline about how to specify these relations for more accurate reasoning results.

We compared the results in our framework with the results in RequisitePro. Table 4 gives the given and inferred relations in RequisitePro and our framework.

We observe more inferred relations between requirements and use cases in RequisitePro than in our framework. RequisitePro infers links on the base of the

Table 4. Given and Inferred Relations in RequisitePro and Our Framework

relations	# Given	# Inferred	# Inferred
# UC = 32; # R = 81	Document	RequisitePro	Our Framework
UC x UC	24	3	3
UC x R	103	98	0
Step a: R x R	9	1	5
# inconsistencies	-	-	0
Step b: R x R	9	-	735
# inconsistencies	-	-	16

transitivity of *trace* relations without considering the linked artifacts. For example, it assumes transitivity between R_1 and UC in case of $R_1 \text{ traceTo } R_2 \text{ traceTo UC}$, which is debatable. RequisitePro does not define any specific types of relations. This prohibits sophisticated reasoning based on various relation types and leads to some wrong inferred relations as seen in $UC \times R$. In our framework, the relation types and the inference rules allow us to have more precise inferred relations. Having types for relations also avoids finding non-meaningful relations inferred by RequisitePro.

7 Related Work

Several authors address requirements modeling in the context of MDE. In [28] a metamodel and an environment based on it are described. The tool supports graphical requirements models and automatic generation of Software Requirements Specifications (SRS). Baudry et al. [1] introduce a metamodel for requirements and present how they use it on top of a constrained natural language for requirements definition. In [2] they propose a model-driven engineering mechanism to merge different requirement specifications and reveal inconsistencies between them by using their core requirement metamodel. However, their core metamodel is mainly used to produce a global requirements model from a given set of texts. It does not specify entities and core relations and does not support customization.

Some authors [8] [25] use UML profiling mechanism in goal-oriented requirements engineering approach. Heaven et al. [8] introduce a profile that allows the KAOS model [27] to be represented in UML. They also provide an integration of requirements models with lower level design models in UML. Supakkul et al. [25] use UML profiling mechanism to provide an integrated modeling language for functional and non-functional requirements that are mostly specified by using different notations. SysML [18] also uses UML profiling mechanism to provide modeling constructs that represent text-based requirements and relate them to other modeling elements.

Koch et al. [12] propose a requirements metamodel that is specific to web systems. They do not consider general concepts for requirements analysis. They identify the general structure of web systems in order to define the requirements metamodel. Rashid et al. [20] give an activity model in requirements engineering for identifying and separating crosscutting functional and non-functional properties. Moon et al. [15] propose a methodology of producing requirements that can be considered as a core asset in the product line. Lopez et al. [13] propose a metamodel for requirements reuse as a conceptual schema to integrate semiformal requirement diagrams into a reuse strategy. The requirements metamodel is used to integrate different abstraction levels for requirements definitions. Navarro et al. [17] propose a customization approach for requirements metamodels similar to ours. Their core metamodel is too generic and considers only artifact and dependency as core entities. It does not contain any entity specific to requirements. This prevents applying inference rules written for the core entities to customized entities. Requirements Interchange Format (RIF) [22] is a format which structures requirements and their attributes, types, access permissions and relationships. It is tool independent and defined as an XML schema. However, its data model has too generic entities and relations like *Information Type*, *Association*, and *Generalization* instead of entities that can be formalized to reason about requirements and their relations. Ramesh et. al [19] propose models for

requirements traceability. Models include basic entities like *Stakeholder*, *Object* and *Source*. Relations between different software artifacts and requirements are captured instead of core relations between requirements.

A number of approaches suggest reasoning about requirements. Zowghi et al. [29] propose a logical framework for modeling and reasoning about the evolution of requirements. Duff et al. [5] propose a logic-based framework for reasoning about requirements specifications based on goal-tree structures. Rodrigues et al. [22] propose a framework for the analysis of evolving specifications that can tolerate inconsistency by allowing reasoning in the presence of inconsistency.

8 Conclusion

There are several approaches for modeling requirements. These approaches are usually customized to serve specific needs and standards in industrial projects. In this paper, we proposed a metamodel for requirements and a customization approach in the context of Model Driven Engineering. Using metamodels for this customization allows us providing an environment for reuse of tools such as reasoners. The main concepts in our approach are the core requirements metamodel and the customization mechanism. We surveyed existing requirements modeling approaches to extract the core metamodel. We presented definitions and a formalization of requirements relations for the core metamodel. The customization mechanism is implemented on the basis of OWL properties in ontology.

We applied our approach in a case study based on a requirements specification document from a real project. We were able to infer several new relations that were not explicit in the document. We compared the capability of our approach to infer relations with the similar functionality provided by IBM RequisitePro, a commercial tool for requirements management. The relations in RequisitePro lack formal semantics. As a consequence, the inferred relations may not correspond to a “real” relation that may be discovered by inspecting the requirements document.

Since a wide range of inconsistencies can arise during requirements engineering, we did not elaborate on the *conflicts* relation in this paper. Some authors [26] review the main types of inconsistency and formalize them for specific cases. We plan to study definition and formalization of *conflicts* relation as future work. The impact of changes in requirements on inferred relations and checking the consistency of requirements against these changes are another future work in evolution dimension. For the evolution of requirements, we also want to analyze the impact of changes in requirements on architectural and detailed design. We need trace models to link requirement models to design models. These trace models will enable us to determine possible impacts of changes of requirements models on design models.

References

1. Baudry, B., Nebut, C., Le Traon, Y.: Model-Driven Engineering for Requirements Analysis. In: EDOC 2007, pp. 459–466. IEEE, Annapolis (2007)
2. Brottier, E., Baudry, B., Le Traon, Y., Touzet, D., Nicolas, B.: Producing a Global Requirement Model from Multiple Requirement Specifications. In: EDOC 2007, pp. 390–404. IEEE Computer Society Press, Annapolis (2007)

3. Cockburn, A.: *Writing Effective Use Cases*. Addison-Wesley, Reading (2000)
4. Dean, M., Schreiber, G., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D., Patel-Schneider, P., Stein, L.A.: *OWL Web Ontology Language Reference W3C Recommendation* (2004)
5. Duffy, D., MacHish, C., McDermid, J., Morris, P.: *A Framework for Requirements Analysis Using Automated Reasoning*. In: Iivari, J., Rossi, M., Lyytinen, K. (eds.) *CAiSE 1995*. LNCS, vol. 932. Springer, Heidelberg (1995)
6. Gennari, J., Musen, A., Fergerson, R.W., Grosso, W.E., Crubezy, M., Eriksson, H., Noy, N.F., Tu, S.W.: *The Evolution of Protégé: An Environment for Knowledge-Based Systems Development*. *International Journal of Human-Computer Studies* 58(1), 89–123 (2003)
7. *Guide to Software Engineering Body of Knowledge*. IEEE Computer Society, Los Alamitos (last visit 06.02.2008), <http://www.swebok.org/>
8. Heaven, W., Finkelstein, A.: *UML Profile to Support Requirements Engineering with KAOS*. *IEE Proceedings – Software* 151(1) (February 2004)
9. Horrocks, I., Patel-Schneider, P., Boley, H., Tabet, S., Grosof, B., Dean, M.: *SWRL: Semantic Web Rule Language – Combining OWL and RuleML*. W3C (May 2004)
10. Jess, the Rule Engine for the Java Platform, <http://herzberg.ca.sandia.gov/>
11. Kent, S.: *Model Driven Engineering*. In: *Proceedings of the 3rd International Conference on Integrated Formal Methods*, London, UK, pp. 286–298 (2002)
12. Koch, N., Kraus, A.: *Towards a Common Metamodel for the Development of Web Applications*. In: Cueva Lovelle, J.M., Rodríguez, B.M.G., Gayo, J.E.L., Ruiz, M.d.P.P., Aguilar, L.J. (eds.) *ICWE 2003*. LNCS, vol. 2722, pp. 497–506. Springer, Heidelberg (2003)
13. Lopez, O., Laguna, M.A., Garcia, F.J.: *Metamodeling for Requirements Reuse*. In: *Anais do WER 2002 - Workshop em Engenharia de Requisitos*, Valencia, Spain (2002)
14. Meyer, J.J.C., Wieringa, R., Dignum, F.: *The Role of Deontic Logic in the Specification of Information Systems*. *Logics for Databases and Information Systems*, 71–115 (1998)
15. Moon, M., Yeom, K., Chae, H.S.: *An Approach to Developing Domain Requirements as a Core Asset Based on Commonality and Variability Analysis in a Product Line*. *IEEE Transactions on Software Engineering* 31(7) (2005)
16. Mylopoulos, J., Chung, L., Yu, E.: *From Object-Oriented to Goal Oriented Requirements Analysis*. *Communications of the ACM* 42(1) (1999)
17. Navarro, E., Mocholi, J.A., Letelier, P., Ramos, I.: *A Metamodeling Approach for Requirements Specification*. *The Journal of Computer Information Systems* 46(5), 67–77 (2006)
18. *OMG: SysML Specification* OMG ptc/06-05-04, <http://www.sysml.org/specs.htm>
19. Ramesh, B., Jarke, M.: *Toward Reference Models for Requirements Traceability*. *IEEE Transactions on Software Engineering* 27(1) (2007)
20. Rashid, A., Moreira, A., Araujo, J.: *Modularization and Composition of Aspectual Requirements*. In: *AOSD 2003*, Boston, United States, pp. 11–20 (2003)
21. *Requirements for the WASP application Framework*, https://doc.telin.nl/dsweb/Get/Document-27861/WASP_D2.1_version_1.0_Final.pdf
22. *Requirements Interchange Format (RIF)*, <http://www.automotive-his.de/rif/doku.php>
23. Rodrigues, O., Garcez, A., Russo, A.: *Reasoning about Requirements Evolution using Clustered Belief Revision*. In: Bazzan, A.L.C., Labidi, S. (eds.) *SBIA 2004*. LNCS (LNAI), vol. 3171. Springer, Heidelberg (2004)
24. *SPARQL Query Language for RDF*. W3C (January 2008), <http://www.w3.org/TR/rdf-sparql-query/>

25. Supakkul, S., Chung, L.: A UML Profile for Goal-Oriented and Use Case-Driven Representation of NFRs and FRs. In: SERA 2005 (2005)
26. van Lamswerdee, A., Darimont, R., Letier, E.: Managing Conflicts in Goal-Driven Requirements Engineering. *IEEE Transactions on Software Engineering* 24(11) (November 1998)
27. van Lamswerdee, A.: Goal-Oriented Requirements Engineering: A Roundtrip from Research to Practice. In: Invited Minitutorial, Proceedings RE 2001 - 5th International Symposium Requirements Engineering, Toronto, pp. 249–263 (2001)
28. Vicente-Chicote, C., Moros, B., Toval, A.: REMM-Studio: an Integrated Model-Driven Environment for Requirements Specification, Validation and Formatting. *Journal of Object Technology, Special Issue TOOLS Europe 2007* 6(9), 437–454 (2007)
29. Zowghi, D., Offen, R.: A Logical Framework for Modeling and Reasoning about the Evolution of Requirements. In: RE 1997, Annapolis, USA (January 1997)

Model-Driven Security in Practice: An Industrial Experience*

Manuel Clavel^{1,2}, Viviane da Silva², Christiano Braga^{2,**}, and Marina Egea²

¹ IMDEA Software Institute, Madrid

² Facultad de Informática, Universidad Complutense, Madrid

manuel.clavel@imdea.org, {viviane, cbraga, marina_egea}@fdi.ucm.es

Abstract. In this paper we report on our experience on using the so-called model-driven security approach in an MDA industrial project. In model-driven security, “designers specify system models along with their security requirements and use tools to automatically generate system architectures from the models.” Our report includes a discussion of the languages that we used to model both the functional and the security system’s requirements, as well as a description of the transformation function that we developed to build from the security-design models the system’s access control infrastructure. The report concludes with the lessons about the feasibility and practical industrial relevance of the model-driven security approach that we learned from this experience.

1 Introduction

Model-Driven Architecture (MDA) [4] holds the promise of reducing system development time while improving the quality of the resulting products. It is argued that the construction of models during requirements analysis and system design will improve the quality of the resulting systems by providing a foundation for early analysis and fault detection. Moreover, the models constructed in the analysis and design phases will serve as specifications for the later development phases and, when they are sufficiently formal, they will also provide the basis for refinement down to code through well-defined model transformation functions.

Model-Driven Security (MDS) [2] is a recently proposed specialization of the MDA approach. Here, “designers specify system models along with their security requirements and use tools to automatically generate system architectures from the models, including complete, configured access control infrastructures.” It is argued that this approach “bridges the gap between security analysis and the

* Research partially supported by Spanish MEC projects TIN2005-09207-C03-03 and TIN2006-15660-C02-01, and by Comunidad de Madrid Program S-0505/TIC/0407. In addition, Christiano Braga’s and Viviane Silva’s research is supported, respectively, by the “Ramón y Cajal” and “Juan de la Cierva” Spanish MEC postdoctoral programmes, and Marina Egea’s research by a Spanish MEC predoctoral grant.

** On leave from Universidade Federal Fluminense, Brazil.

integration of access control mechanism into end systems. Moreover, it integrates security models with system design models and thus yields a new kind of model, *security design models*.¹

This paper reports on our experience on using MDS in a mid-size MDA industrial project. The project was funded by a major Information Technology company¹ with a two-fold goal: on the one hand, it was aimed towards enhancing the test report configuration facility currently provided by a general-purpose, automatic test system, which is developed and commercialized by the company; on the other hand, it was conceived as a pilot project for assessing the benefits of MDA when applied to concrete in-house software development projects.

For this project we have used the ComponentUML language (a simplified version of the UML [6] class diagram's language) to model the functional requirements of the test report configuration utility, and the SecureUML language [2] (an extended version of Role-Based Access Control [3]) to model the requirements regarding the configuration utility access control policy. In fact, we have combined the SecureUML and ComponentUML languages to produce the security-design model of the test report configuration utility, which integrates in a single model our system and security models. Moreover, to precise the above models, we have extensively used the OCL language [5], first, to specify the invariants on the system's classes (including their methods' pre- and post-conditions), and, second, to formulate the authorization constraints associated with the system's access permissions. Finally, we have followed a step-by-step methodology to build, from our security-design model, the complete access control infrastructure for the test report configuration utility.

The enhanced test report configuration utility was delivered to our client on time. In our opinion, the success of this project provides further evidences of the potential benefits of applying MDS: mainly, it gives rise to security-aware models that are technology independent, reusable, and evolvable; and, it allows to build security-aware applications that are consistent with the security-aware models. Despite this potential, the current lack of appropriate tool support will hinder its applicability in large-size industrial projects.

Organization. The paper is organized as follows. First, Section 2 summarizes the project requirements and Section 3 provides background material on the MDS languages that we used in this project. Then, Sections 4 and 5 explain our MDS approach for specifying the functional and the access control requirements as security-design models, while Section 6 describes our MDS transformation function to build security-aware applications from security-design models. Finally, Section 7 summarizes the lesson that we have learned in this project.

¹ The company is ranked among the three first European companies in its sector according to stock market capitalization, and it is one of the three Spanish companies with more investment in R&D. In 2007, revenues will exceed €2.150M, of which a third comes from the international market. The company employs more than 23.000 professionals and has clients in more than 80 countries.

Table 1. Some requirements for the test report configuration utility

Clause#1	A repository of TRCs shall exist in the system.
Clause#6	The scope of a TRC can be one of the following: Global: the TRC is accessible for reading to everybody; Private: the TRC is accessible (for reading and writing) only to its owner.
Clause#7	Every TRC has one and only one owner. The user that creates a TRC is its owner.
Clause#8.1	Any user with Test Supervisor and Test Administrator privileges is allowed to create a TRC.
Clause#8.4	Every TRC shall be univocally identified, both to the user and to the system. The identification shall include the owner and the name, so that one user cannot define two TRCs with the same name.
Clause#8.5	Users in the Test_Supervisor or Test_Administrator groups can create TRCs with either Global or Private scope. At the moment of creation, the user shall establish the scope of the TRC. Users not in the above groups can only create TRCs with Private scope.
Clause#9	A TRC with Global scope shall be accessible to all users with read permission. A TRC with Private scope shall only be accessible to its owner and any user with Supervisor and Administrator privileges with read/write permissions.
Clause#11.1	Only users with write access on the TRC are allowed to delete that TRC.

2 The Project Requirements

At the start of the project, we were provided with a two-page document explaining the overall features of the existing test report configuration utility and the enhancements that were expected from this project. These enhancements were defined in a five-page document listing fifty clauses written in English by the Chief Software Engineer of the test system development team. The enhanced test report configuration utility should allow the users to select (and possibly modify) the configurations (called TRCs) to be used for reporting the results of their tests. To this effect, the users will choose from a pool of available test report configurations, which may include private, global, and default ones, the latter associated with individual test programs or with families of test programs. The permissions to create, edit, delete, or apply report configurations will depend both on the user's role and on the actual properties (including the ownership) of the configurations. In Table 1 we show the clauses used in this paper to illustrate the application of MDS in this project.

3 The SecureUML+ComponentUML Language

This section provides background material on the modeling language that we used for applying MDS in this project, namely, SecureUML and ComponentUML. This material is borrowed from [2,1].

SecureUML. This is the security modeling language that we chose in our project. It is a language for formalizing access control requirements that is based on Role-Based Access Control (RBAC) [3]. In RBAC, permissions specify which roles are allowed to perform given operations. These roles typically represent job functions within an organization. Users are granted permissions by being assigned to the appropriate roles, based on their competencies and responsibilities in the organization. RBAC additionally allows one to organize the roles in a hierarchy, where roles can inherit permissions along the hierarchy. In this way, the security policy can be described in terms of the hierarchical structure of an organization.

SecureUML extends RBAC with *authorization constraints* to specify policies that depend on dynamic properties of the system state. Thus, it formalizes access control decisions of two kinds:

1. Declarative access control decisions which are based on static information, namely the assignments of users and permissions to roles.
2. Programmatic access control decisions which are based on dynamic information, namely the satisfaction of authorization constraints in the current system state.

In practical terms, SecureUML provides a language for modeling *Roles*, *Permissions*, *Actions*, *Resources*, and *Authorization Constraints*, along with their *Assignments*, i.e., it allows for the specification of which permissions are assigned to which roles, which actions are assigned to which permissions, which resources are assigned to which actions, and which constraints are assigned to which permissions. In addition, actions can be either *Atomic* or *Composite*. The atomic actions are intended to map directly onto actual operations of the modeled system. The composite actions are used to hierarchically group more lower-level ones and are used to specify permissions for sets of actions. SecureUML leaves open what the protected resources are and which actions they offer to clients. These are specified in a so-called *dialect* and depend on the primitives for constructing models in the system design modeling language of choice.

ComponentUML. This is the system design modeling language that we chose for our project. It is a simple language for modeling component-based systems. Essentially, it provides a subset of UML class models: *Entities* can be related by *Associations* and may have *Attributes* and *Methods*.

SecureUML+ComponentUML. This is the SecureUML dialect that provides the language for expressing SecureUML access control policies over ComponentUML resources. As a SecureUML dialect, its metamodel [2,1] specifies:

1. The *protected resources*. In SecureUML+ComponentUML, the protected resources are the entities depicted in the ComponentUML model, as well as their attributes, methods, and association-ends (but not the associations as such).
2. The *actions* that the protected resources offer and their *hierarchies*. The actions considered in SecureUML+ComponentUML are shown in the following table, where underlined actions are composite actions.

Resource	Actions
Entity	create, <u>read</u> , <u>update</u> , delete, <u>full access</u>
Attribute	read, update, <u>full access</u>
Method	execute
AssociationEnd	read, update, <u>full access</u>

3. The *default access control policy* for those actions where no explicit permissions are defined in the models (i.e., whether access is allowed or denied by default). In SecureUML+ComponentUML, the access is granted by default.

4 Modeling the Functional Requirements

Following the MDS approach, we first built a design model of the the test report configuration utility, based exclusively on the functional requirements included in the requirements document. For this task we used the modeling language ComponentUML. To add precision to our models, we imposed invariants on the classes, and pre- and post-conditions on the methods using the constraint language OCL.

To illustrate our approach we show in Figure 1 the ComponentUML model corresponding to the functional requirements listed in Table 1. First, we modeled the static part of the configuration utility with a class *TRC* that has the attributes *owner*, *name*, and *scope*, and the methods *create*, *delete*, and *read*. Then, we specified the (first part of the) Clause#8.4, namely,

“Every TRC shall be univocally identified, both to the user and to the system. This identification should include the owner and the name [...]”

with the following OCL class invariant:

context TRC **inv** uniqueIdentifier:

TRC.allInstances->forAll(trc |
trc<>self implies (trc.owner<>self.owner or trc.name<>self.name))

which restricts the valid instances of our model to those where each TRC has a unique identifier, namely, the name of its owner followed by its own name. Next, we specified the (second part of) Clause#8.4, namely,

“[The identification shall include the owner and the name,] so that one user cannot define two TRCs with the same name.”

with the following OCL pre-condition and post-condition:

context TRC::create(p_scope:TypeOfScope,p_owner:String,p_name:String)

pre: TRC.allInstances->forAll(trc|trc.owner<>p_owner or trc.name<>p_name)

post: self.scope = p_scope and self.owner=p_owner and self.name=p_name

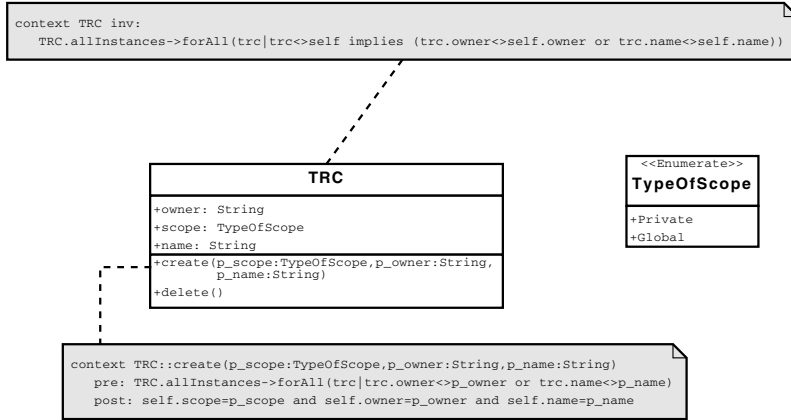


Fig. 1. Modeling the functional requirements

which guarantees that the method *create* preserves the invariant *uniqueIdentifier*. Notice, however, that the above post-condition only partially specified the (second part of) Clause#7, namely,

“The user that creates a TRC is its owner.”

The problem is that to fully specify this restriction we must also specify that the method *create* is always called with the name of the user who intends to create a TRC as its second argument. However, since a ComponentUML diagram only models the static part of the system, the actual *caller* of the method *create* can not be modeled here (and, consequently, can not be referred to either). In the next section, we will see how this restriction is specified in the SecureUML+ComponentUML diagram.

5 Modeling the Access Control Policy

Next, following the MDS approach, we constructed a security-design model of the test report configuration utility by modeling the access control policy defined in the requirements document *on top of* the model describing the static part of the system. For this task we used the modeling language SecureUML+ComponentUML, which requires the use of OCL to express the authorization constraints restricting the access permissions.²

To illustrate our approach, we show in Figure 2 the SecureUML+ComponentUML model corresponding to the access control policy for the creation of TRCs as defined in Table 1. We specified the user groups with three roles (hierarchically

² In an OCL authorization constraint, the variable ‘caller’ refers to the the user who intends to access the resource, while the variable ‘self’ refers to the resource that is being accessed. Here we assume that users have a *name*.

organized): *Test_Operator*, *Test_Supervisor* and *Test_Administrator*. Then, we specified the Clause#8.5 and the (second part of) Clause#7, namely,

“Users in the *Test_Supervisor* or *Test_Administrator* groups can create TRCs with either *Global* or *Private* scope. At the moment of creation, the user shall establish the scope of the TRC. Users not in the above groups can only create TRCs with *Private* scope.”

“The user that creates a TRC is its owner.”

with the permissions *NewPrivate* and *NewGlobal*. These permissions are assigned, respectively, to the *Test_Operator* and *Test_Supervisor* roles. First, notice that the permissions *NewPrivate* and *NewGlobal* are both constraint by the OCL expression

`p_owner = caller.name`

which authorizes the execution of the method *create* only when it is called with the name of the user who intends to create a TRC as its second argument. With this authorization constraint, together with the postcondition of *create*, we effectively guarantee that “The user that creates a TRC is its owner.”

Also, notice that the permission *NewPrivate* is additionally constraint by the OCL expression

`p_scope = Private.`

With this authorization constraint, together with the postcondition of *create*, we guarantee that users not in the *Test_Supervisor* or *Test_Administrator* groups “can only create TRCs with *Private* scope.”

Finally, notice that, since *Test_Supervisor* is a sub-role of *Test_Operator*, users in the *Test_Supervisor* group inherit the permission *NewPrivate*. Similarly, since *Test_Administrator* is a sub-role of *Test_Supervisor*, users in the *Test_Administrator* group inherit the permissions *NewPrivate* and *NewGlobal*. Therefore, we guarantee that “users in the *Test_Supervisor* or *Test_Administrator* groups can create TRCs with either *Global* or *Private* scope.”

6 Building the Security-Aware Application

As part of our application of the MDS approach, we defined a transformation function \mathcal{F} from SecureUML+ComponentUML models to C++ code.³ As it is well-known, transformation functions from models to models and, ultimately, from models to code, are a key component of MDA. In our project, the transformation function \mathcal{F} was indeed crucial not only to implement the access control policy which was defined in the original requirements document, but also to

³ In [2] the authors defined two different transformation functions to translate security-design models to either EJB or .NET technology.

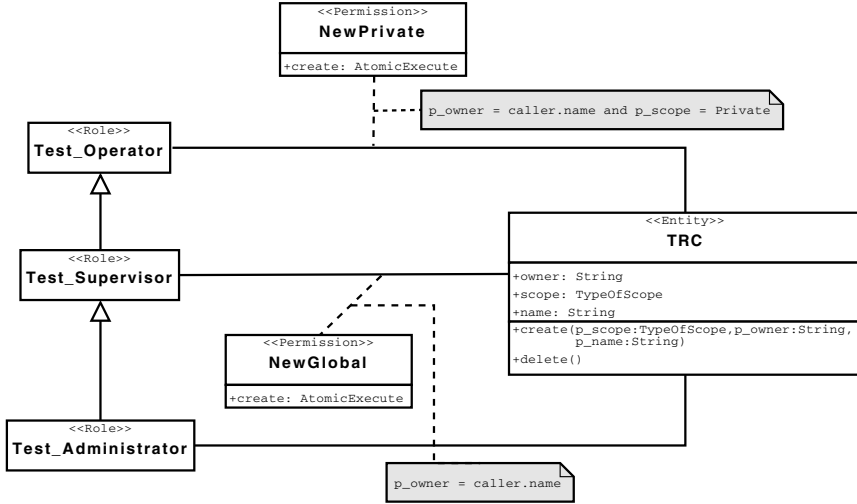


Fig. 2. Modeling the access control requirements

modify this implementation when the client introduced refinements or changes in the original document, as happened on several occasions throughout the project.⁴

As expected, the transformation function \mathcal{F} takes into consideration (and, consequently, is limited by) the technology used by our client to implement access control policies, which includes, in particular:

- A specific XML document, named **UserRights**, in which one can define the relationship between the user groups and the so-called “topics”, as well as the hierarchy among the user groups. Topics can have different meanings: we will use them to denote permissions.
- A specific method, named `isServiceGranted(topic)`, with which one can test whether the logged-in user belongs to (either directly or indirectly) the group associated with *topic* in the **UserRights** XML document; this method is provided by a component `IUserAdmGet`.

In particular, the transformation function \mathcal{F} provides the following step-by-step methodology to implement in C++ the access control policy modeled in a diagram. Although we have not implemented this transformation function (due both to time constraints and the experimental nature of the project, which as mentioned before was conceived by the company as a pilot project), we did manually follow the steps defined here when implementing the enhanced test report configuration utility’s access control policy.

Step 1. Define as groups in the **UserRights** document the roles depicted in the diagram, as well as the hierarchy among them. For example, the following

⁴ Interestingly, these refinements or changes were mostly prompted by some unintended implications of the clauses contained in the original document, which were easily exposed after analyzing the corresponding security-design model.

XML code is part of the *UserRights* document that we have used to translate the model depicted in Figure 2.⁵

```
<VIRTUAL_USER_GROUPS>
  <VIRTUAL_USER_GROUP name="L1">
    <USER_GROUP name="Test_Administrator"/>
  </VIRTUAL_USER_GROUP>
  <VIRTUAL_USER_GROUP name="L2">
    <USER_GROUP name="Test_Supervisor"/>
    <VIRTUAL_USER_GROUP_REF name="L1"/>
  </VIRTUAL_USER_GROUP>
  <VIRTUAL_USER_GROUP name="L3">
    <USER_GROUP name="Test_Operator"/>
    <VIRTUAL_USER_GROUP_REF name="L2"/>
  </VIRTUAL_USER_GROUP>
</VIRTUAL_USER_GROUPS>
```

Step 2. Define as topics in the *UserRights* document the permissions depicted in the diagram, associating with each topic the group linked to the corresponding permission. For example, the following XML code is part of the *UserRights* document that we have used to translate the model depicted in Figure 2.⁶

```
<TOPIC name="NewPrivate">
  <RIGHTS>
    <VIRTUAL_USER_GROUP_REF name="L3"/>
  </RIGHTS>
</TOPIC>
<TOPIC name="NewGlobal">
  <RIGHTS>
    <VIRTUAL_USER_GROUP_REF name="L2"/>
  </RIGHTS>
</TOPIC>
```

To explain the next step, we introduce first some notation. For each *method* in the security-design diagram, let $PRM(method)$ be the set of permissions that grant (conditional or unconditional) access to execute *method*. Also, for each *permission* in the diagram, let $CTR(permission)$ be the OCL expression that constrains *permission* (or simply ‘true’ if no authorization constraint is associated

⁵ In a *UserRights* XML document, the tag `VIRTUAL_USER_GROUP` is used to declare a group: its attribute `name` provides a reference to it. The name of the group is declared using the tag `USER_GROUP`. A hierarchical relationship between two groups is declared using the tag `VIRTUAL_USER_GROUP_REF`: its attribute `name` refers to the name of the group “above” in the hierarchy. The `VIRTUAL_USER_GROUP` declarations must occur inside the tag `VIRTUAL_USER_GROUPS`.

⁶ In a *UserRights* XML document, the tag `TOPIC` is used to declare a topic. A topic is related with a given group by means of the tag `VIRTUAL_USER_GROUP_REF`: its attribute `name` refers to the name of the group. This association must be declared within a `RIGHTS` tag.

with *permission*). In addition, for each *method* in the diagram, let $PRE(method)$ be the OCL expressions that (pre-)conditions the execution of *method* (or simply ‘true’ if *method* does not have a precondition). Finally, for any OCL Boolean *expression*, let $C++(expression)$ denote the C++ Boolean function that implements *expression*.

Step 3. Make the execution of each *method* in the diagram to depend on the satisfaction of its precondition and of its access control policy. The former is implemented by the function $C++(PRE(method))$, while the latter is coded by $CHK(method)$, which is a Boolean function (typically with the same parameters as *method*) that returns **true** if and only if there exists a *permission* in $PRM(method)$ such that both $isServiceGranted(permission)$ and $C++(CTR(permission))$ return **true**.

Notice that a naive implementation of $CHK(method)$ can be easily generated for each *method* in a diagram as follows. Let $PRM(method) = \{p_i\}_{1 \leq i \leq n}$. Then, $CHK(method)$ can be implemented by the following C++ code, where the variable `m_uag` references the object of type `IUserAdmGet` that encapsulates the information about the logged-in user.

```
if (m_uag->isServiceGranted(p1) && C++(CTR(p1)))
{
    return true;
}
...
if (m_uag->isServiceGranted(pn) && C++(CTR(pn)))
{
    return true;
}
return false
```

Of course, more efficient implementation of $CHK(method)$ can be generated if we take into consideration the role hierarchy and/or the logical implications among the authorization constraints (we omit here the details). For example, the following C++ code is part of the function $CHK(create)$ that we have used to translate the model depicted in Figure 2:

```
m_uag->GetUser(user_name);
if (m_uag->isServiceGranted('NewPrivate'))
{
    if (p_scope = Private && p_owner = user_name)
    {
        return true;
    }
    else
    {
        if (m_uag->isServiceGranted('NewGlobal')
            && p_owner = user_name)
```

```

        {
            return true;
        }
    }
    return false

```

Notice that, since every user in the *Test_Supervisor* (or *Test_Administrator*) group is also a user in the *Test_Operator* group, our (efficient version of the) function *CHK(create)* directly returns **false** if the logged-in user does not belong to the group associated with the topic “NewPrivate” (namely, *Test_Operator*).

7 The Lessons Learned

We conclude this report with a brief summary of the lessons we have learned with respect to both the MDA approach and the MDS approach. First, with respect to the MDA approach:

- *The construction of models during requirement analysis and system design provides a foundation for early analysis and fault detection.* The analysis of the models constructed from the original document prompted us to refine those requirements that were ambiguous, to eliminate those that were duplicated or implied by others, and to fill those that were simply missing. As an example, we found out that the requirements neither specify whether a TRC with Global scope can be deleted (and by whom), nor do they unambiguously specify if the user that modifies a TRC can become its owner
- *The models constructed in the analysis and design phases provide a basis for refinement down to code.* The generation of the C++ code following our transformation function \mathcal{F} helped us to correctly implement (and modify) the test report configuration utility’s access control policy. As an example, in several occasions throughout the project, the client refined (or modified) the system’s access control policy in order to fix the inconsistencies and ambiguities that were detected in the original document; to adapt our code to the new requirements, we simply modified the models and applied to them our transformation function \mathcal{F} .

Now, with respect to the MDS approach:

- *The security design models integrate security models with system design models, remaining at the same time technology independent, reusable, and evolvable.* First, the system design and the security models helped us to understand (and discuss) the original requirements document by allowing us to independently model each clause based on its principal concern (whether functional or security-related). Then, the security design models helped us to integrate the resulting functional and security models, without having to commit ourselves to a specific technology.

- *The security design models are understandable by those familiar with the UML-notation.* The security design models became in fact the *lingua franca* used in the discussions between the requirements and software engineers (from the company) and the software developers (from our group).

Finally, SecureUML+ComponentUML (with OCL) was expressive enough to model the access control policy defined in the original requirements document provided by our client. However, access control policies are not for sure the only security requirements found in industrial specifications. In the near future, we plan to apply the MDS approach to other security requirements, which may require the use (or even the definition) of other modeling languages and tools. We also plan to design these projects so as to be able to gather more quantitative results about the benefits of applying MDS in system development.

Acknowledgments. We thank our student Felipe Padilha for helping us with the C++ implementation of the enhanced test report configuration utility. We also thank our contacts in the company for their availability (and patience!) to answer our questions about the original requirements document. Finally, we thank David Basin, Jordi Cabot, Robert Clariso, and the anonymous referees for their helpful comments on previous versions of this paper.

References

1. Basin, D., Clavel, M., Doser, J., Egea, M.: A metamodel-based approach for analyzing security-design models. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735, pp. 420–435. Springer, Heidelberg (2007)
2. Basin, D.A., Doser, J., Lodderstedt, T.: Model driven security: From UML models to access control infrastructures. *ACM Transactions on Software Engineering and Methodology* 15(1), 39–91 (2006)
3. Ferraiolo, D.F., Sandhu, R.S., Gavrila, S., Kuhn, D.R., Chandramouli, R.: Proposed NIST standard for Role-Based Access Control. *ACM Transactions on Information and System Security* 4(3), 224–274 (2001)
4. Kleppe, A., Bast, W., Warmer, J.B., Watson, A.: *MDA Explained: The Model Driven Architecture—Practice and Promise*. Addison-Wesley, Reading (2003)
5. Object Management Group. Object Constraint Language specification (2004), <http://www.omg.org>
6. Object Management Group. Unified Modeling Language specification (2004), <http://www.uml.org>

Supporting the UML *State Machine Diagrams* at Runtime

Franck Barbier

PauWare Research Group - Nefective Technology
BP 1155, 64013 Pau CEDEX, France
Franck.Barbier@FranckBarbier.com

Abstract. Input models that are not completely checked generate ill-formed output models in MDA transformation processes. Model executability is a means for, at development time, simulating/testing models and thus making them compliant with requirements. At runtime, persistent models bring added values like the monitoring and control of applications through the observation of the active states, the guards which hold true, the occurring events... This paper on purpose presents a Java-based execution engine for the UML *State Machine Diagrams*. In order to incorporate this UML interpreter into MDA tools, the execution semantics of the UML *State Machine Diagrams* is first analyzed and next disambiguated. Execution semantics choices are thus proposed and justified accordingly.

Keywords: UML, Statecharts, model executability.

1 Introduction

This paper discusses the constituents of a Java library and its associated API [1] for executing the UML *State Machine Diagrams* [2, pp. 520-582]. It is acknowledged that Harel's Statecharts [3] with their numerous object-oriented (or not) variants, *e.g.*, [4], [5], [6], and the UML *State Machine Diagrams* do not offer the same execution semantics. Many theoretical contributions exist, *e.g.*, [7], [8], [9], but few explain how a well-defined execution semantics may be implemented in a CASE tool. To that extent, the Java framework presented in this paper is currently incorporated into the Topcased MDA platform [10] which itself lays out on the top of the Eclipse Modeling Framework (EMF).

The execution semantics of UML is not really formal (it has some ambiguities). It is also open through the notion of "semantic variation point" [11]. In fact, the nature of encountered problems is conceptual rather than technical in the sense that one must first of all disambiguate the execution semantics of UML.

Models based on the formalism of the UML *State Machine Diagrams* are written in XMI (XML Metadata Interchange). This XMI code is transformed into Java code which itself relies on the proposed Java library/API. The generated code is a trustworthy representation of the upstream models so that this code may serve for simulations/tests (development time) or for building end user applications (runtime). So, in simulation/test phases, the observation of unexpected behaviors allows us to correct models in relation with requirements. This supposes that the built models are fully deterministic. To create this determinism, this paper discusses how the UML

State Machine Diagrams are accurately and uniformly interpreted. Most of the time, the interpretation rules are textually described in the UML documentation [2, pp. 520-582], but some of these rules require further investigations.

In order to elaborate on this topic, Section 2 provides some key design principles associated with the use of the UML *State Machine Diagrams* execution support presented in this paper. Section 3 discussed what is, in our opinion, the key problem of a UML *State Machine Diagrams* interpreter: the management of conflicting transitions when one wants to move a state machine from one stable consistent context to another. Section 4 is about the notion of allowed event which has a different denomination in UML. We show the lack of specification of this notion in UML. Finally, we conclude in the fifth section.

2 Design Principles

The key design principles promoted by the Java framework presented in this paper are as follows: from a UML state machine diagram (Fig. 1), one derives the micro-architecture of a software component (*My component* in Fig. 2) in which its provided interface (*A provided interface* in Fig. 2) and its implementation (see the list of operations within *My component* in Fig. 2) are clearly separated. In short, events on transitions become services of the component having the state machine diagram as behavioral specification. In contrast, states, actions and event self-sending are parts of the component’s inner workings.

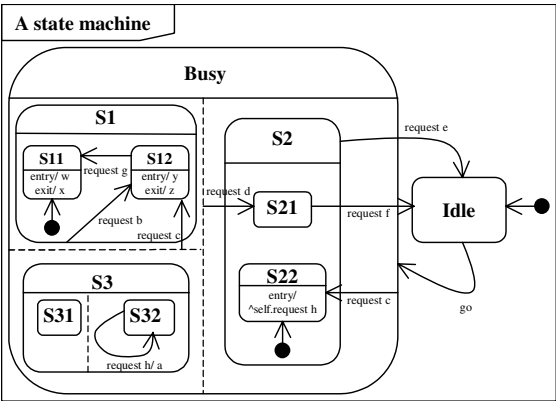


Fig. 1. Behavioral specification of a software component

2.1 Event Processing Based on Run-to-completion Steps

At runtime, instances of *My component* process events according an execution semantics. This amounts to obey to the execution semantics assigned to the UML *State Machine Diagrams* plus some additional hypotheses below described.

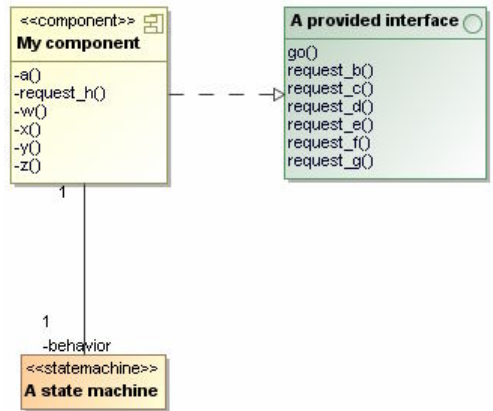


Fig. 2. Architectural organization of *My component* which owns the model in **Fig. 1** (*A state machine*) as behavioral specification (*request h* is not part of *My component*'s provided interface since it is only sent internally, i.e., *entry/self.^request h*, when entering into *S22*).

A run-to-completion cycle is in essence bound to the processing of a single event occurrence. This event is typed by its name, or better by its signature (if any) in a state machine diagram. Event occurrences are not shared by state machines. Moreover, as many occurrences as needed of a given event type must be sent, if several components have to be informed of (and thus have to process) the event at a given time. For example, two instances of the *My component* type may exist at a given time and thus two state machines (**Fig. 3**) as well. If present, an occurrence of the *go* event is assigned to one and only one state machine among the two in **Fig. 3**. In other words, a given event occurrence cannot generate two effects. This means that this occurrence is consumed once and for all by one and only one state machine.

This consumption model enables the distribution of state machines, especially when they are encapsulated in distributable software components, like *Enterprise JavaBeans* (EJBs) for instance [12], which may run on distinct deployment nodes. This rule is also relevant for mobile systems (Java ME platform) in which components, and thus state machines, are embedded in wireless devices [13]. Sharing

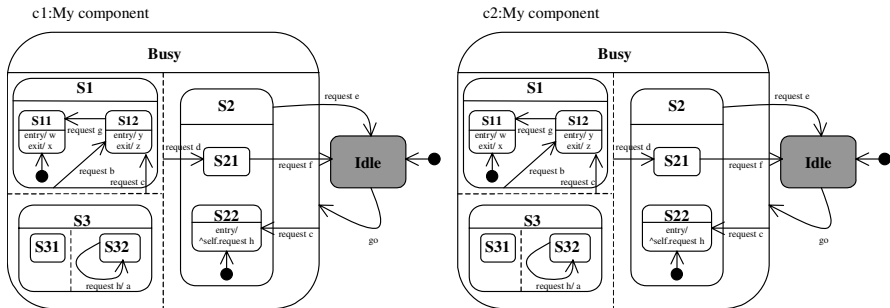


Fig. 3. Two instances of *My component* (*c1* and *c2*) with both the *Idle* state active

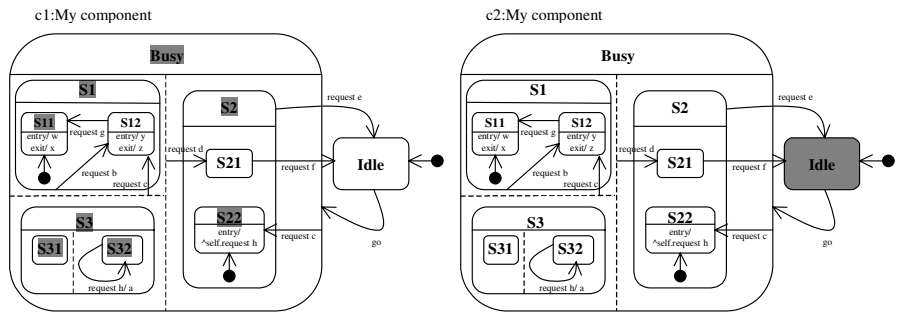


Fig. 4. Two instances of *My component* (*c1* and *c2*) after the processing of only one occurrence of the *go* event

events is conceptually appealing for theoretical computing paradigms but unlikely in today’s applications which tend to avoid centralization.

As a result, in **Fig. 4**, a single occurrence of the *go* event has moved the state machine on the left hand side of **Fig. 3** to a new set of active states, while that on the right hand side has not changed.

The execution semantics is therefore based on the principle that event occurrences are partly defined/identified through the unambiguous identity of their unique expected receiver (here, the component which has the state machine on the left hand side of **Fig. 4**). The principle relating to the fact that senders and receivers have identities and that events carry these identities, relies on the notion of component connector in UML [1, p. 177]: “Specifies a link that enables communication between two or more instances. This link may be an instance of an association, or it may represent the possibility of the instances being able to communicate because their identities are known by virtue of being passed in as parameters (...)” A detailed accurate discussion on this topic also appears in [14].

2.2 Lost of Events, Event Consumption Principle

The notion of “deferred events” in UML [1, p. 550] is a semantic variation point [11]. More precisely, events are picked up in state machines’ queues as soon as a run-to-completion steps terminate. In this line of reasoning, let us make the following observation about the *c1* component instance on the left hand side of **Fig. 4**: its *Idle* state is **not** grayed meaning that it is **not** active. Let us now suppose that, at this moment, the *go* event occurs and is dispatched to *c1*. In this context, the model in **Fig. 1** tells us that this occurrence is discarded or “lost” (*deferred events* = *false*, the default UML mode). With *deferred events* = *true*, the state machine’s interpreter has to find another interpretable event occurrence in its queue: the *go* occurrence is not lost and is intended to be later processed. No accurate rigorous policy is however provided about this issue in the UML documentation. In the Java framework presented in this paper, we only use the *deferred events* = *false* mode because of its immediate compliance with UML.

3 Transitions Conflicts

In the UML doc. [1, p. 562], it is written: “In other words, in case of conflicting transitions, only one of them will fire in a single run-to-completion step.” This rule first supposes the availability of a definition about what are conflicting transitions: “Two transitions are said to conflict if they both exit the same state, or, more precisely, that the intersection of the set of states they exit is non-empty. Only transitions that occur in mutually orthogonal regions may be fired simultaneously.” In this section and in Section 4, we discuss why this characterization is inadequate.

3.1 Transition Overriding

In the beginning of a run-to-completion step, the presence of the current event occurrence to be processed, the evaluation of the active states and the evaluation of the guards attached this event occurrence, amount to computing a set of eligible transitions. If this set is empty, the event occurrence is discarded (see above). If this set has more than one enabled transition, the run-to-completion engine has to establish all mutual conflicts, if any. Finally, this engine must develop a policy based on choosing between some transitions to be triggered and the rest to be put aside.

According to UML, in **Fig. 5**, if *Substate* is active (and thus *Superstate* is also active) and an occurrence of *r* exists, then only the *g* action is executed. In other words, the two depicted transitions are eligible, but they are in conflict. As written above, “(...) the intersection of the set of states they exit is non-empty.”: {*Superstate*, *Substate*}. So, *f* is not launched even if *Superstate* is also active and *r* occurs because the transition labeled *r/g* hides *r/f*. This specific case is ruled by an implicit priority policy in UML: “By definition, a transition originating from a substate has higher priority than a conflicting transition originating from any of its containing states.” [1, p. 562]. This rule is considered in [8] and [9] has a key difference between the classical Harel’s Statecharts and UML.

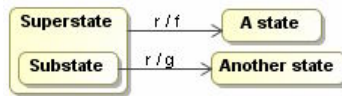


Fig. 5. Potential transition conflict

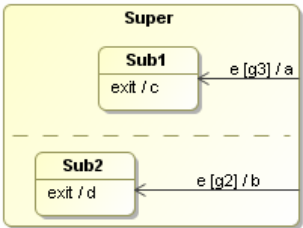


Fig. 6. Potential transition conflict versus no conflict at all

However, the UML criterion which allows an execution engine to determine if two or more transitions are mutually in conflict is, in our opinion, limited in scope. For example, the model in **Fig. 6** is not a source of conflict.

In **Fig. 6**, the single significant difference with **Fig. 5** is the fact that the two “conflicting” transitions are from *Super* to *Sub1* and from *Super* to *Sub2* **with *Super* being the superstate** (whether direct or indirect is of no importance) **of *Sub1* and *Sub2***. Applying the UML execution semantics leads to considering the model in **Fig. 6** as a source of conflict, if g_2 and g_3 are true when e occurs. Indeed, the set of states the two “conflicting” transitions exit is non-empty and equal to $\{Super\}$ ¹. However, the model pattern in **Fig. 6** is common in Statecharts-based modeling. For example, the a and b actions may correspond to the refreshing of distinct display elements within a MVC interaction. In this case, g_2 and g_3 may determine if the data to be displayed has changed. We thus do not apply the UML precepts for situations like that in **Fig. 6**.

3.2 Transitions with Orthogonal States as Destinations

The state machine diagram which formalizes the behavior of another software component in **Fig. 7** shows, once again, a situation in which the g_2 and g_3 guards are a potential source of conflict. If $S1$ is active and e occurs while g_2 and g_3 are true, the two terminal states ($S2$ and $S3$) of the two enabled transitions can be potentially attained due to the fact that they are orthogonal.

The execution semantics of UML is clear about this issue: this case is a “true” conflict. Nevertheless, the conflict does not rely on the two target states which are compatible (*i.e.*, parallel). In UML, the rule to comply with is that one transition at the most must be triggered among the two that are leaving $S1$ in **Fig. 7**. So, if g_2 and g_3 are true at the same time, triggering only one transition among the two conflicting ones is based on expressed priorities. This is an option in UML, but noting priorities in diagrams is not yet instrumented (by stereotypes or any native modeling construct). Thus, in the most common case (*i.e.*, no priority is setup), one might interpret the execution semantics of UML as follows: the state machine remains is $S1$. This

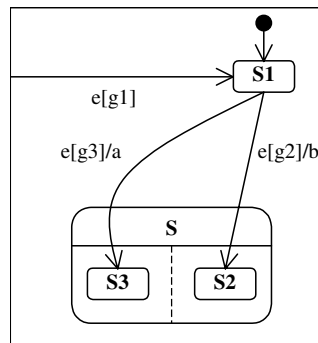


Fig. 7. The behavior of a second software component

¹ The notation in **Fig. 6** is equivalent to having two self-transitions connected with *Sub1* and *Sub2*. This notation is useful when one wants to factorize specification elements.

solution is unsatisfactory because an exception must also be thrown to make the conflict visible. Alternatively, one may consider the model in **Fig. 7** as closely related to that in **Fig. 6** in the sense that the two reachable states are orthogonal. To have a uniform approach, we reject the UML semantics and view the situation in **Fig. 7** as non-conflicting. So, both transitions are triggered and thus the *a* and *b* actions are run.

3.3 Transitions with Nested States as Destinations

In **Fig. 8**, there is a slight variant of the state machine diagram of **Fig. 7**. In **Fig. 8**, the target states of the transitions *S1* \rightarrow *S2* and *S1* \rightarrow *S3* are nested. Like orthogonal states, they are always active at the same time. So, if *S1* is active and *e* occurs while *g2* and *g3* are true, UML views these two transitions as incompatible.

What the models in **Fig. 6** and **Fig. 7** have in common is that the two (orthogonal) attainable states never constitute the source of the conflict. The same analysis can be done for the model in **Fig. 8**: they do not cause the conflict. However, the model in **Fig. 8** can be, with some additional elements, a source of tricky problems. For instance, non-decidable situations may arise if there is another default input state (see *S4* in **Fig. 9**). In this case, state machines truly become non-deterministic at runtime, if *g2* and *g3* are both true at the same time when *e* occurs.

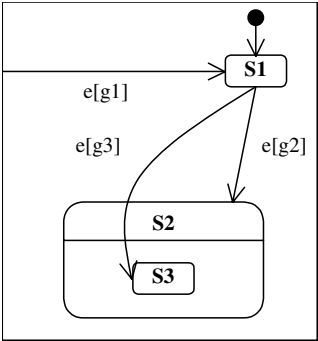


Fig. 8. The behavior of a third software component

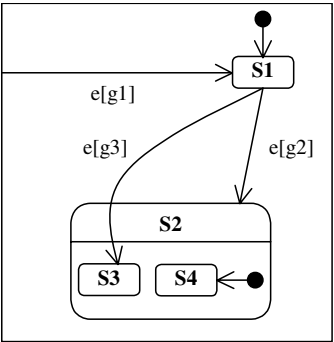


Fig. 9. Variant of the behavior of the software component in Fig. 8

To sum up, in **Fig. 8**, the two target states ($S2$ and $S3$) of the two conflicting transitions are “compatible”. In contrast, in **Fig. 9**, $S2$ and $S3$ are “incompatible”: reaching $S2$ means reaching $S4$ which has a exclusiveness relationship with $S3$.

To keep a uniform approach, we apply the UML semantics to the two models in **Fig. 8** and in **Fig. 9**. So, if $g2$ and $g3$ are both true at the same time when e occurs, the execution of the models in **Fig. 8** and in **Fig. 9** leads to raising an exception². Since we do not have a method for arbitrarily choosing between the two conflicting transitions, $S1$ remains active. The key issue is the fact that the model in **Fig. 8** is not managed like the models **Fig. 6** and **Fig. 7**. We arrive at the following amended definition of a conflicting transition: two eligible transitions are in conflict if and only if they target states are not orthogonal.

4 Allowed Events

An allowed event is an event which does not trigger any transition. This concept has been formally defined in [14]. An allowed event is associated with a given state and, by definition, bypasses the entry and exit actions of this state. In UML, such a concept is named “internal action”³ since an allowed event in a state (many allowed events may however appear) is connected with an action to be executed when this event occurs and the said state is active (**Fig. 10**, see e in the inside of $S1$ or $S11$). Otherwise, without associated action, an allowed event is useless.

Having a formalism dedicated to allowed events precludes for creating fictitious states and unintelligible transitions that do not correspond to the business logic. In other words, allowed events have to be processed, without impact on the course of a state machine: there is no state change at all when an allowed event is processed.

However, allowed events may be the source of conflicting transitions (**Fig. 10**): “An internal transition in a state conflicts only with transitions that cause an exit from that state.” [1, p. 562]. Indeed, for a given state, an event can be declared both as “allowed” and as a label of an outgoing transition. Moreover, this phenomenon may be generalized for many states, which are ($S1$ and $S11$ in **Fig. 10**) or are not nested.

Conflict problems in **Fig. 10** come from the possible values of the $g1$, $g2$, $g3$ and $g4$ guards. First, one may notice that the e event is an allowed event for both $S1$ and

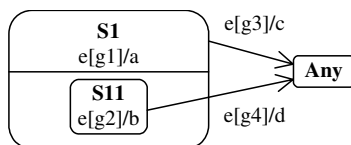


Fig. 10. Conflicts linked to the use of allowed events

² Stopping a state machine after an exception or going on is a decision which is external to the interpreter.

³ The “internal transition” expression is also used for characterizing the effect of allowed events even though no transition actually occurs. One may also notice that an “internal transition” is not equivalent to a self-transition which, in contrast, does not bypass entry and exit actions.

S11. We consider that there is no conflict if $g1$ and $g2$ are true while $g3$ and $g4$ are false when e occurs. Because *S1* and *S11* are in essence active at the same time, one executes a and b . Moreover, *S1* and *S11* remain active. There is no predefined order when executing a and b . The modeler must be aware that the $a;b$ sequence or $b;a$ can be performed.

An alternative semantics is executing b while a is not executed (same conditions: $g1$ and $g2$ are true while $g3$ and $g4$ are false when e occurs). This semantics relies on what follows: “(...) nested states override enclosing states.” [1, p. 552]. To assess the relevance of this semantics, the following text extract may also be studied: “For example, consider the case of two transitions originating from the same state, triggered by the same event, but with different guards. If that event occurs and both guard conditions are true, then only one transition will fire.” [1, p. 562]. In **Fig. 10**, the two “internal transitions” have no true “origin state” since they have no extremities at all. So, this alternative semantics (b is executed while a is not) is rejected. This choice is inspired by the fact that *S1* and *S11* are “compatible”: *S1* and *S11* are always, by definition, both active or both inactive at the same time. To sum up, our semantics enables three kinds of execution: a ($g1 \wedge \neg g2$), b ($\neg g1 \wedge g2$) or, a and b ($g1 \wedge g2$). The alternative semantics only allows us to execute a ($g1 \wedge \neg g2$) or b ($g2$). It thus seems poorer in terms of expressiveness.

An extended conflict case is when e occurs, $g1$ and $g2$ are true, and $g3$ is true or $g4$ is true or both. So, do we enter into *Any*? What is executed? From a model checking viewpoint, an appropriate solution is to demonstrate, by means of a symbolic evaluation, some definitive logical dependencies between $g1$, $g2$, $g3$ and $g4$. For example, a dependency like $g1 = \neg g3$ definitely exclude runtime conflicts about, leaving or not *S1* and thus, executing c or a . Unfortunately, such a symbolic computation is not always possible. A conflict management policy is thus required at runtime. Besides, even if the behavior assigned to substates contractually overrides the behavior assigned to superstates, one also needs to establish a priority between “internal” transitions and “external” transitions. In our framework, for a given state, “internal transitions” have higher priorities than “external transitions”. Again, no strategy is provided by UML about this issue. Returning to the example in **Fig. 10**, one may accordingly observe at runtime the following execution semantics:

- $g2 \rightarrow b$ (no situation change: *S1* and *S11* active)
- $\neg g2 \wedge g4 \rightarrow d$ (*Any* active)

The first clause shows that when $g2$ is true, the execution of d is bypassed ($g4$ being true or not). We have a dual situation for the $(g1, g3)$ pair. This clause illustrates the highest priority of “internal transitions” compared to “external transitions”. The second clause illustrates the crossing of concerns between internal/external transitions on one side and, enclosing/nested states on the other side. If $g4$ is true ($g1$ being true or not), a is bypassed. This means that an external transition of a nested state has higher priority than an internal transition of an enclosing state. We adopt this priority type to be closer as possible to UML (*i.e.*, “(...) nested states override enclosing states.”).

In any case, we think that we obtain a robust modeling framework. Even if the choice for internal/external transitions is debatable, it can be easily managed through a stereotype, say «*InternalVersusExternal*», which is associated with two tagged

values. This stereotype is either an extension of the UML *State* metatype or *StateMachine*. It depends upon the degree of desired precision.

5 Conclusion

There are many papers which discuss execution semantics variants for the UML *State Machine Diagrams*, but few discuss implementations in a tool and explain how a chosen execution semantics has to be controlled by this tool. Moreover, we show in this paper that arbitrations and tradeoffs are numerous and important if we want to have a rigorous interpreter of the UML *State Machine Diagrams*. We especially show that this objective does not always favor a compliance with UML.

The code which aims at simulating models can also be used at runtime: such a code is model-centric and acts as an appropriate support for component observability and controllability. It greatly benefits from the upstream models it is derived from. Consequences are manageability, including the real-time visualization of state machine changes. Furthermore, state machines may act as appropriate supports for having self-configuring and self-healing components: the idea of autonomic computing. Self-healing extends the idea of self-configuring as follows: the availability of facilities for carrying out analyses and diagnoses of sequential run-to-completion cycles, may favor the identification/construction of correction strategies. As it happens, corrections may be defined as applying undo/rollback actions to cancel the effect of triggered transitions associated with caught exceptions. All of these challenges and research orientations are relevant perspectives to enhance the Java framework presented in this paper.

References

1. PauWare software and users' guide, http://www.PauWare.com/PauWare_software
2. Object Management Group: Unified Modeling Language: Superstructure, version 2.1.1, formal/2007-02-03 (February 2007)
3. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8, 231–274 (1987)
4. Harel, D., Naamad, A.: The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology* 5(4), 293–333 (1996)
5. Harel, D., Gery, E.: Executable Object Modeling with Statecharts. *IEEE Computer* 30(7), 31–42 (1997)
6. State Chart XML, <http://www.w3.org/TR/2007/WD-scxml-20070221/>
7. Mellor, S., Balcer, S.: Executable UML - A Foundation for Model-Driven Architecture. Addison-Wesley, Reading (2002)
8. von der Beeck, M.: A structured operational semantics for UML-statecharts. *Software and Systems Modeling* 1(2), 130–141 (2002)
9. Crane, M., Dingel, J.: UML vs. classical vs. Rhapsody statecharts: not all models are created equal. *Software and Systems Modeling* 6(4), 415–435 (2007)
10. Topcased CASE environment, <http://www.topcased.org>

11. France, R., Ghosh, S., Dinh-Trong, T., Solberg, A.: Model-Driven Development Using UML 2.0: Promises and Pitfalls. *IEEE Computer* 39(2), 59–66 (2006)
12. Barbier, F.: An Enhanced Composition Model for Conversational Enterprise JavaBeans. In: Gorton, I., Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Szyperski, C.A., Wallnau, K. (eds.) *CBSE 2006*. LNCS, vol. 4063, pp. 344–351. Springer, Heidelberg (2006)
13. Romeo, F., Barbier, F., Buel, J.-M.: Observability and Controllability of Wireless Software Components. In: Indulska, J., Raymond, K. (eds.) *DAIS 2007*. LNCS, vol. 4531, pp. 48–61. Springer, Heidelberg (2007)
14. Cook, S., Daniels, J.: *Designing Object Systems - Object-Oriented Modelling with Syntropy*. Prentice-Hall, Englewood Cliffs (1994)

Model-Based Generation of Interlocking Controller Software from Control Tables

C. Chevillat¹, D. Carrington², P. Strooper², J. G. Süß², and L. Wildman²

¹ Sofismo AG, Sgestrasse 50, CH - 5600 Lenzburg
cec@sofismo.ch

² The University of Queensland, Australia
Information Technology and Electrical Engineering (ITEE)
(davec,pstroop,jgsuess,luke)@itee.uq.edu.au

Abstract. Railroad interlocking software drives specialised micro-devices, known as interlocking controllers. These controllers primarily actuate railroad points and change signal aspects in real-time, based on sensor and timer input. Due to their central function in railroad control, interlocking controllers and their firmware are safety-critical. The firmware programs, which mimic physical relays, are written in variants of domain-specific programming languages based on ladder logic. The programs have to comply with a more abstract specification of allowable states of sections of railroad track and equipment, known as a control table. The translation of a track layout and associated control tables into ladder logic-based code is manual, and hence subject to costly review and rework cycles. In this report, we describe a case study that uses a model-driven tool-chain as an automated alternative to the existing process. The two domain languages, control table and ladder logic, were modelled and transformations were implemented between the two models, and from model to program text. We report on implementation challenges, and describe the outlook and scalability of the approach in this application domain.

1 Introduction

Railway interlocking software drives specialised micro-devices, known as interlocking controllers. These controllers primarily actuate railroad points and change signal aspects in real-time, based on sensor and timer input. Currently, interlocking software is written by railroad engineers. The programs have to comply with a more abstract specification of railroad tracks and equipment, known as track layout and control tables. Interlocking software is written in variants of domain-specific programming languages based on ladder logic. The translation of a track layout and associated control tables into ladder logic is manual, and hence subject to costly review and rework cycles. It is guided by policy documents specific to a railway company or to an area of a company's network. The resulting variability makes it challenging to automate this process.

Since the MDA is particularly suitable for generating domain-specific software [2] and the ladder logic programming language is fairly simple, it seems

sensible to apply the Model Driven Architecture (MDA) for generating railway interlocking software. In this case, the Platform Independent Model consists of a model of the track layout and control tables, and the Platform Specific Model corresponds to the ladder logic code.

In this paper, we present a proof-of-concept of this idea. The work was a collaboration between The University of Queensland and Ansaldo STS, whose support for this work we would like to acknowledge. Ansaldo develops railway interlocking software for their Microlok II logic controller for many railway companies. Longer-term, it is hoped that the MDA technology will support the development of railway interlocking software for different domains, but in this initial proof-of-concept, we applied the technology to a single interlocking from a single railway company. In particular, we developed meta-models for the track layout and control tables, and for the Microlok II Control Logic Language (MLC), as well as transformations between them. These meta-models and transformations, as well as additional pre- and post-processing steps, have been integrated in the LokGen prototype tool. It has been applied to two small, but real railway interlockings. We describe the prototype tool and the results of the case study, including several implementation challenges that we faced, which provides an indication of the current state of tool maturity in MDA. We also describe the outlook and scalability of the approach in this application domain.

The remainder of this paper is organised as follows. Related work is reviewed in Section 2. We provide the necessary background to the relevant railway signalling terminology and documents that we use in Section 3. In Section 4, we present an overview of the approach. Section 5 presents the results of the case study. The outcomes of the project and case study are discussed in Section 6. Section 7 discusses future plans and concludes the paper.

2 Related Work

The safety-critical nature of railway interlocking systems has made them a popular target for research in software development and verification. One of the goals of such research is to ensure that the controlling software correctly implements the safety requirements so that trains do not collide or derail. The requirements of railway interlocking systems are complicated since they involve multiple pieces of equipment (signals, tracks, points, etc.) and assumptions about accepted behaviour of trains. As a consequence, abstract models are normally developed to capture the requirements precisely. The value of such models is generally accepted [3]: “Through modelling, we have found many things that were inconsistent, over-specified (ruling out of states which logically never arise) or had simply been forgotten and thus even if the UML model were not to be used outside of our office, the improvements it led to would remain”.

Many different modelling notations have been used [4,5,6,7], with formal notations normally being more amenable to rigorous analysis and verification. UML as a general-purpose modelling language has also been applied to the railway interlocking domain [3,8,9], even though its semantics are not fully formalised

and its analysis support tools are less powerful than more specialised modelling notations. Rastocny et al. [8] believe that the “general availability and understandability of the standard UML is an undoubted advantage, together with minimum cost needed to learn it and permanent support of the UML standard from the main providers of software tools”.

A critical challenge with using a model to perform software verification is ensuring that the implemented software corresponds to the verified model. This is why the Model-Driven Architecture (MDA) approach is important, since it promotes models as primary development artifacts and generates the corresponding executable software via model transformation techniques. Naturally, the transformations also need to be verified, but the MDA approach reduces the likelihood of inconsistencies between the models and the executed software, especially as systems evolve over time. Majzik et al. [10] use a subset of UML state machines to avoid problems with semantic ambiguities and describe a tool chain that enables them to simulate the state machine behaviour to check the completeness and consistency of the specification, and to generate the control software and test cases. Rastocny et al. [8] model a railway interlocking system in UML and use the I-Logix Rhapsody tool to animate the model and automatically generate code (“at least 90% of the generic code of the interlocking logic”).

3 Background

Controlling railroad lines is safety-critical, safety requirements are made more explicit than in many other areas of software engineering. For example, the equipment that interacts with trains and track, called ‘field equipment’ in domain terminology, is classed as vital or non-vital. The procedures followed by railroad engineers ensure that any failure of vital equipment will maximally cause operations to cease, but never endanger human lives in the process. Also, railway signalling uses a rigid but safe approach to collision avoidance. Every train is assigned a route, which is a configuration of tracks starting at a signal and ending at another signal, that provides an exclusive path with no other simultaneous user. Once a route is locked in, a train can traverse it without danger of a collision. In its simplest term, railway signalling involves the safe establishment and release of routes for trains. Routes will also be inspected for safe travel conditions. For example, configuring two consecutive points in such a way that a train has to change direction twice may be safe for a low-speed train, but dangerous for a high-speed train. As a result, the set of routes is smaller than the set of potential paths through the network. Designing routes is not an easy task and it requires that railroad engineers foresee conditions of contention.

The control technology for field equipment has evolved from a box of mechanical levers, to electrical circuits and relays, which are true parallel evaluation systems, to single-processor microcontrollers like the MLK2, that simulate parallel behaviour. As a result, the ladder logic programming language of the MLK2 is modelled on the older relay technology. The translation of the requirements of the control table into the logic program of the microcontroller has so far been

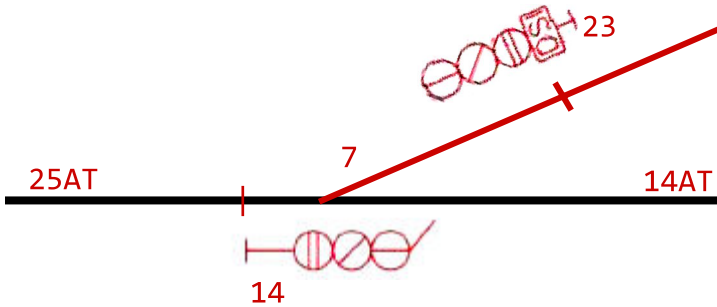


Fig. 1. Example of a track layout showing signals 14 and 23, tracks 14 and 25, and point 7

carried out by specialists and each change in the control table requires a re-translation to code and subsequent inspection of the solution. In this project the translation step was mechanised to make it more repeatable and reliable.

Two types of documents are used to produce MLC programs for a specific part of a railroad network. The first type consists of technical documents that describe the particular section of the railway network that the MLC program is intended to control. Primarily these are the Track Layout and the Control Tables. In addition, the electrical signalling connection of the microcontroller that runs the resulting program is specified in an Input/Output Bit List. The second type consists of documents that define implementation policies specific to a company or an area of a company's network. In this case they are called the Data Structure Document (DSD) and the Signalling Principles. Policy documents lay out best practices including quality and safety requirements. The following sections describe these documents in more detail.

The Track Layout diagram shows the physical arrangement of field equipment. It also defines blocks of track that serve as interfaces to adjacent areas of control, usually neighbouring stations. Figure 1 is an excerpt of a track layout.

A *Control Table* is a tabular representation of all vital dependencies within the signalling system for control and release of signals, points, and other equipment. Control tables are constructed by a railroad engineer applying signalling principles to the track layout. They define the prerequisites that have to be satisfied before an action can be taken. For example, turning a signal to green will require that following block is free and all other tracks leading into it are sealed off with signals. The design of control tables varies between railroads. Table 1 is an example of a control table for signals.

Bit Lists declare Boolean variables that map to input/output registers in the external communication interface of the controller. They define the interface to the local field equipment and to the Train Control System (TCS), which provides operational control of the whole network.

Table 1. Signal control table

SIGNAL ROUTE Id.	ASPECT	REQUIRES							
		SIGNAL NORMAL	TRACK CIRCUITS CLEAR		POINTS SET & LOCKED N : R	ROUTE LOCKING			
			STICK TRACK	OTHER TRACKS		TRACKS	OR TIME RELEASED Secs	AFTER SIGNAL USED	
13	Y	35 42 (41 w 6R)	13AT	13BT (20AT (54BT w 20 cl) w 8N 6R &w 7N 6N)	8				
	G								
20A	Y w JI	42 36 (27 w 8R 6R) (44 w 8R 6N 7R &w 8R 6R 7N)	20AT	24AT (27CT w 10R)		6	28AT 28BT	CLEAR	28A 43A
	G w JI								
20B	Y		20AT	19AT w 7R	6		27AT	180	44A
	G								

A *DSD* of a railroad project is a best-practice catalogue of patterns for programming in Microlok II Control Logic Language (MLC). It evolves during the project. It is produced by rail engineers that interpret the provisions of the ‘Signalling Principles’ document and then write quality MLC programs based on the track layout, control tables, and bit lists. Whenever a typical pattern occurs, the input from these three sources and the corresponding output code is listed, leading to a mapping template. When a pattern reoccurs, the same mapping template is applied. As new situations are encountered, the pattern catalogue grows. As a result, a DSD grows into an informal and partial description of a mapping or transformation.

Signalling Principles define design principles for a specific railway company and/or area. They cover the operational requirements that need to be considered when producing the signalling arrangements, control tables, interlocking design, and programs. For example, a high-speed rail may have minimal separation distances between signals, or maximal approach speeds. These affect switching times required for points and signals, and track speeds. Signalling Principles thus affect the strategies used to implement the application logic. Consequently, a DSD is derived from the signalling principles applied during its creation.

4 MDA Approach

This section discusses the underlying technology and workflow of our solution and describes interesting aspects of its major components: the control-table and code metamodels, and the model transformation.

4.1 Technology

The project used the Eclipse Modelling Framework (EMF) as its basis for implementation. EMF consists of libraries and code generators that convert a meta-model into a storage layer, a data manipulation layer that provides undo/redo,

and a user interface component that is compatible with the Eclipse IDE. This high degree of automation and integration enabled us to concentrate on the modelling side much more than would have been possible with previously available Meta Object Facility (MOF) approaches like the Java Metadata Interface. However, EMF does not yet provide model transformation tools as part of the basic framework. There is still a great deal of competition in this area despite the fact that the OMG has already published the QVT standard. In this project the model transformation was implemented in Java. Transformations from model to text were implemented using Java Emitter Templates (JET) technology, which is used by EMF itself to generate source code. JET combines text templates with data drawn from a storage facility. JET is crucial to the functioning of EMF, hence it is mature, so we can confidently apply it in the project.

4.2 Workflow

The MLC Program Generation Software (LokGen) generates MLC following the workflow in Figure 2. It shows the MOF levels M2, which holds the model of the domain language, and M1, which holds instances of those languages. The model transformation is an interpretation of the DSD and formalizes how an instance of the Control Tables Extended Metamodel (CTExtended) is mapped to an instance of the Microlok II Program Metamodel (MLKCode). The transformation takes one control table model as its input and returns one microlok II code model as its output. Internally, it uses EMF OCL to query the source model.

EMF uses a mechanism called 'resource factories' to load and store models in different external formats. MLC Program Generation Software (LokGen) contains two resource factories for loading the control table: either it is read from a ZIP archive containing five tables stored as Comma Separated Values (CSV) files or one Excel file containing the same tables as worksheets. The layout of the tables follows the layout of the work items that rail engineers are used to. The control tables follow the layout used in the CAD drawings of the stations map and the bit lists follow the layout used in specification documents. The code output is generated using JET.

4.3 Metamodels

The degree of detail and semantic strength of the two metamodels are different: The Extended Control Table is modelled as expressively as possible, while the code model is very close to the syntactical presentation of the output text.¹

The CTExtended model is segmented into four packages. Three of these packages reflect the domain entities described earlier: track layout including field equipment, control tables and bit lists. The fourth is a system container. This modularity simplifies adaptation and maintenance of the models.

¹ A detailed discussion of the metamodels is available from <http://www.itee.uq.edu.au/~mdavv/>.

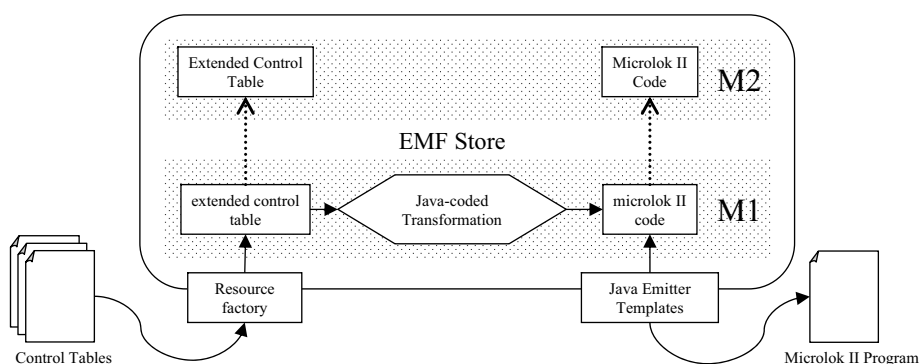


Fig. 2. MLC Program generation workflow

Part of the control table model is a model of a textual grammar. The customer-specific variant of control tables implemented in this project uses a simple notation for conjunctive formulas to express required conditions on pieces of field equipment and tracks, as shown in Table 1. For example, the statement below specifies that Track 13B must be unoccupied, and Track 20A must be empty **either** (&) when point 8 is Normal and point 6 is Reverse, or when points 7 and 6 are both Normal. In addition to the point conditions that apply to Track 20A, Track 54B is required to be unoccupied when signal 20 is clear.

13BT (20AT (54BT w 20c1) w 8N 6R &w 7N 6N)

Similar to the strategy applied to OCL2 in the UML2 standardisation, the constraints are parsed and turned into model elements to make them accessible to model-based tools. In contrast to the OCL approach, our parser retains the syntactical order and form of terms. It does not canonize the formula or optimize redundancies. This was done deliberately to retain formatting based on syntactical conventions that may implicitly be used by the domain engineers.

All models use derived attributes to describe secondary properties using domain terminology. Internally, these derived attributes are realised as OCL queries and can hence be expressed in a more compact and explicit way than would be possible with a Java implementation. The model does not make use of meta-model operations or the EMF validation framework because most conditions are currently checked by the resource factory during table load.

The code model is a close rendition of the grammar and style used for MLK2 programs. As it follows the style laid down in the DSD, it cannot be used to reverse-engineer every syntactically correct MLK2 program. Rather it is intended as a blueprint for well-formatted MLK2 programs that adhere to the DSD.

4.4 Transformation

The model transformation implements rules of the DSD that define the rendering of control tables to code by patterns. As a result, these rules are not described strictly in terms of either the source or target model. However, the code examples that are used to illustrate the rules are concrete, so they are

primarily used in structuring the transformation program. The set of rules was expressed as a set of Java methods that have no order dependence. The Java methods recursively decompose the transformation problem, and do not access overlapping sections of the target. These restrictions were adopted to allow a refactoring of the implementation in the future to change to a declarative model transformation approach. In addition, the methods use OCL query expressions to select parts of the source model. These are shorter and more expressive than imperative loops, and hence avoid errors when navigating the model.

5 Evaluation

The tool was evaluated on a real station design, and the output compared with the actual implementation that had been verified by railroad engineers. The implementation was designed and evaluated with inputs encoded for two station designs. The first design was used as sample data for testing the implementation during development. The second design was used only after the implementation of the tool was completed, to separate implementation and validation. The station designs are small, but representative of typical stations in local railroads. The corresponding track layouts contain three points, six main signals, two shunt signals, a siding line and two block sections. In both cases, the design contains 5 condition lines in the points control table and 11 condition lines in the signals control table. The following sections list some interesting aspects of the results of the case study.

The process generated syntactically correct MLC programs for the samples used in the case study. The MLC compiler accepted the code in all cases. This is interesting, as the available documentation did not include the compiler's documentation or a BNF. It shows that building a code generation facility from semantically well-defined samples can effectively create a sound sub-language.

As the transformation system is built as an implementation of the guidelines of the DSD, all generated programs comply with the DSD's provisions. Code produced by LokGen offers improved readability and orientation through consistent variable ordering and naming, based on the order of occurrence in the control tables. In addition, indentation and section titles make it easy to navigate the produced code. Assignments are explicit and fine-grained, using only basic constructs of the Microlok II (MLK) language, which aids debugging in the simulator.

While the project initially assumed that the information to generate the code was completely provided within the tables being processed, it became apparent that the CAD drawing of the station layout must also be considered as input for the transformation. For example, the tables do not contain information about routes that use the same piece of track in opposite directions, which leads to an additional requirement of setting blocking signals for the opposing direction. Similar issues arise if a track is subdivided into smaller consecutive units.

Listing 1.1. Example of the effect of Shorthand notation use

ASSIGN	HS13DPR	TO	HS13D2PR;
ASSIGN	HS13DPR * HS15DPR	TO	HS15D2PR;
ASSIGN	HS13.15DPR	TO	HS13.15D2PR;
ASSIGN	HS13DPR * HS15DPR	TO	HS13D2PR;
ASSIGN	HS13DPR * HS15DPR	TO	HS15D2PR;

Although LokGen produces MLC code that, after compilation will ensure safe operation according to the definitions of the DSD, railroad engineers criticised its style. Investigating why revealed that railroad engineers use the code as a *specification document* in its own right and add information to it that is not available in the control table or track layout, and cannot be computed. Listing 1.1 is intended to illustrate such an issue. It shows source code for a **H**ypothetical **S**tation (**HS**). Each line contains a statement that **ASSIGNs** the evaluation result of a logical formula over variables to output variables.

As the railroad engineers use the code as an additional specification document, they are naturally interested in terseness. This leads to the use of syntactic shorthand notations of the MLK language, which often implicitly alter the meaning of the program. For example, the notation $xN.M$ translates into the conjunction of xN and xM if used on the input side, and causes the creation of two assignment lines if used on the output side. In theory, these forms can hence be translated mechanically. Consider the example in Listing 1.1. It reads as follows: Assign the state of the Clear Repeat Relay of track 13 (HS13DPR) to the second Clear Repeat Relay of track 13 (HS13D2PR). Assign the state conjunction of the Clear Repeat Relays of tracks 13 and 15 to the second Clear Repeat Relay of track 15. This is the smallest possible and explicit solution, and it is generated by LokGen. The railroad engineers prefer to use the line below as an abbreviation. It expands to the two explicit lines shown below it, which are not semantically identical, as they introduce an additional unnecessary condition. These effects grow even more severe if another shorthand “.” is used, which acts as a “for all” construct. The differences observed all stem from the fact that the code is seen as a detached artefact, rather than an intermediate product in a chain from station design to executable binary.

6 Discussion

Summarily the case study can be rated as a success, as the partner is interested in pursuing this approach further. The main motivator is the substantial savings in time to produce a base implementation. While the definition of practice by example as used in the Data Structure Document (DSD) yields acceptable results for the largest part of the work, it also reveals that full automation will require a more thorough formalisation of the rules. Such formalisation would implicitly

standardise the procedures of railroad engineers, reducing the degrees of freedom they currently enjoy in making implementation decisions. In addition, the case study has shown that the assumed inputs for the generation are insufficient as they do not include certain topological information from the station layout, such as opposing routes or subdivided blocks of track. These limitations require a railroad engineer in the design loop, and reduces the traceability of outputs to the original designs. To capture the topological information CAD-drawn station layouts would need to follow more stringent conventions to make them parseable. Finally, an MDA process raises issues of version control and build management. Currently, work artefacts are versioned, moved and archived in paper form. In an MDA setting, this could be automated to ensure that inputs and outputs remain traceable.

The case study also showed that while the core frameworks of MDA are reliable and provide time savings to programmers, other more peripheral building blocks for MDA solutions are still maturing. These limitations prompted certain design decisions in the implementation.

Although the OMG has finalized the QVT standard, uptake of the standard has been slow. Implementations are still rare. There is no complete implementation of the QVT standard as of this writing. Further, although most implementations use EMF as a basis, transformation models are not interchangeable between implementations. Originally we had intended to use the Tefkat [11] model transformation system to implement the core model transformation. However, Tefkat describes its transformations in terms of the source model, which did not match the way rules were described in the DSD. More prominently, Tefkat was not as mature as we had expected and also suffered from an underlying bug in the EMF core. As a result, the Java-based solution was implemented, using the restricted implementation pattern described previously.

As the design of the tool started with tests of sample data to be imported into the model, the resource factory which is responsible for parsing the data checks a large number of well-formedness conditions on the imported table data. As a result, this part of the system is comparatively complex, while the control table model remained a structured data container. In the future, responsibility for upholding well-formedness rules should probably be refactored into a separate validation module. In addition, the parser that disassembles the inline condition statements could be represented as a metamodel operation. That this has not been done yet exposes an issue with EMF: Clean specification of behavioural model aspects (invariants, derived attributes, and operations) are complicated because they are implemented in Java by default.

The development of frameworks that link text-editors and underlying models, such as the Textual Editing Framework, is just beginning. Consequently, the development of text-based model editors currently involves substantial programming effort and was not undertaken in this project. There is currently no text editor for MLK code. However, we believe that a semantically rich model would be a useful contribution towards code analysis and refactoring of existing source.

7 Conclusion and Outlook

Ansaldo, our project partner in the development of this case study, has continued our joint work with development of the tool for two other railroad projects. In these projects MLC Program Generation Software (LokGen) is being used in parallel to the individually programmed solutions provided by the railroad engineer. Both extensions require moderate changes to the input model and transformation, while the output stage has remained the same.

Although building a new but similar variant of the tool for different clients and rail-road types still presents some savings in labour, the optimal solution would encompass a generic model. Such a model would canonize the sets of conditions that are expressed in the different forms of control tables. Designs could then be transferred between projects, and engineers would not need to learn different tools. Maintenance and development could be streamlined, with version management, distribution and update provided by the facilities of the Eclipse framework. Combining these factors will maximise reuse and pay-off. To explore this perspective, we are planning a follow-on project with Ansaldo.

Representation of the control tables and other associated input as models opens up the possibility to revisit previous work on verifying these as specifications using model checkers. In previous work, the presentation of generation dependencies and counter examples presented a challenge [12]. The ability of model-based technology to build cross-referenced models should contribute improvements in this area.

While controllers theoretically represent a correct representation of the electrical relay-based mechanism that is simulated in the ladder-logic programming languages, they are in practice restricted by the limits of a classical von-Neumann machine. As such, parallel operations in the language translate into prioritized queues of operations in machine language. In extreme cases, the queues can cause critical real-time limits to be exceeded, or stacks can overflow. As a result, regression testing of the system is necessary. Currently, this is another labour-intensive manual task that offers potential for automation.

The capacity limits of the micro controller may require several collaborating controllers to realise larger station layouts. Ideally, the MDA tool would assign portions of the required program to the different processor nodes and furnish them with operationally safe communication code. Currently, the decomposition of a larger station layout is done manually. Potentially, the MDA approach would allow for safe optimisation of the protocols, or allow automated decomposition and thereby reduce the number of processing nodes required to realise a project.

References

1. Lecomte, T., Servat, T., Pouzancre, G.: Formal methods in safety-critical railway systems. In: Proc. Brazilian Symposium on Formal Methods: SMBF 2007 (2007)
2. Baker, P., Loh, S., Weil, F.: Model-driven engineering in a large industrial context - Motorola case study. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 476–491. Springer, Heidelberg (2005)

3. Bayley, C.: Modelling interlocking systems with UML. In: The IEE Seminar on Railway System Modelling - Not Just for Fun, September 30, 2004, pp. 8–18 (2004)
4. Petersen, J.L.: Automatic verification of railway interlocking systems: a case study. In: FMSP 1998: Proc. the second workshop on Formal methods in software practice, pp. 1–6. ACM, New York (1998)
5. Winter, K., Robinson, N.J.: Modelling large railway interlockings and model checking small ones. In: ACSC 2003: Proc. the 26th Australasian computer science conference, pp. 309–316. Australian Computer Society, Inc. (2003)
6. Borälv, A.: Case study: Formal verification of a computerized railway interlocking. *Formal Asp. Comput.* 10(4), 338–360 (1998)
7. Hartonas-Garmhausen, V., Campos, S.V.A., Cimatti, A., Clarke, E.M., Giunchiglia, F.: Verification of a safety-critical railway interlocking system with real-time constraints. *Sci. Comput. Program.* 36(1), 53–64 (2000)
8. Rástocný, K., Janota, A., Zahradník, J.: The use of UML for development of a railway interlocking system. In: Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E. (eds.) *INT 2004. LNCS*, vol. 3147, pp. 174–198. Springer, Heidelberg (2004)
9. Hon, Y.M., Kollmann, M.: Simulation and verification of UML-based railway interlocking designs. In: *Automatic Verification of Critical Systems*, INRIA, pp. 168–172 (2006)
10. Majzik, I., Micskei, Z., Pintér, G.: Development of model based tools to support the design of railway control applications. In: Saglietti, F., Oster, N. (eds.) *SAFE-COMP 2007. LNCS*, vol. 4680, pp. 430–435. Springer, Heidelberg (2007)
11. Lawley, M., Steel, J.: Practical declarative model transformation with tefkat. In: Bruehl, J.-M. (ed.) *MoDELS 2005. LNCS*, vol. 3844, pp. 139–150. Springer, Heidelberg (2006)
12. van den Berg, L., Strooper, P., Johnston, W.: An automated approach for the interpretation of counter-examples. *Electron. Notes Theor. Comput. Sci.* 174(4), 19–35 (2007)

Model-Driven Simulation of a Maritime Surveillance System

M. Monperrus^{2,3}, F. Jaozafy¹, G. Marchalot¹, J. Champeau²,
B. Hoeltzener², and JM. Jézéquel³

¹Thales Airborne Systems

²ENSIETA/DTN

³INRIA/Triskell

Abstract. This paper reports an industrial experiment made at Thales to use Model Driven Architecture (MDA) for system engineering. System engineering processes are currently mainly document-centric. The main experiment goal was to study the applicability of MDA at the system engineering level. The experiment consisted of setting up a model-driven simulation environment for a maritime surveillance system. The simulation is achieved thanks to 3 models conform to 3 metamodels. The implementation uses the Eclipse Modeling Framework and is written in the Java Programming language. This pilot project met the deadline, the budget and the threshold of desired functionalities. We report the main advances given by the MDA approach in the context of simulation for system engineering.

1 Introduction

Simulation is a key step in the development of a maritime surveillance system [1]. At the very beginning of the life cycle of the system i.e.; at the system engineering level, simulation is a way to communicate with the customer in order to elicit the needs. Therefore, simulation eases the consistency between the customer requirements and the delivered product. From an engineering point of view, simulation is a way to validate parts of the architecture, as well as the behavior of the product. Last but not least, this permits a better cost estimation and a better planning of technical and human resources.

A maritime surveillance system (MSS) is a multi-mission system. As depicted in figure 1 as a UML use case diagram, it is intended to supervise the maritime traffic, to prevent pollution at sea, to control the fishing activities, to control borders. It is usually composed of an aircraft, a set of sensors, a crew and a large number of software artifacts. The number of functionalities, the relationships between hardware and software components and the communication between the system and others entities (base, other MSSs) indicate the complexity of the system.

Maritime surveillance system designers have been developing simulators for several decades. Current simulation methods, sometimes processed at early stage of design through technico-operational simulations, do not fully warranty the

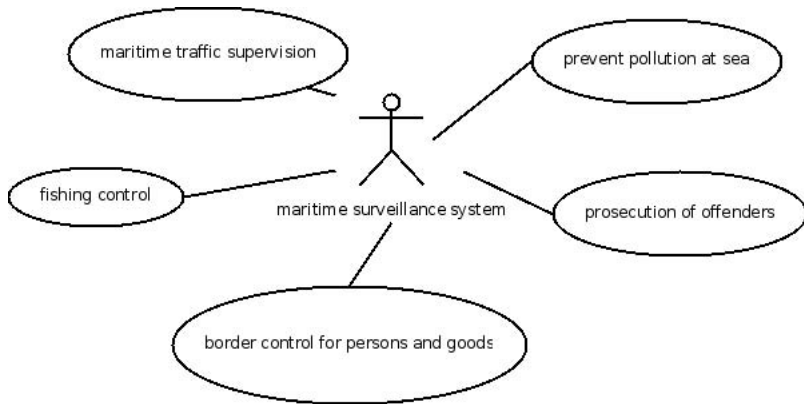


Fig. 1. Use cases of a maritime surveillance system

design quality. Because there is no link between the system model checked by simulation and the system design. Moreover, the techniques used did not meet the required openness for reuse and modifiability. One of the key point of a simulator is to let the end-user, the customer, to test as many versions of the product as needed, and to let the engineers to add easily new features wanted by the customer.

In this paper, we report an industrial experiment we made to develop a new simulator for maritime surveillance systems following a Model Driven Architecture (MDA) approach [2]. The innovation did not reside in 1) new functionalities since the new simulator does not have more functionalities than the legacy ones 2) neither in the modeling activities since simulation has always been based on simulation models. The explored innovation consisted in the use of MDA.

The main goal of this experiment was to study the applicability and the advantages of MDA for system engineering activities. The experiment was a success. We mainly found that the MDA principles enable:

- to keep a complete independence between different concerns that are system architecture, simulation scenarios, and simulation-based analysis.
- to align the system engineering model and the simulation model which can be both explicitly specified.

The paper is organized as follows. In section 2, we present the design of the model-driven simulator. We then present the lessons learned in section 3 and conclude.

2 The Design of the Model-Driven Simulator

Our model-driven simulator was built following the process as follows: 1) identify the domain objects to be modeled; 2) elaborate the corresponding metamodels; 3) design the simulator; 4) develop the simulator.

The remainder of this section does not follow this chronological process. For sake of understandability, we will first present the architecture of the simulator, the metamodels and models used, and we conclude by exploring more in depth how we leverage model orientation in a simulation point of view.

2.1 The Model-Driven Simulation Process

In figure 2 is depicted the architecture of the model-driven simulation process. The backbone of the simulator is the Y pattern depicted as bold lines. The simulator takes as input two models: a model of maritime surveillance system and a model of the tactical situation, called a scenario model. It outputs a model of simulation traces. The three models are specified by three metamodels respectively. The contents of these metamodels are presented in the next sections.

This architecture is built on two model-driven principles: 1) all data are exchanged as models specified by a metamodel 2) the domain concepts are kept independent of implementation concerns. This is slightly different from the Platform Independent Model (PIM) / Platform Model (PM) / Platform Specific Model (PSM) of the seminal MDA paper [2]. However, the architecture follows exactly the same principles. To a certain extent, the MSS model corresponds to a PIM, the scenario model to a PM and the simulation trace model to the PSM.

This architecture has several desirable properties. All the variability of the simulator is encoded into models. For instance, adding a given radar, or changing a feature of the radar to the maritime surveillance system is made by modifying the input MSS model.

The MSS model and the scenario model are not coupled at all. Therefore, we can create several MSS models, several scenario models and study the comportment of each system in each scenario.

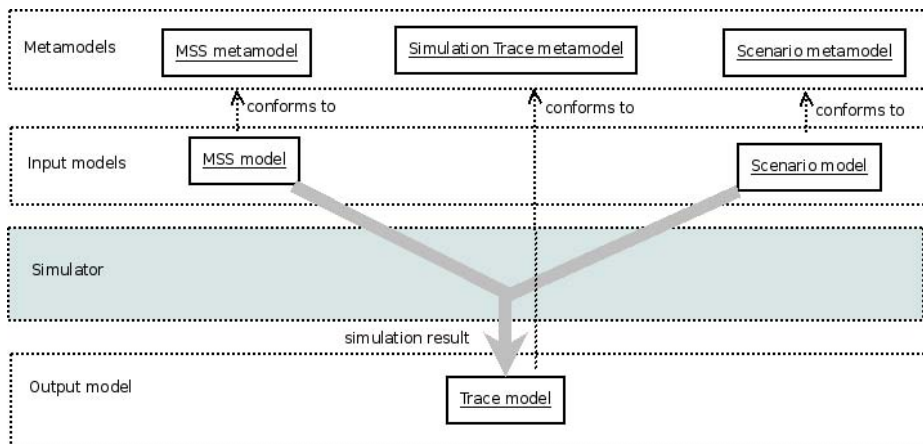


Fig. 2. The MDA architecture of the simulation process

The graphical output of the simulator is absolutely not coupled with the simulation rules. This property enables to easily have many tools that analyze the simulation output. That is to say, we have one and only one interface for plugging modules to the simulator. In such a model-driven architecture, the interface is a model, and is fully, consistently specified by a metamodel, the simulation trace metamodel. This point is further explored in section 2.3.

2.2 The Metamodels and Models Used

Maritime Surveillance System Metamodel. The maritime surveillance system metamodel is divided in five packages. The criterion used to create the packages is a functional decomposition. The navigation system package contains classes whose roles are to represent routing and positioning components (hardware, software and composite). For instance, there is a GPS¹ class. The attributes of this class are the main characteristics of a GPS.

The detection package contains classes representing detection components of a MSS. For instance, the MSS model can be simulated with a classical radar, a IFF (Identification Friend and Foe), a FLIR (Forward Looking Infra Red). The attributes of a model of the radar include the antenna angle, the rotation speed, the impulse period.

The communication system package explicit the communication type and systems that will be simulated with the model. It ranges from internal communication between the crew members and external communications. The classes covers the communication support (e.g. VHF, IHF, satellite) and communication properties (encryption, protocols).

Finally, the crew package contains classes to specify the number of crew members, their respective skills and their functions. The database package acts like a schema of the embedded database of tactical information (e.g.; radar signatures).

The model mainly used for testing purpose represents a maritime surveillance system composed of an airplane Falcon, embedding a radar and an inertial system.

Scenario Metamodel. The scenario metamodels contains all the needed classes to represent a tactical situation. A model, instance of this metamodel, specifies the surveillance zone, the number and types of objects that are in the zone. For each object is specified a trajectory including the speed of the object. Furthermore, at the system engineering level, and for simulation purposes, we had to specify the concept of surface equivalent radar (SER) per object and the concept of weather for a given zone. During the simulation, the weather is used to compute the quality of the information given by the embedded sensors.

Simulation Trace Metamodel. The simulator takes two models as inputs: a MSS model and a scenario model. Then according to the semantics of the simulation, it outputs a simulation trace. Following a model-driven approach, this simulation trace is a model too, specified by a simulation metamodel.

¹ Global Positioning System.

The simulation traces contain all the events that have a semantic with respect to the simulation at the system engineering level. They are :

- the position of the MSS system;
- the position of every objects of the zone;
- the internal attributes of the MSS: fuel, detection field of the radar;
- the semantic events, for instance the detection and identification of the objects achieved by the system core.

In the next section, we show how we use this information to leverage as much as possible of the information given by the simulation.

2.3 Leveraging Model Driven Orientation

The core of a model-driven simulator is to specify all inputs and outputs with metamodels. To a certain extent, each of these metamodels specify an interface. This enables to introduce a kind of composition between the simulator functions. The MDA architecture of the simulator presented in figure 2 illustrates this point.

However the architecture goes far beyond. The simulation part of figure 2 contains only the simulation rules that represent the interactions between the model and the scenario. The architecture enforces that no analysis is done during this phase.

Our idea is to consider that any treatment on the simulation output could be absolutely decoupled of the rest of the application. The treatments are mainly visualization and measurement. However, there are several possible visualization as well as several measurements. We encapsulate each of them into a simulation analysis component, that takes as input a simulation trace model specified by the simulation trace metamodel. This is depicted in figure 3. We apply this design principle to several simulation analysis components.

The first component is a video player of the simulation events contained in the simulation trace model. It outputs the tactical view of the zone of surveillance. A screenshot of this player is given figure 4. The main screen is a 2D representation

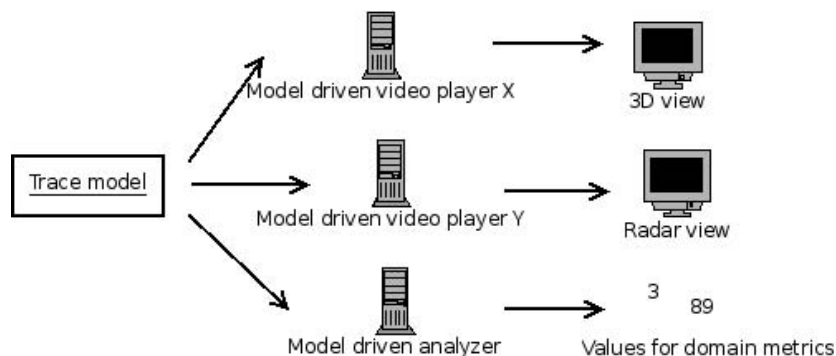


Fig. 3. Usages of the simulation traces

of the zone, with a geographic map, the maritime surveillance system (the yellow dot), the objects of the zone (several boats represented as orange dots), and the detection field of the radar, which is not visible due to the size of the figure. In the upper bar, there are the indicators of the simulation characteristics (time, position and fuel consumption of the MSS). In the left bar is a list of the tactical events that occur, for instance ship detections. The remaining items are widgets to control the simulation (play/pause/stop buttons). The scale that controls the speed of the video player lets to play the scene at a speed different of the simulated time. This simulation component is a demonstrator to communicate with customers as well as to illustrate the impact of a certain system design choice.

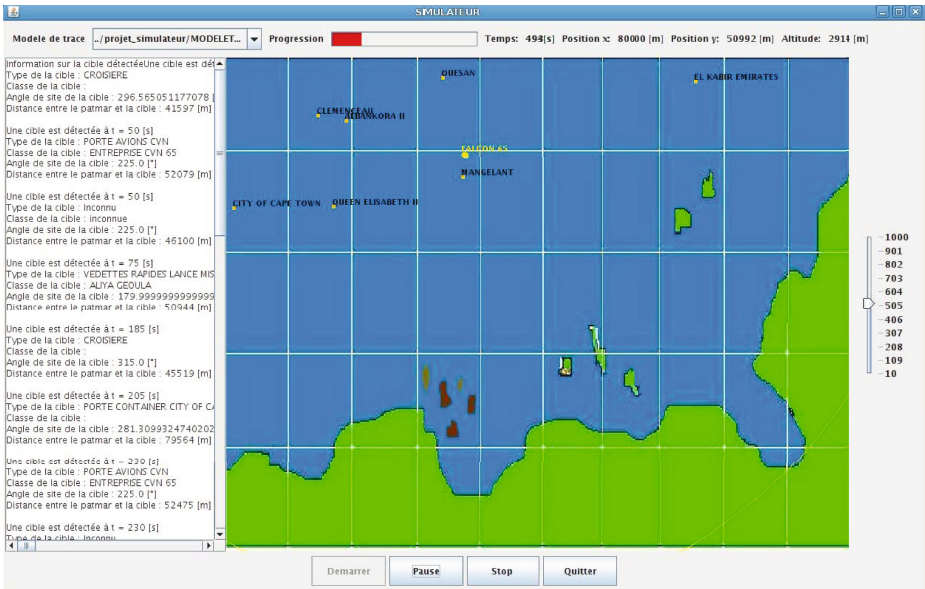


Fig. 4. Screenshot of the model-driven video player

Since we already have human training processes using simulators, we developed an another simulation analysis component to represent the view of a radar operator. The last simulation analysis component computes several domain metric values such as the ratio of detected objects or the ratio of identified objects. This component is generated from an abstract specification of domain metrics, thanks to a dedicated framework developed in another project [3].

Finally, since the simulation analysis components are decoupled from the simulation itself, we plan to reuse these components to analyze some traces of real missions. This will enable to elaborate powerful processes for training and debriefing, as well as analyzing real missions in order to improve future versions of the simulator.

3 Lessons Learned

First of all, this project to apply MDA at the system engineering level was a success. The project met the deadline, the budget and the threshold of desired functionalities. This validates the MDA as an architectural principle for a maritime surveillance system simulator.

Productivity Gain. Our experiment gives some clues about a potential MDA-specific productivity gain. First of all, we obtained the prototype within six months of work of a junior engineer. The technologies used were the Eclipse Modeling Framework [4] coupled to Java to implement the simulator. Thanks to the totally generated EMF model editor, we could right away, and for free, give the system engineers, who are not necessarily computer scientists, a tool to express the models. This is to be compared with the legacy simulators where a specific graphical user interface had to be developed to specify the models involved in the simulation.

An Agility Problem. The main disadvantage of EMF is the code generation step. Code generation is not a problem when one does not have to modify or enrich the generated source code (e.g.; a classical mature compiler). But in the EMF case, some methods and method bodies have to be added into the generated code. Our metamodels were very volatile during the prototyping phase, hence required many code re-generation and the associated method modifications. On the one hand, this hampers the agility of the development process. On the other hand, some re-generations were destructive, i.e; destroyed manually added code, due to the number, the complexity and the sensitivity of the generation parameters. This problem was partially addressed with a source revisioning system.

Separation of Concerns. The MDA architecture of the simulation process enables a real separation of concerns. The variability of a MSS simulator lies in the system architecture, the simulation scenarios, the kinds of post-simulation analysis. All the variability of the models is explicitly concentrated into different metamodels. There is no dependency links between the metamodels hence we obtained a total independence between concerns. This architecture is totally open w.r.t. system models, scenarios and analysis; e.g.; we are able to test different scenarios with a same model or perform different analysis from a same simulation.

Early Model Alignment. The real innovation of our experiment was the meta-modeling of the domain of maritime surveillance systems. Thanks to this step, the creation of a model involves only domain concepts and is not polluted by implementation concerns. This was perceived as highly valuable: the system engineering model facilitates the communication between the different stakeholders (a kind of mental alignment); it can be linked to a design model; and it is totally aligned with the simulation model by construction.

4 Conclusion and Perspectives

In this paper, we presented our experiment to apply Model Driven Architecture at the system engineering level. The experiment consisted in developing a model-driven simulator for maritime surveillance systems. This experiment was a success and raises new issues, which will be explored in further R&D projects.

First of all, the domain metamodel can be seen as the specification of a domain specific modeling language (DSML). We will explore the ways to leverage the metamodel with a dedicated human-machine interface that supports an intuitive concrete syntax for the models. Thus, we would have a intuitive and usable modeling backbone to support the system engineering process.

The core of the simulator is an interpreter of a domain model, achieved by dedicated transformation of two models (MSS and scenario) into a simulation trace model. For prototyping and efficiency reasons, it was implemented in Java. We would like to evaluate the relevance of transformation languages in our case, with respect to the requirements specific to simulation.

The product line of maritime surveillance systems is highly driven by the need for composability. We are exploring how to express this composability within the MDA architecture of the simulator. Studies will be made to explore various solutions, e.g.; expressing the composability directly into the maritime surveillance system metamodel, or building a set of composable models of maritime surveillance systems.

Finally, the great promises of MDA for system engineering would have to be compared to other approaches within a controlled comparative experiment, which was not the goal of ours.

References

1. Ince, A., Topuz, E., Panayirci, E., Isik, C.: Principles of Integrated Maritime Surveillance Systems. Kluwer Academic Publishers (2000)
2. Soley, R.: Model driven architecture. tech. rep., Object Management Group (2000)
3. Monperrus, M., Jézéquel, J.-M., Champeau, J., Hoeltzener, B.: Measuring models. In: Rech, J., Bunse, C. (eds.) Model-Driven Software Development: Integrating Quality Assurance, IDEA Group (2008)
4. Budinsky, F., Steinberg, D., Merks, E., Ellersick, R., Grose, T.J.: Eclipse Modeling Framework. Addison-Wesley, Reading (2004)

Towards Utilizing Model-Driven Engineering of Composite Applications for Business Performance Analysis

Mathias Fritzsche¹, Wasif Gilani¹, Christoph Fritzsche¹,
Ivor Spence², Peter Kilpatrick², and John Brown²

¹ SAP Research CEC Belfast
Shore Road, BT370QB Newtownabbey
United Kingdom

² Queens University Belfast
Queen's Road, BT39DT Belfast
United Kingdom

Abstract. Composite Applications on top of SAPs implementation of SOA (Enterprise SOA) enable the extension of already existing business logic. In this paper we show, based on a case study, how Model-Driven Engineering concepts are applied in the development of such Composite Applications. Our Case Study extends a back-end business process which is required for the specific needs of a demo company selling wine. We use this to describe how the business centric models specifying the modified business behaviour of our case study can be utilized for business performance analysis where most of the actions are performed by humans. In particular, we apply a refined version of Model-Driven Performance Engineering that we proposed in our previous work and motivate which business domain specifics have to be taken into account for business performance analysis. We additionally motivate the need for performance related decision support for domain experts, who generally lack performance related skills. Such a support should offer visual guidance about what should be changed in the design and resource mapping to get improved results with respect to modification constraints and performance objectives, or objectives for time.

Keywords: Model-Driven Engineering, Model-Driven Performance Engineering, Performance Decision Support, Composite Applications, Composition Environment, Enterprise SOA.

1 Introduction

Enterprise SOA offers flexibility for the definition of business processes by the orchestration of business control logic based on Enterprise Services which are technically implemented as Web Services extended with proprietary features [1]. Flexibility in the design of business control logic is further realized by distinguishing between two kinds of process orchestrations: Back-end process

orchestration is done to define business processes with longer lifecycles, whereas front-end orchestration, encapsulated in Composite Applications, is done to compose business processes with shorter lifecycles.

Composite Applications are self-contained applications that combine Enterprise Services with its own business logic, and thereby provide user centric front-end processes. These processes transcend functional boundaries, and are completely independent from the underlying architecture, implementation and software life cycle.

In the development process of Composite Applications, as well as back-end processes, MDE lets domain experts confine their focus to the creative tasks of defining new business processes on top of existing ones by utilizing Domain-Specific Languages (DSLs). The utilization of MDE concepts, furthermore, means that the error prone and time consuming task of actually integrating the new business processes with the complex enterprise scale underlying platform, such as provided by the SAP products Business Suite for large size companies and Business ByDesign for small and mid-size companies, is done in a more efficient way.

A high degree of flexibility along with the fact that domain experts are usually non-performance experts raises the need to develop tools and methodologies to support them in their daily work from the performance perspective.

This paper is structured as followed: Section 2 describes, based on an industrial case study, how Composite Applications are developed by utilizing MDE concepts. In section 3 we apply the MDPE process to the presented example. Section 4 stipulates the need for performance related decision support. In section 5 we describe the state of the art. Finally, section 6 concludes the paper.

2 MDE for Composite Applications

Our case study involves a *Wine Seller* who gets wine supply from several suppliers, and thereafter sells the wine to different customers. The *Sales and Distribution Organization* of the *Wine Seller* is supported by a standard software product implementing standard back-end processes. The back-end process under consideration is organized in functional units providing its own business behaviour called *Process Components*. Thus, a concrete business application uses functionality provided by multiple Process Components. A complete set provided by the Business Suite can be found in [2].

The manager of the *Sales and Distribution Organization* of the *Wine Seller* now wants to extend the existing business process so that it is supported to add an extra free bottle of wine to orders of those customers who notified a quality issue for their previous order. The decision as to which wine will be added for free has to be taken manually by a wine specialist based on a wine rating and the customer's purchase over the last 12 months. Additionally, the selection has to be approved by the manager of the Sales and Distribution Organization.

This raises the need for an extended version of one Process Component, which is *Sales Order Processing* in this case. It is, however, not desirable to change

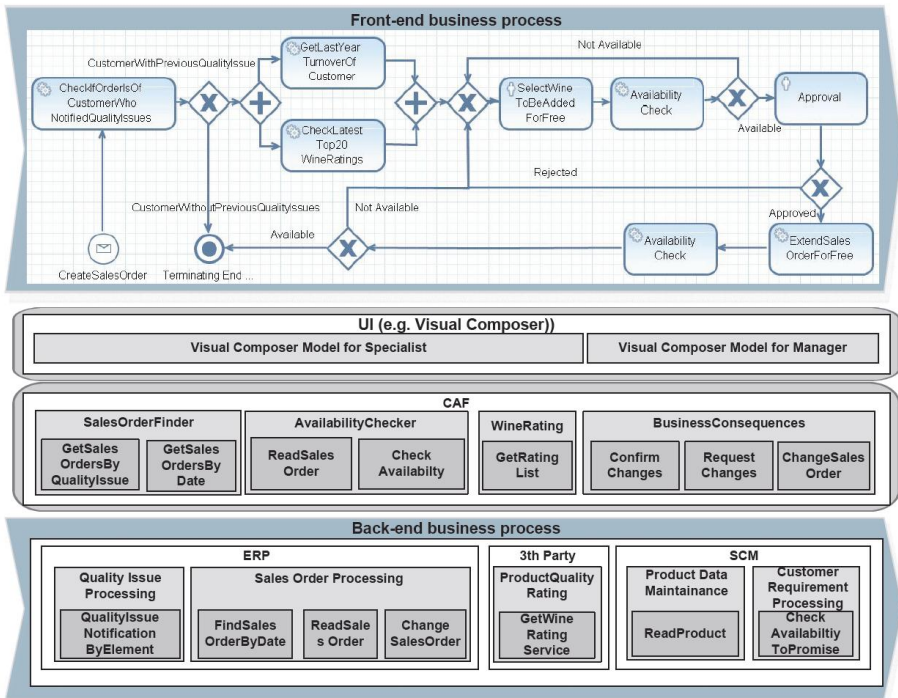


Fig. 1. Development of Composite Applications

the business process directly in the back-end because the application should be independent of the software vendor life cycle. Therefore, a platform independent technology is needed that could use the platform back-end business logic. To meet this requirement, we developed the Composite Application called **Wine Under Special Treatment (xWURST)**.

Our research mainly targets Business ByDesign systems. However, at the time of our first experiments a running instance was not available yet for our needs. Therefore, the Composite Application we use to motivate our research was implemented as an extension based on two Business Suite sandbox systems: HU2 [3] provided us with access to these Process Components which are part of SAP Enterprise Resource Planning (ERP) and HU8 [4] gave us access to the SAP Supply Chain Management (SCM) Process Components¹.

A description of the building blocks of our case study is now presented.

The bottom layer of figure 1 depicts those Process Components which are relevant for our case study. As can be seen, we mainly use ERP Process Components, such as *Quality Issue Processing*, providing Enterprise Services, such as *QualityIssueNotificationByElement*. Additionally, Process Components available in SCM are employed as well. A 3rd party service provided by an *Independent Software Provider (ISV)* is also used. For our case study, we extended the Process

¹ Interested readers can apply for a user of the HU2 and the HU8 systems [2].

Component *Sales Order Processing* with two manual steps called *SelectWine-ToBeAddedForFree* and *Approval*, which modify the behaviour of the overall back-end business process.

The functionality required to develop, deploy and maintain Composite Applications is encapsulated in the SAPs Composition Environment (CE) providing the design-time and the run-time of Composite Applications. The design time tooling, which is of interest for this paper, is integrated in the NetWeaver Developer Studio. Three groups of technologies could be identified in order to better understand the MDE-concepts in the development of Composite applications:

The Services Group, called Composite Application Framework (CAF, see figure 1 second layer from the bottom), enables the technical integration of the new front-end business logic of the Composite Application with the existing business process logic provided as services. For xWURST we used the functionality of CAF to provide infrastructure to publish services (see for instance *Get Sales Order by Quality Issue* in figure 1 that can be used by the front-end business processes. CAF also generated the code for the dependencies between the xWURST Composite Application and back-end processes. Additionally, we had to define new business logic, the 3rd party wine rating, which is defined within a new Business Object that can have a local persistence. CAF also enables the development of Business Objects with remote persistence. We published the functionality of our business object as a service called *Get Wine Rating* to be consumed by front-end business processes. Services and Business Objects provided by CAF are implemented as Enterprise Java Beans (EJBs) and can rely on the existing persistence and life cycle mechanisms of the Java EE technology.

The Views Group provides model-driven tooling for the development of user interfaces for Composite Applications, such as the *Visual Composer*, which is a completely code free environment for user interface development. This environment consumes Web Services and provides a DSL to define the mapping between service inputs and outputs to user interface elements such as forms, tables or charts. In [5] a more detailed description can be found on model-driven development of user interfaces with the Visual Composer. Other technologies, such as Adobe Interactive Forms, mobile user interface technologies, and voice-based user interface technologies are supported as well but not used in our case study. In our use case we published the central operations of xWURST as Web Services. This enabled us to use them with two wizards modelled with the Visual Composer, one for a specialist who selects the wine to be added and one for a manager who has to approve this selection (see figure 1 third layer from the bottom).

The Processes Group enables definition of new front-end business processes by a domain expert, who utilizes MDE concepts. For modelling the front-end business processes, a tool called Galaxy has not been released yet to customers or ISV's. However, we assume the availability of a tool to model front-end business processes by using Business Process Modeling Notation (BPMN) [6] as DSL for front-end business process development, as depicted by figure 1 in the fourth layer from the bottom.

Concluding, models and MDE concepts are for the development of user interfaces, front-end business processes and proprietary models of back-end processes are available. Therefore, not only the Composite Applications, but also back-end processes can be developed by applying MDE concepts and by involving domain experts in the software development process. Additionally, service-oriented concepts enable a high degree of flexibility, as presented in the xWURST application by extending a back-end process independent of the software vendor lifecycle.

Based on the xWURST scenario, the authors see the following challenges for the design of business processes and the mapping of human or artificial resources to process steps:

- Several steps within the business applications have to be processed by human resources. The decision about which human resource performs which step in a business process is not trivial to make.
- The former difficulty is further increased by the flexibility introduced with the possibility of extending the back-end processes with front-end processes by a domain expert, with no performance related skills. This can dramatically change the behaviour of already running and stable systems.

It is obvious that a domain expert, with no performance skills, cannot decide while introducing the extensions or customizations if this could lead to any performance related consequences. In the xWURST example context, the performance related issue would be that if the two additional manual steps in the back-end business process now needs more time than defined as an objective. We applied MDPE in order to deal with this problem as described in the following section.

3 MDPE for Composite Applications

MDPE [7], which is refined here, is defined as an extension of MDE enabling performance engineering based on development models and additional performance related data. Hence, the approach utilizes MDE concepts. In comparison to other approaches, the process supports n kinds of development models and m kinds of performance prediction techniques. Additionally, the process enables, as proposed by the SPE approach, stepwise horizontal refinement of performance models by automated transformations.

In order to support multiple kinds of development models and multiple performance prediction techniques and reduce the number of required transformations, MDPE uses *Tool Independent Performance Models (TIPMs)* (see figure 2), as the performance modelling language containing all information about the development models, relevant for performance engineering. This enable us not only to deal with proprietary modelling languages but also with well known modelling languages, such as UML. Additionally, tool independence is enabled which is of high value for business software vendors, such as SAP, in order to be not dependent on one specific simulation tool.

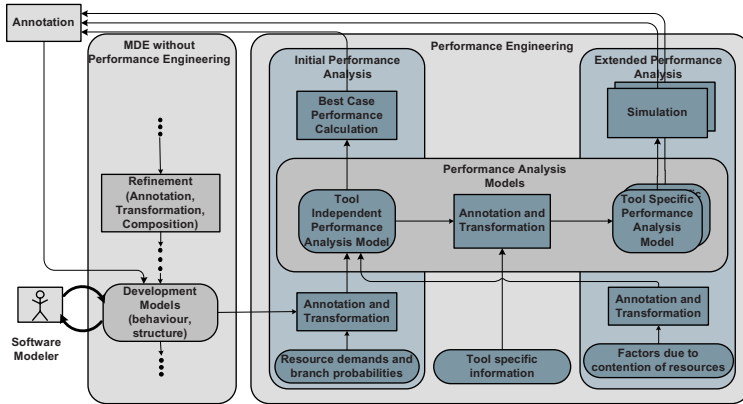


Fig. 2. MDPE concept as Block Diagram [8]

Within the MODELPLEX project, we contributed to the definition of the TIPM meta-model together with TU-Dresden, the simulation tool provider XJTech, and IBM Haifa. The TIPM meta-model has been defined as a refined and slightly extended version of the CSM [9]. It focuses on behaviour information, available resources and consumption of resources. It can be horizontally refined by annotating stepwise performance related information (resource demands, branch probabilities, factors due to contention for resources).

Initial Performance Feedback is based on development models, resource demands of steps and branch probabilities. The computation of the mean, best-case and worst-case response time values can be done based on TIPMs.

Extended Performance Feedback additionally takes factors due to contention for resources into account. Performance prediction is more advanced than in the former case. Multiple tools are supported that enables multiple performance prediction techniques. Hence, *Tool Specific Performance Models (TSPMs)* (see figure 2), are used as input for performance prediction tools.

For the xWURST case we tried to utilize annotated proprietary models of back-end business processes and of front-end processes for MDPE. We consider that both the original and the customized Process Component are available as the UML Activity Diagram. The initial UML representation has been selected due to its popularity and good tool support for editing models. Additionally, UML provides tool supported meta-model extension mechanisms via UML-profiles. Figure 3 shows the UML Model of the Sales Order Processing Process Component which is extended by the xWURST front-end-process. It is obvious that it is required to not only consider models of one Process Component but of a whole back-end process consisting of multiple Process Components for future research, in order to perform more realistic business performance analysis.

The composition of front-end and back-end process models but also the composition of models of different Process Components can be realized with the Reuseware tool. Further details about the Reuseware tool are available at [10].

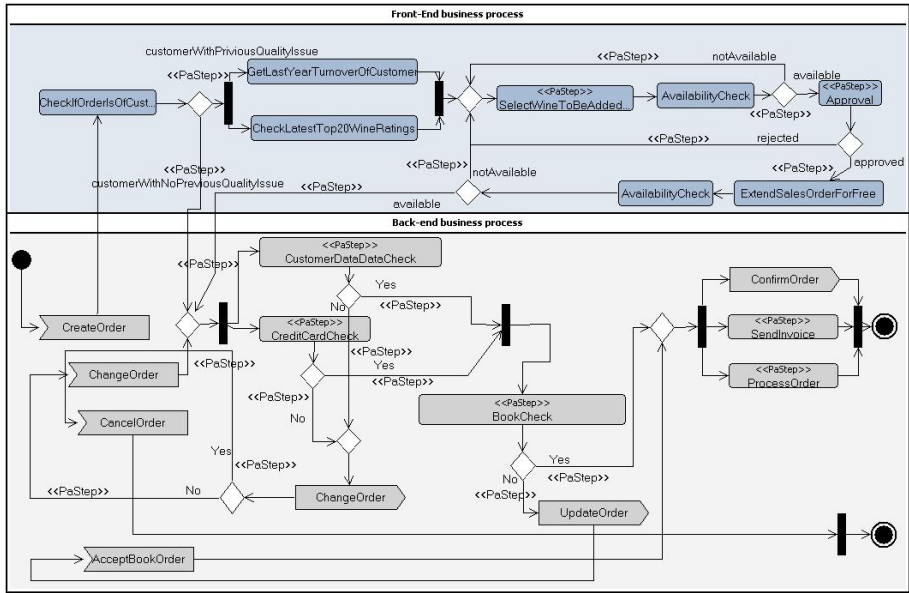


Fig. 3. Example model of the Process Component *Sales Order Processing* extended with new front-end logic

We can assume that the back-end process of the xWURST was already running for a period of time. Hence, we are able to get the average resource demands in employee time for each manually processed Action in the Activity Diagram of our original back-end business process out of already existing business performance analysis tools on top of SAPs Business Intelligence (BI). Since automated Actions do not acquire human resources we ignored them. Additionally, based on the data in the BI, we are able to calculate the probabilities of paths in the Activity Diagram and know about the resource mapping, in particular, how many employees are working on which manual step in our behaviour model. For newly defined front-end processes assumed values have to be annotated in order to specify resource demands, probabilities of paths and resource mappings.

Like other authors such as [11], we initially ignored the business domain-specifics. Due to the neglect of business domain-specifics we were able to manually annotate the resource demands of manually processed actions, path probabilities and resource mapping to the UML model elements conforming to the Performance Analysis Modelling (PAM) package of the MARTE profile [12] which was originally intended to extend “UML for model-driven development of Real Time and Embedded Systems” [12]. MARTE is currently the latest profile for performance annotations specified by the Object Management Group (OMG). In [13] we propose a tool supported approach to perform this task systematically for large model repositories. In order to automatically transform the the annotated UML model to a TIPM, we implemented a transformation between the UML meta-model and the TIPM meta-model. We selected the ATLAS

Transformation Language (ATL) [14] to implement the transformation. Another transformation from TIPMs to TSPMs for the Simulation Tool AnyLogic [15] has been developed by developers of XJTech². Hence, for UML as development model we were able to perform an *Extended Performance Analysis* in the MDPE process.

ATL is the standard transformation language for EMF based models and good tool support is provided as well. AnyLogic enables on-the-fly analysis by simulation as well as graphical representation of simulation models. Hence, it was applicable for our experiments.

From the simulation based on the UML representation of the business process we are able to predict the throughput time for a Sales Order. We are also able to predict the utilization for each department for the Wine Seller.

However, we identified the need for taking conditional resource mappings into account since for the business performance analysis of business processes it could be required to express that, for instance, *if the value of the added wine increases by at least 10 Euro, the modification has to be approved by a manager, otherwise by a sales representative*. Hence, our current MDPE implementation does not support business performance analysis yet as the PAM package of the MARTE profile but also the TIPM is not dealing with conditional resource mappings. Thus, we have to consider these specifics for an updated version of the TIPM and for the definition of a performance annotation formalism for our proprietary models.

Additionally, we identified the need for performance related decision support described in the following section.

4 Identified Challenge for Future MDE Research

In [16] the problem of layered bottle-necks is described. According to [16], a bottle-neck in one layer may in fact result in a bottle-neck in another layer by a push-back effect which makes interpretation difficult.

This problem can occur in the case of layered use of resources and can be applied for business performance where one resource, e.g. the Sales Order Processing department in figure 4, requires services of other resources, such as the department responsible for the manual Customer Requirement Processing steps to process its requests.

If, for instance, a simulated utilization of the three different departments performing *Sales Order Processing*, *Customer Requirement Processing* and *Supplier Invoice Processing at Customer* is over 90 percent, each of the departments looks initially like a bottleneck.

As can be seen, the automated generation of performance simulation helps to obtain predictions for several performance parameters, but the results have

² The details of the transformations are not relevant here. However, the interested reader is referred to [7] containing a description and a reference to the sources of a direct transformation from similar UML models to AnyLogic simulation models.

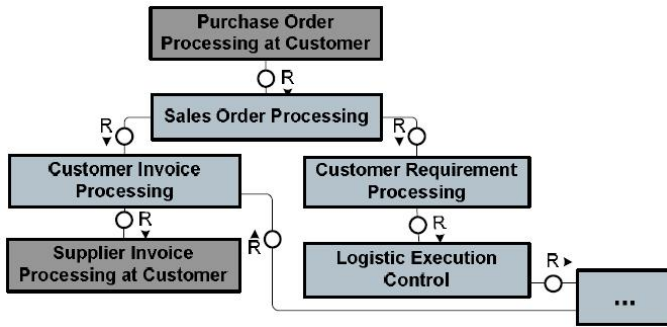


Fig. 4. Layered use of resources involved in a simplified business process as block diagram [8]

still to be interpreted in order to make a decision about how the business process design and the resource mapping needs to be changed. Specifically for the xWURST use case, questions arise as to which of the departments should get which and how much additional resources.

Based on these questions, which are specific for our use case, we identified the following general requirements for targeting performance related decision support for MDE:

- *Information filtering:* For the domain expert developing Composite Applications, like xWURST, only relevant information should be provided with respect to the modification constraints and performance objectives. In this paper performance objectives are defined as performance requirements and performance improvements. Performance requirements include, for instance, maximum throughput time for a scenario or minimum resource utilization. Performance improvements are concerned with maximizing the resource utilization and minimizing the response time of the modelled system.
- *Information interpretation:* Provide support to domain experts, who generally lack the performance engineering knowledge, in interpreting the measurement data, performance models, and performance prediction results by providing performance related metrics or concrete proposals about how the performance can be improved. A similar observation can be found in [17]. There, it is mentioned that performance information and models still have to be interpreted: “We must [...] learn how to combine measurement data interpretation with model interpretation and to get the most out of both”. A first step towards this kind of interpretation is taken in [16], in which a metric is introduced for the detection of bottle-neck sources for decision support, in order to apply improvements and realistically estimate their effectiveness. The decision support in that work is based on a metric called “Bottle-Neck Strength” offering a first step towards combining measurement interpretation and model interpretation.
- *Systematic model synchronization:* Provision of an approach for the systematic integration of performance metrics or proposals back into the

development models in the MDE process, which are used by domain experts for their daily work. Hence, the domain expert should not be required to analyse simulation models which may contain unknown model elements.

- *Assessment visualization*: Visualization support for the graphical representation of identified performance metrics or proposals.

5 State of the Art

A number of approaches, such as [18], [19], [20], [21] and [22] are available to generate performance analysis models from development models by the utilization of MDE techniques. In the business domain, products like IBM WebSphere Business Modeller [23] enable generation of simulations out of business behaviour models. However, an approach is required to directly use BI-data and models of back-end and front-end business processes which are available due to the application of MDE concepts for the development of Composite Applications but also back-end processes. Additionally, performance related decision support based on models is not considered yet since the relationship between performance objectives, modification constraints and design decisions is currently not taken into account.

In [16], the authors define the “Bottle-Neck Strength” which is a first step towards addressing the identified requirement of *Information interpretation*. In any case, this approach is concerned only with the calculation and interpretation of one single metric, and does not address the integration of performance related decision support in the MDE process. In the former section we described the need for such an integration.

6 Conclusion

In this paper we presented how MDE concepts can be used for the development of Composite Applications for a Wine Seller on top of Enterprise SOA. We also showed that development models describing the behaviour of back-end and of front-end business processes can be used to apply MDPE enabling business performance prediction when performance data and resource mappings are accessible and can be mapped to these models.

Based on our first hand experiences requirements have been identified to realize valuable business performance analysis. As a follow up we are currently working on a transformation of the used proprietary models to TIPMs by taking multiple Process Components and conditional resource mappings into account in order to deal with business domain specifics. Also, an extension mechanism which enables refinement of proprietary models towards the business performance domain has to be defined.

The TIPM saves us a lot of effort, since we can reuse the transformation from TIPM to the tool AnyLogic in order to simulate our proprietary models. The simulation tool AnyLogic is usable for business process simulation but can be easily replaced by another one due to the tool independence introduced by the

TIPM. This tool independence is of high value for business software vendors, such as SAP, to be not dependent on one specific simulation tool.

Concluding, we identified the issue that, even if business domain specifics are taken into account, Extended Performance Analysis is still not appropriate to support domain experts, with no performance expertise, in the MDE process. Based on this observation we identified requirements for how such decision support should look like.

Acknowledgement

This research has been co-funded by the European Commission within the 6th Framework Programme project MODELPLEX contract number 034081 (cf. <http://www.modelplex.org>).

References

1. Heidasch, R.: Get ready for the next generation of sap business application based on enterprise service-oriented architecture (enterprise soa) (2007)
2. SAP AG: SAP Enterprise Services Workplace (2008), <http://erp.esworkplace.sap.com>
3. SAP AG: HU2 sandbox system (ERP6.0) (2008), <https://erp.esworkplace.sap.com/sap/bc/gui/sap/its/webgui>
4. SAP AG: HU8 sandbox system (SCM2005) (2008), <https://scm.esworkplace.sap.com/sap/bc/gui/sap/its/webgui>
5. Bnnen, C., Herger, M.: SAP NetWeaver Visual Composer. SAP PRESS (December 2006)
6. OMG: Business Process Modeling Notation Specification, Final Adopted Specification (2006)
7. Fritzsche, M., Johannes, J.: Putting Performance Engineering into Model-Driven Engineering: Model-Driven Performance Engineering. In: Engels, G., Opdyke, B., Schmidt, D.C., Weil, F. (eds.) MODELS 2007. LNCS, vol. 4735. Springer, Heidelberg (2007)
8. Knöpfel, A., Gröne, B., Tabeling, P.: Fundamental Modeling Concepts: Effective Communication of IT Systems. John Wiley & Sons (2006)
9. Petriu, D.B., Woodside, M.: An intermediate metamodel with scenarios and resources for generating performance models from uml designs. In: Software and Systems Modeling, vol. 6(2), pp. 163–184 (2007)
10. Henriksson, J., Johannes, J., Zschaler, S., Afmann, U.: Reuseware – adding modularity to your language of choice. In: Proc. of TOOLS EUROPE 2007: Special Issue of the Journal of Object Technology (to appear, 2007)
11. Bertolino, A., Marchetti, E., Mirandola, R.: Real-time uml-based performance engineering to aid manager’s decisions in multi-project planning. In: WOSP 2002: Proceedings of the 3rd international workshop on Software and performance, pp. 251–261. ACM Press, New York (2002)
12. OMG: UML profile for modeling and analysis of real-time and embedded systems (2007)

13. Mehr, F., Fritzsche, M., Schreier, U.: QUAL: A Query and Annotation Language for the UML models of Service-oriented Applications. In: International Journal of Business Process Integration and Management (IJBPM) (submitted, 2007)
14. ATLAS Group: ATLAS transformation language (2007),
<http://www.eclipse.org/m2m/at1/>
15. XJ Technologies: AnyLogic — multi-paradigm simulation software,
<http://www.xjtek.com/anylogic/>
16. Franks, G., Petriu, D., Woodside, M., Xu, J., Tregunno, P.: In: QEST 2006: Proceedings of the 3rd international conference on the Quantitative Evaluation of Systems, Washington, DC, USA, pp. 103–114. IEEE Computer Society, Los Alamitos (2006)
17. Woodside, M., Franks, G., Petriu, D.C.: The future of software performance engineering. In: FOSE 2007: 2007 Future of Software Engineering, Washington, DC, USA, pp. 171–187. IEEE Computer Society, Los Alamitos (2007)
18. D'Ambrogio, A., Bocciarelli, P.: A model-driven approach to describe and predict the performance of composite services. In: WOSP 2007: Proceedings of the 6th international workshop on Software and performance, pp. 78–89. ACM Press, New York (2007)
19. D'Ambrogio, A.: A model transformation framework for the automated building of performance models from uml models. In: WOSP 2005, pp. 75–86. ACM Press, New York (2005)
20. Cortellessa, V., Gentile, M., Pizzuti, M.: Xpmit: An xml-based tool to translate uml diagrams into execution graphs and queueing networks. In: QUEST 2004: First International Conference on Quantitative Evaluation of Systems. IEEE, New York (2004)
21. Gordon Gu, P., D.C.P.: Xslt transformation from uml models to lqn performance models. In: WOSP 2002, pp. 227–234. ACM Press, New York (2002)
22. Ramrao, W.: Transformation of uml design model into performance model: A model driven framework. In: ECOOP Student Workshop (2006)
23. IBM: Websphere, <http://www-306.ibm.com/software/integration/wbimodeler/>

From Business Architecture to SOA Realization Using MDD

Avivit Bercovici , Fabiana Fournier, and Alan J. Wecker

IBM Research Laboratory in Haifa, Israel
{avivitb, fabiana, wecker}@il.ibm.com

Abstract. One of the major challenges in today's complex business environment is the delivery of a complete solution from business architecture design downstream to SOA IT realization. The IBM Service Oriented Modeling and Architecture (SOMA) methodology attempts to address this objective. Our project captures information from business architecture design and presents this information in a graphical user interface for its later utilization by the solution development team, compliant to the SOMA methodology. Our model-driven development (MDD) cycle consists of modeling with XML Schema Definition (XSD), generating code with Eclipse Model Framework (EMF) and Eclipse Graphical Model Framework (GMF), customizing the code, and testing the solution. The tool was tested on a supply chain management scenario. The results demonstrated the feasibility of capturing business design and using the artifacts for IT realization with MDD. Our overall MDD experience is discussed.

Keywords: CBM, SOA, SOMA, services, EMF, GMF, collaboration diagram.

1 Introduction

Advances in information technology (IT), coupled with the increased competition brought about by globalization, are strongly affecting the business environment. To succeed in this environment, businesses are transforming themselves and becoming more agile and more efficient.

Business leaders are demanding more from business and looking for new ways to achieve growth, productivity, and an optimized technology environment aligned to the business. As industry leaders position themselves to adapt and thrive in an environment of continuous, unpredictable change, a new approach is required to analyze and transform the business [1]. Many companies are turning to *Component Business Modeling* (CBM) [2], [3], [4], [5], [6].

Component Business Modeling (CBM) offers a new way to model a business, enabling specialization in areas where it commands a comparative advantage in the marketplace. CBM is an IBM proprietary business architecture methodology. Business architecture can be defined as the grouping of business functions and related business objects into clusters ("business domains") over which meaningful accountability can be taken as depicted in the high-level description of the related business processes [7]. For more details on business architecture, see [8], [9], [10], [11], and [12].

CBM is an aggregation of models, methods, and techniques designed to organize, understand, evaluate, and ultimately transform an enterprise [2], [3], [4], [5], [6]. *Business components* are the modular building blocks that make up enterprises according to CBM. Business components are bounded groups of tightly linked business activities. Instead of stages in a traditional business process, we can think of components as discrete nodes in a configurable value network, and the whole organization as a collection of components networking together [2], [3], [6]. Components have well-defined interfaces; each receives input, adds value, and outputs the results to other components in the network [6].

In this paper, we describe our approach for driving an end-to-end solution from a componentized business architecture to IT solution following the SOMA methodology using MDD. Our project captures information from business architecture design and presents this information in a graphical user interface for its later utilization by the technical architecture team compliant to the SOMA methodology. The tool developed was tested on a Supply Chain Management (SCM) scenario. The results demonstrated the feasibility of capturing business design and using the artifacts for IT realization with MDD. We discuss our practice and experience in implementing such a solution and present lessons learned.

The rest of this paper is organized as follows. Section 2 briefly describes background and some basic terms. Section 3 outlines the approach and technologies used. Section 4 describes a case study in which the developed tool was tested. Lessons learned are presented in Section 5. We summarize the paper with concluding remarks and future directions.

2 Background and Terms

2.1 Business Component

A component-based approach divides an enterprise into a set of non-overlapping modular building blocks, which IBM calls *components*. A component is defined along five dimensions [2], [3]:

- A component's *business purpose* is the logical reason for its existence within the organization, as defined by the value it provides to other components.
- Each component conducts a mutually exclusive set of *activities* to achieve its business purpose.
- Components require *resources*: people, knowledge, and assets that support their activities.
- Each component is managed as an independent entity, based on its own *governance model*.
- Similar to a standalone business, each business component provides and receives *business services*.

2.2 Business Service

A business service is some well-defined value offered to other business components [3], [5]. Business components interact with each other by providing and consuming

business services [3]. This enables businesses to think of a component as a service center within the enterprise [3]. Business components expose the functions they support only through their provided or offered business services. These services are enabled by the component activities contained within the components.

Following the principle of a component being a “black box” to other external business components in the network [2], [3], all activities and “business logic” are encapsulated in a single component. Furthermore, a component’s activities are unique to a single component [13].

Components interact with one another via services based on the requirements of the business activities defined within their operations. When an internal business process requires information/functionality that does not appear in its component’s scope, the business process must get it from external resources. For that purpose, a business service offered by an external component is invoked through a *service invocation activity* [5], [13].

2.3 Business Collaboration

To achieve the desired business outcome, components collaborate with other components through services [2]. When one service component requires business information from another service component, an external service is invoked, and an *interaction* occurs. In other words, an interaction involves exactly two components—a service requester and a service provider—which “agree” on a single service. *Collaborations* are interactions between two or more components, working together towards a common business goal. As services reflect the component’s functionality to the external world and serve as component-only interfaces, collaborations can be formulated only through the exchange of business services. Collaborations do not include semantics for the service control flow such as decision nodes, joins and merges, if-else rules, and do-while loops. Alternatively, they do provide a high-level view of the “wiring” between the business services interactions [13]. A *collaboration diagram* depicts a business collaboration in swim lane form. Each swim lane describes possible interactions of a single component with other components in the collaboration on a timeline. Fig. 1 provides an example of an illustrative collaboration diagram. In this collaboration diagram, three components (Component 1, Component 2, and Component 3) interact through services Service 1.1, Service 2.1, Service 3.3, and Service 2.4, as depicted in the diagram. The envelope icon represents the business items transported through the interactions (represented by the arrows).

2.4 Business Item

Business components are responsible for managing business items (BIs) and exposing their functionality to the external environment through business services. BIs are tangible or intangible assets that are managed by the enterprise through its components and exploited to create economic value for the firm. Examples of tangible BIs include people, buildings, customer accounts, manufacturing machine capacity, supplier orders, and documents containing business information. Examples of intangible business entities include brand, marketplace insight, exclusive “know how” capability, knowledge, design, and content. CBM design enforces each BI to be

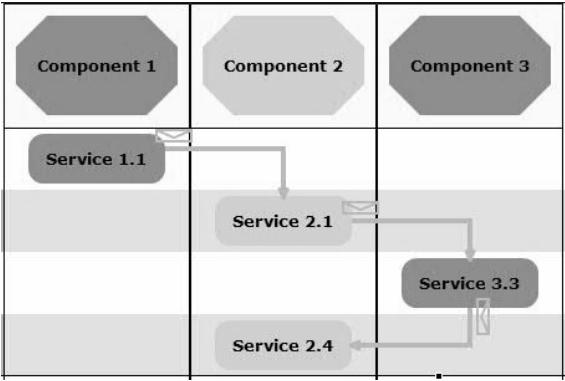


Fig. 1. Illustrative example of a collaboration diagram

managed by a single component, while a component can manage more than one BI. Moreover, CBM design rules ensure that the specified business component is accountable for the whole lifecycle of its BIs [13].

2.5 SOA and SOMA

Service Oriented Architecture (SOA) is a discipline that spans the entire spectrum from business architecture to IT implementation. At the business end of the spectrum are business components that describe a business as a collection of service-bounded building blocks of the entire business. The business services are generally coarsely grained and usually specified informally. However, at the other end, IT services (e.g., web services) are fine grained and specified precisely. The goal is the realization of business services that a component provides, as a composition of well-defined IT services [5].

Basically, SOA is a "business to IT aligned" approach in which applications rely on available services to facilitate business processes. SOA is a model that guides the establishment of a loosely coupled system with flexibility and extensibility. Implementing an SOA mainly involves componentizing enterprise and/or developing applications that use services, making applications as services for other applications to use, and so on [9].

Although in CBM, components offer services to each other, and in SOA there are software components that offer services to one another, these are two different distinct concepts and the transformation between them is not trivial [10]. The business service concept has attributes that are relevant to communication among business people (such as terms and conditions associated with business service consumption, governance, and management), but it does not include the solution aspects. It is important to understand this distinction to fully appreciate the concepts associated with modeling the business architecture of a service-oriented enterprise [10].

For mapping the business structure to the IT layer, IBM has developed SOMA. SOMA is a consulting-oriented services development method.

Service-oriented modeling is necessary for the creation of an SOA. This modeling uses the results of the business componentization analysis as inputs. The output is an

SOA architecture independent of any specific technology that can then be realized using the appropriate technologies. SOMA starts from the description of the business process to be implemented. The most important result from the SOMA analysis is the services model, which comprises a set of IT services that support the business services and processes and their goals. For more details on SOMA refer to [3].

3 Approach

The purpose of our tool is to capture information at the business architecture level and transform it to artifacts that the solution development team applying IBM SOMA method can utilize. This section describes the requirements from the tool and the system architecture used to provide the desired deliverables.

3.1 Requirements

Our major requirements were:

- **Model Formalization** - One of the major requirements was to formalize the business architect's mindset in a model. From our perspective, a business architect (BA) is someone who designs and develops a company's business architecture. This was done by gathering information (mostly in PowerPoint) and having conference call sessions with the business architectures that sponsored the project.
- **End-to-end Solution** - Our project captures information from business architecture design and presents this information in a graphical user interface for its later utilization by the solution development team compliant to the SOMA methodology. The goal is to demonstrate a possible solution that encompasses all phases from business architecture information to IT realization, providing an end-to-end business solution.
- **Data Capture** - The tool captures the component names and business goals along with the activities performed by the components. In addition, for each component, the services it provides and receives and the business items transported by these services are specified. In alignment with CBM methodology, some of the activities encapsulated in a component participate in the realization of the services provided by the component, while other activities are service invocation activities [13]. To capture the data, two types of editors were designed. A tabular editor was used to collect data concerning individual components including services, goals, activities, and business items, while a graphical diagram editor was used to collect information concerning business collaboration diagrams.
- **Output** - The work-in-progress artifacts provided by the tool are used by the business architect to iteratively refine the CBM design and contents. As mentioned before, the outputs of the tool are transferred to the solution development team for IT realization in alignment with the SOMA methodology. The main output artifact produced by the tool is an XML file containing all of the CBM design and contents: component names, goals, activities, services, business items, and collaboration diagrams.

- Look-and-feel and Ease-of-use - The primary user of our tool is the business architect, who is not an IT person (who is mostly familiar with data entry applications), but rather a business user who is used to work with office tools (e.g., Excel, PowerPoint, and Word). Therefore, the look and feel of our tool needed to be easy to learn and use without requiring any pre-requisite software in the user's machine.

Fig. 2 illustrates the flow of data end-to-end from the BA input to the consumption of the artifacts by the solution development team.

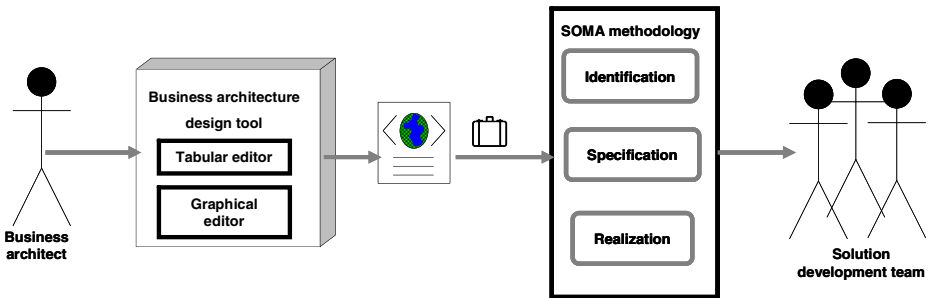


Fig. 2. End-to-end data flow

3.2 Tools Used

The Eclipse development platform was used for the development phase of the project. Eclipse is an open source community, whose projects are focused on building an open development platform comprised of extensible frameworks, tools, and runtimes for building, deploying, and managing software across its lifecycle [14]. We used a number of Eclipse projects: XML Schema Definition (XSD) editor, Eclipse Modeling Framework (EMF [15]), and the Graphical Modeling Framework (GMF [16]).

The EMF project is a modeling framework and code generation facility for building tools and other applications based on a structured data model. We started from a model specification described in XSD. XSD provides a model and API for manipulating components of an XML Schema. EMF provides a tool that transforms the XSD to an Ecore file (a model specification described in XMI (XML Metadata Interchange)). EMF provides tools and runtime support to produce a set of Java classes for the model, along with a basic editor and a set of adapter classes that enable viewing and command-based editing of the model [17]. The generated Java classes were changed during the customization phase to provide a progressive editor and other capabilities. The EMF platform was used to generate the tabular editor, which was then used to capture the components' data content.

The Eclipse Graphical Modeling Framework (GMF) provides a generative component and runtime infrastructure for developing graphical editors based on EMF and GEF (Graphical Editing Framework [18]). GEF allows developers to take an existing application model and quickly create a rich graphical editor. The combination of EMF and GEF enabled the creation of the model-based graphical editor used to model collaboration diagrams.

3.3 Architecture of Our Solution Cycle

Prior to the tool, the business architects captured their information in a number of Excel spreadsheets. We needed to model that information and 1) allow the architects to capture that information and validate it as much as feasibly possible 2) pass this information on to other processes in the food-chain. Drawing mockups gave us our requirements for how our end-user interface should look and feel. The XSD file gave us the original definition of how the business information would be transferred.

To capture the information at the business architecture design level, we composed a metamodel that describes the business architects' way of work. Next we looked at the artifacts SOMA uses for the transformation to the realization phase and modeled how information can be transferred.

Model Using XSD. Using the IBM Rational Software Architect (RSA) tool, a UML (Unified Modeling Language) model of the BA Excel spreadsheets was crafted. Since there was a requirement to produce XML and not XMI files, the model was manually transferred to an XSD model, which was used to generate an Ecore model.

Generate Code via EMF and GMF. From the Ecore model, using EMF JET [19] technology, code was automatically generated for both the model and the end-user interface for the table editor. To generate the base code for the graphical diagram editor, we needed to create a graphical definition file (consisting of node figures and connection figure), a tooling definition file (creation tools), and a mapping file (mapping from the model to the graphical definitions) using GMF wizards. Once that was done, code was generated for the graphical diagram editor.

Customize. To meet the ease-of-use requirement, we changed the automatic generated editors. The EMF editor needed to be customized to give the user the look-and-feel of Excel-like spreadsheet typing (primarily in place editing, and multiple column table support). To support multiple-column in-place table editing, a custom

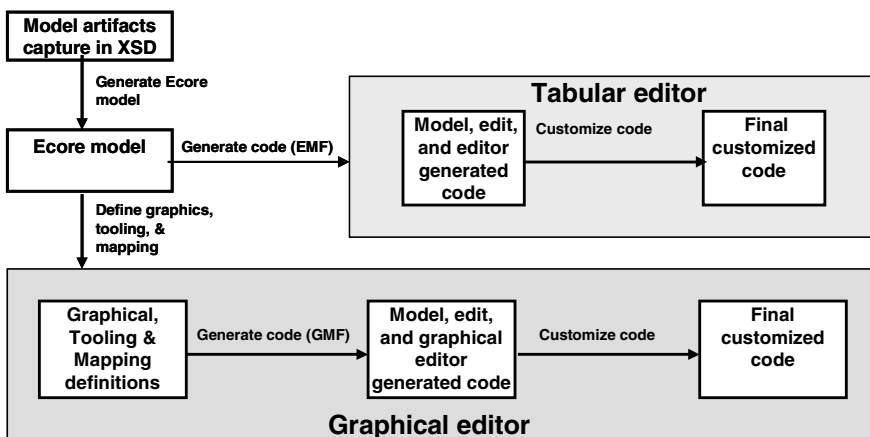


Fig. 3. Development cycle and tools deployed

table widget was developed. The model needed less customization and a small number (~10%) of methods was added or altered from the generated form. The GMF generated editor needed to be customized to conform to our mock-up requirements. Items such as layouts, alternating colors, and pop-ups for in-place choices needed to be altered or added to the generated code.

Test and Cycle. The methodology used was that of iterative, agile [20] design. We did a number of cycles adding features to each cycle and occasionally changing the design. Changes in a cycle could involve changes to our UML model, XSD definitions, GMF model files, Ecore model, or customized code. Each individual change caused propagation of additional derived changes throughout the development cycle. Fig. 3 depicts our development cycle along with the tools deployed.

4 Case Study

The solution was tested on a prototypical supply chain management scenario. The components in this scenario needed to support a supply/demand synchronization facility that balances and maintains customer product positions according to sales, customer agreements, and availability. The specific scenario does on-demand allocation of a product across multiple sales lot inventory states.

The scenario included three components: the first component provides a current view of product allocations across all customers. The second tracks available product sales and submitted product replacement requests. The third component orders products based upon need and current allocation of available inventory. These components interact to formulate and maintain a position for the allocation of a product across customers.

The first stage of the case study was capturing and collecting the data through the tabular and the collaboration diagram editors.

The next stage was transforming the outputs for use in an internal IBM SOMA tool. The transformation was done using a model-to-model transformation via UML prototypes [21].

The solution development team received all artifacts produced by our tool. Using the SOMA methodology, these artifacts were consumed and processed to deliver an IT SOA solution.

The end-to-end solution was tested successfully, and checked to see if it could perform the proposed scenario. In the next section, we describe our findings regarding the use of the tool.

4.1 Findings

Capturing the information in a model-driven tool allows validation checks and detection of possible conceptual bugs, previously unnoticed by the business architects while using Excel sheets. Over 10 conceptual and functional bugs were found in the components details and collaboration diagram descriptions. These bugs were reviewed with the business architecture team, which emphasized to them the additional power and advantages of working with our tool.

Capturing the information was an iterative process, as understanding the BA's intent evolved over a number of discussions and through our demonstration of the tool.

Feedback was received from users on both ends of the tool. The first feedback we received was from the BA team. They were impressed by the tool's ability to do consistency checks of the information fed. This reduced the number of problems in later phases, when this information is passed on to the solution development team. In later phases, the problems found are more expensive in terms of time and effort (the user at that stage is a solution architect rather than a business architect and the resolution of such problems requires a greater amount of time). Good feedback regarding the tool's ease-of-use and its graphic visualizations were also provided.

The second feedback we received was from the solution development team. They reported ~30% reduction in the total effort invested in the SOA IT realization phase.

Overall, the accomplishments of this tool are as follows:

- Business architecture design artifacts were captured by a model-based tool and produced meaningful artifacts related to CBM and SOMA methodology.
- The same model was used for both the business architecture team and IT realization team without forcing them to use a different methodology.
- A feasible, complete, repeatable, generic solution was produced for the problem described above.
- Model-to-model transformation of the business architecture information was performed and the information transferred in a structured way. The major byproduct of this automatic transformation is the dramatic reduction of the number of bugs in this process, which was previously performed by people from the solution development team (we saved errors both in the business architecture and in the transformation).
- The tool enabled a knowledge artifact transfer to the SOMA methodology in a structured way. Previously, it was clear what necessary inputs for the SOMA methodology are, but deriving them from a business design was somewhat of an art requiring skilled personnel.
- The tool can be reused to produce repeatable end-to-end solutions. Since the tool is not specific for the SCM realm, it can be reused for any other domain. When an engagement for a different domain will occur, the consulting teams could reuse this tool. The reuse of this tool could be made at two levels: first, existing data in the tool could be reused. A client might have a need for similar components, or slightly different components that can be built reusing the data in the existing components. The second level of reuse could be for the solution teams. If inputs to the solution teams are similar, then the realization phase could also reuse realizations done in previous engagements.

5 Lessons Learned

In this section we discuss the main findings concerning the development frameworks used throughout the project and MDD.

5.1 Experience in Development

The learning curve for these technologies is steep (three-plus weeks each), even for experienced developers. Although tutorials are available for certain topics, in general there is a scarcity of information. However, once that obstacle was overcome, initial working versions of the editors were rapidly generated (2-4 days). The customization phase required a longer period of time. Again, although forums exist, there is a paucity of help resources for non-standard problems.

5.2 Analysis of Key Technologies

In this section, we discuss the technologies used to develop the tools, and their advantages and disadvantages.

In general, the editors for model definition were easy to use and their usage intuitive. The code generated was efficient, understandable, and relatively easy to modify. The mechanism for round-tripping works well for code generation, excluding the code generated for constructors.

Conversely, the ease of usage of the technology depended on its maturity. The ranking of most mature to emerging technology, and correspondingly the ranking of easiest to most difficult to use, was UML, XSD, EMF, and finally GMF. As long as standard items were being developed, the use of the various tools was straightforward. However, as soon as one strayed into modifications, it became cumbersome. As mentioned above, the round-tripping from the Ecore model to the code worked well. What didn't work as well was the round-tripping from XSD to Ecore.

The use of wizards in EMF and GMF was complicated at times because it wasn't always clear where errors were generated from. An even greater difficulty was that at times, it wasn't even clear what the nature of the error exactly was.

Non-standard layout in GMF proved problematic but surmountable.

Connection to the semantic model, and diagram partitioning, are areas that are not sufficiently documented. Creating variables so that colors and sizes could be used consistently across the application was not part of the framework's capabilities.

One of the main drawbacks we encountered was related to the lack of support for the transformation from UML to XSD and Ecore, and vice-versa. This lack of capability imposed upon us the need to maintain two separate models (UML and XSD). As previously mentioned, the XSD model was needed to produce an XML file (an XMI file can be created from UML).

6 Conclusions and Future Work

Advances in IT, coupled with increased competition, are strongly affecting business structures. To succeed in this new environment, many companies are turning to CBM as a new approach for analyzing and transforming their business.

One of the main challenges in an SOA enterprise is the bridge from the service-oriented architecture as provided by CBM, downstream to the IT realization of the services. IBM SOMA methodology addresses this gap.

In this paper, we described our experiences following SOMA methodology by developing a tool that captures business design and content in a MDD manner. The

artifacts generated by the tool are transferred and consumed by a solution development team to realize the IT services. This end-to-end solution was tested successfully on a supply chain management scenario.

The MDD approach has proven to be efficient in developing the tool and the necessary artifacts along with their transformation for later use. Due to MDD, changes to the model can be propagated easily and in a coherent manner, shortening the total lead time of the complete solution provided.

Future work should include testing the tool and the approach on larger engagements. Improvements to the tool include the use of a central repository for the sharing of the designs and contents for their reuse. Smart searches connected to the model for the management of the artifacts produced by the tool can enrich the level of reusability and repeatability achieved by the tool.

References

1. Ortiz, D., Ramchandani, K., Harwood, K., Pich, N., Bicard-Mandel, J., Puckle Hobbs, E., Im, S., Ikeda, S.K., Miller, P.D.: New Competitive Weapons in the Insurance Business, <http://www-935.ibm.com/services/us/imc/pdf/g510-4033-new-competitive-weapons.pdf>
2. Pohle, G., Korsten, P., Ramarmurthy, S.: Component Business Models – Making Specialization Real (2005), <http://www-935.ibm.com/services/us/imc/pdf/g510-6163-component-business-models.pdf>
3. Cherbakov, L., Galambos, G., Harishankar, R., Kalyana, S., Rackham, G.: Impact of Service Orientation at the Business Level. *IBM Systems Journal* 44(4), 653–668 (2005)
4. Ramamurthy, S., Robinson, M.: Simplify to Succeed (2005), <http://www-935.ibm.com/services/us/imc/pdf/g510-9109-00-simplify-to-succeed-retail-banking-in-2005-full.pdf>
5. Flaxer, D., Nigam, A.: Realizing Business Components, Business Operations and Business Services. In: CEC 2004, pp. 328–332 (2004)
6. Latimor, D., Robinson, G.: Component Business Modeling – Financial Services Firms Prepare for an On-demand World, <http://www-935.ibm.com/services/us/imc/pdf/ge510-3607-00f-component-business-modeling.pdf>
7. Business Architecture definition, http://en.wikipedia.org/wiki/Business_architecture
8. Dandashi, F., Siegers, R., Jones, J., Blevins, T.: The Open Group Architecture Framework (TOGAF) and the US Department of Defense Architecture Framework (DoDAF) (2007), http://www.mitre.org/work/tech_papers/tech_papers_07/06_0987/06_0987.pdf
9. Liang-Jie, Z., Jia, Z., Hong, C.: *Services Computing*. Springer, Heidelberg (2007)
10. Nayak, N., Linehan, M., Nigam, A., Marston, D., Jeng, J.-J., Wu, F.Y., Boullery, D., White, L.F., Nandi, P., Sanz, J.L.C.: Core Business Architecture for a Service-Oriented Enterprise. *IBM Systems Journal* 46 (2007)
11. Sessions, R.: Comparison of the Top Four Enterprise Architecture Methodologies (2007), <http://www.objectwatch.com/whitepapers/4EAComparison.pdf>
12. Zachman, J.: A Framework for Information Systems Architecture. *IBM Systems Journal* 26(3), 276–292 (1987)
13. Fisher, A., Fournier, F., Gilat, D., Rackham, G., Razinkov, N., Wasserkrug, S.: A Top-Down Approach from Service Centers to Business Processes. In: *IEEE/INFORMS International Conference on Service Operations and Logistics, and Informatics (SOLI 2007)*, pp. 400–405 (2007)
14. Eclipse website: <http://www.eclipse.org/>

15. EMF documentation website, http://www.eclipse.org/projects/project_summary.php?projectid=modeling.emf
16. GMF documentation website, http://www.eclipse.org/projects/project_summary.php?projectid=modeling.gmf
17. EMF description article, <http://help.eclipse.org/help33/index.jsptopic=/org.eclipse.emf.doc//references/overview/EMF.html>
18. GEF website, <http://www.eclipse.org/gef/>
19. Using JET Transformations with EMF Documents, <http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.jet.doc/tasks/usingJetWithEMF.xhtml>
20. Abrahamsson, P., Warsta, J., Siponen, M.T., Ronkainen, J.: New Directions on Agile Methods: A Comparative Analysis. In: ICSE, p. 244 (2003)
21. UML superstructure specification, <http://www.omg.org/cgi-bin/doc?formal/05-07-04>

Realizing an MDA and SOA Marriage for the Development of Mobile Services

Mariano Belaunde¹ and Paolo Falcarin²

¹ Orange Labs, 8 Avenue Pierre Marzin,
22300 Lannion, France

mariano.belaunde@orange-ftgroup.com

² Politecnico di Torino, Dipartimento di Automatica e Informatica (DAUIN),
Corso Duca degli,
Abruzzi 24, I-10129, Torino (Italy)
paolo.falcarin@polito.it

Abstract. The paper presents an approach for developing composite telecommunication services running on mobile phones which takes advantage of the use of model driven techniques as well as the loose coupling paradigm in SOA. A domain-specific UML dialect named SPATEL has been developed which serves as the basis for generating applications that can be deployed in distinct terminals and servers technologies. The composite services typically combines telecommunication enablers - like SMS sending and GSM localisation - with traditional IT components accessible over the internet, such as a Yellow Page facility. This work has been conducted in the context of the IST SPICE European collaborative project.

Keywords: MDA, SOA.

1 Introduction

The emergence of SOA[1] and MDA[2] as engineering approaches to build software and systems in the IT world will have a significant impact in the way telecom infrastructures are built and telecommunication services are created. In this paper we describe our experience on combining the strengths of both paradigms to facilitate agile and portable development of telecommunication services running over different server execution technologies and mobile terminals.

1.1 Meaning of SOA in Our Context

The SOA acronym, meaning Service Oriented Architecture, is an over-used term used to describe very different situations and products. Generally speaking it refers to the mechanisms that provide service functionality remotely in a loosely coupled and distributed way over web resources.

Now, what is the impact of SOA in telecom? Telecom operators have realized the importance for the growth of their market to open, in a controlled way, the APIs to access to their telecom network resources, in a similar way as important IT players do. More specifically, various companies like Orange are now offering to third party

service providers the possibility to access elementary functionality, like SMS sending, call control and localization (called telecommunication enablers) through simple SOAP web services.

Hence, from the point of view of services developers that have to build complex composites services that make use of these telecom enablers, effective adoption of SOA principles is of major importance since it represents the ability to program telecom services without having to be necessarily experts in the telecom domain. This is a significant point, since it may imply important cost reduction in development.

1.2 Meaning of MDA in Our Context

The MDA acronym has been defined by the OMG standardization body in 2000. It means Model Driven Architecture but may represent very disparate things. In the context of service creation a *model-driven* approach consists primarily in the ability to generate large amounts of a service implementation from a high-level abstraction definition of the service, exploiting object-oriented modelling techniques – like MOF [3], UML[4] and QVT[5] standards. Apart from this technical aspect – which is mainly intended to improve productivity – another aspect of MDA relevance in telecom domain is the issue on heterogeneity of terminals and execution platforms. A telecom service for mobiles ideally would require to be developed once and yet be able to run, at the client side, in different kinds of mobile terminals and, at server side, be prepared to evolve from a technology to another (like evolving to the IMS architecture [6]).

1.3 Organization of the Paper

The following chapters will describe with some detail the SPATEL service description language, the service creation tool associated with this language and an application use case illustrating the usage of our model-driven framework in a typical *context-aware* telecom service combining IT and TELCO components.

Finally we will discuss some of the interesting issues raised by our experiments.

2 The SPATEL Service Description Language

The SPICE project has defined a *high-level* and *executable* language for describing composite telecommunication services. This formalism, named SPATEL, meaning SPICE Advanced language for Telecommunication services, can essentially, be seen as a customization of the UML language for expressing the definition of service interfaces and service composition logic that is well-suited to the telecom domain.

In contrast with most IT web services, telecom services are generally transactional, asynchronous, state-full and sometimes long-running processes. In addition a telecom service can be designed to be multi-modal – the ability to achieve a conversation using parallel interaction means like voice, text and image. Also the behavior of a telecom service needs to be often split in two parts: one running in the mobile terminal of the user – dealing with GUI aspects and local activation of telecom resources (like SMS sending) - and the other part running at server side, usually hosted by a telecom operator.

In the service developer formalism, a service is primarily described through an *external view* which provides information that is useful for service clients. The external view is basically an interface declaring a list of operations, input and output events, multimedia streams and relevant side-effects. The constraints on the service interface such as the ordering of operation invocations can be precisely defined through a contract. An important feature of SPATEL is the ability to annotate the elements of the interface (like the operations and the parameters) with semantics tags and non functional features to enable rich scenarios for service discovery and dynamic composition. Non-functional features are partitioned on the basis of categories like quality of service (QoS), charging, internationalization or resource usage. The annotation mechanism, which is similar to the approach taken in SA-WSDL approach – as it relies on pointers to pre-existing ontologies – is not detailed in this paper – which is focused on static composition.

The service developer formalism also allows representing the *internal view* of a service (white box representation) by means of a set of inter-connected service components. Two distinct views are available: an architectural view showing the list of involved components and their connections and a behavioral view consisting of state machines that define precisely the logic of an operation – an orchestration of components being a particular case. We will see some examples of the usage of this formalism in Section 4.2. The choice of state machines – rather than activity diagrams – is motivated by the idea of integrating "voice-based" dialogs in a service specification, since state machines are the most used paradigm for expressing the complexity that can be found in human-machine voice conversations. We should note however that the scope of SPATEL is much broader than the scope of traditional voice services since we have to deal with remote synchronous and asynchronous invocations, parallel threads of execution and dedicated GUIs definition at terminal level.

Concerning the definition of user interaction at client side, SPATEL provides the ability to represent potentially the usage of different GUI frameworks found in mobile world like, the very constrained J2ME [7] GUI environment or the richer GUI framework available in S60 Nokia [8] *smartphones*. This heterogeneity is enabled in the SPATEL meta-model by the fact that the coding of GUIs elements is generic: a Container contains recursively GUI Elements which in turns define GUI properties – which are name/value pairs. In addition to that GUI events can be connected to service events used within the logic of the service. Hence, thanks to this very pragmatic approach – not trying to model the whole IT world! – in our context, supporting a GUI framework means having the corresponding library of the GUI widget model components instantiated in the SPATEL design tool and having the corresponding code generator targeting the specific GUI framework.

In addition to this GUI aspect, SPATEL provides means to represent typical voice-based interactions: recognition of voice as *utterance* events, buffered construction of voice messages – which are delivered when reaching a stable state and support of specific events – like inactivity or failure recognition. The SPATEL language and execution environment inherits from previous research work done in the field of voice service modeling [9]. Hence, this aspect will not be detailed here. However it is important to point out that the combination of voice interaction modeling with GUI modeling brings the ability to model multi-modal services.

As we can notice, despite the specificity in telecom services, from a design point of view, the concepts needed by the service description language are not significantly different from those exposed by the well-know SOA standards like WSDL [10] and BPEL [11] and formalized in a more abstract way by UML [4]. The SPATEL formalism aggregates in fact well-know constructs coming from different sources (ITU-SDL [12], SA-WSDL [13], VoiceXML[14]) in order to provide the needed subset – not less, not more – that is needed for a high-level and executable formalism usable in telecom context. Among the potentially infinite design choices that UML can offer, SPATEL makes a very precise and exclusive selection like: using simple UML 1.4 state-machines instead of the full UML2 capability, not using collaborations, representing an orchestration as the behavior specification of an operation. Selectivity in the usage of the constructs offered by UML for behavior definition is necessary to have at the end an unambiguous and executable formalism that can be implemented at reasonable cost. Notice that we are not claiming that the choice we made is the only possible one. In our case the state-machine formalism had the advantage not only to be well accepted within the community of service designers but also to have well-established and robust implementations that could be used as entry points for our developments.

Technically speaking the SPATEL formalism has been defined by an EMOF meta-model and is accompanied with a UML2 profile defining the conventions for using the UML graphical notation – like adding an specific icon to represent the invocation of remote service operations. This approach, which makes the distinction between abstract syntax and concrete notation, allows using a rather compact and understandable XMI serialization format as exchange and pivot format, significantly less complex than the one associated with the complete UML2 metamodel. Also it allows attaching alternative notations to the SPATEL language, such as a dedicated textual syntax, and still relying on a common abstract representation in memory and in persistent storage.

3 The SPICE Service Creation Environment

In this section we will describe the overall architecture of the service creation environment developed by the SPICE project, as well as, some highlights on the different components that together provide the necessary ingredients for an agile development of telecommunication services in line with the formalism defined in the previous section.

3.1 Architecture

The SPATEL Developer Studio contains four main macro constituents depicted in Figure 1:

- The *SCE Service Designer* is a graphical editor to edit SPATEL service interfaces as well as to edit the logic of composite services,
- The *Service Design Repository* is a catalog of re-usable service descriptions,
- The *Analysis and Testing tools* is basically a "native" execution engine capable of interpreting almost directly the SPATEL definitions. It is used in particular to simulate and test the services before real deployment,

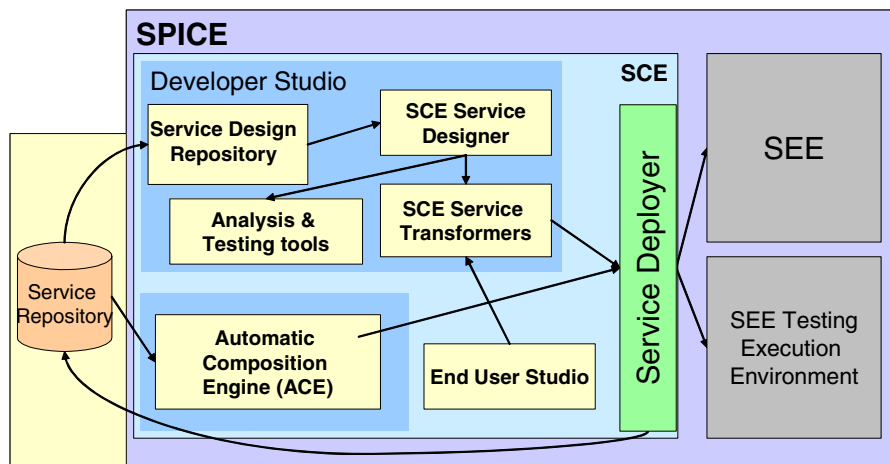


Fig. 1. Data flows between macro components in the Service Creation Environment

- The *SCE Service transformers* module represents a set of model transformers and code generators producing service implementations on top of the supported platforms.

In this figure we see a list of complementary macro components which are out of the scope of this paper: these are the Automatic Composition Engine (ACE) to create automatically SPATEL service compositions from semantically annotated SPATEL service descriptions and the End-User Studio which offers a dedicated user interface for non-professional service designers.

3.2 Implementation of the Developer Studio components

The actual implementation of the SCE Service Designer is build on top of the UML StarUML tool [16]. This component may be replaced by any other graphical facility capable of creating SPATEL XML files – the pivot format used by SCE for service definitions (see Section 2). An alternative editor based on GMF/Eclipse framework [17] is currently under construction.

The model transformations and the code generators are implemented using two alternative techniques: firstly through direct usage of the meta-modeling APIs generated from the SPATEL meta-model, and secondly using the dedicated QVT [5] model to model transformation language. Two general purpose languages are supported for the APIs: firstly Java, using the ECLIPSE/EMF [18] generated APIs for the Java language and, secondly Python, using the PYMOF [19] framework. These two APIs have in common to use the same storage format, which is the XMI format used by ECLIPSE/EMF. The SPATEL to BPEL and the SPATEL to Nokia S60 terminals are developed using the Python API whereas the WSDL to SPATEL importer is developed using the *QVT Operational* formalism by means of the SmartQVT open source tool [20]. In general QVT usage makes a transformation definition much more readable and compact. It requires however that the sources and targets are already expressed in the form of meta-models. Otherwise an extra work is needed to comply with this requirement.

The Developer Studio is connected to two other SPICE components:

- The SPICE repository, to import and export the definition of running spice services. This differs from the Design Repository, which only contains reusable fragments of SPATEL service definitions.
- The SPICE Life Cycle Manager, to deploy and activate services in the SPICE Service Execution Environment.

3.3 Process for Using the Developer Studio

The typical steps when using the Developer Studio are:

- The service designer opens the graphical front end, initializes its design model using the available SPATEL-specific menus, such as a command to initialize model contents with some pre-defined constituents. In general the use of the tailored menus is perceived as very important by practitioners since it avoids having to deal with all UML complexity,
- The service designer imports from a catalog of pre-existing service designs the service components to be re-used in the composition. This service design repository is generally linked to the SPICE service repository which provides deployment information on the list of available running services,
- The service designer defines the interface of any additional non existing service that would need to be invoked within the logic of the composite service,
- The service designer opens the pre-initialized behavior diagram to edit the state-machine expressing the logic of the composite service,
- The service designer uses the SPATEL specific menu to generate the terminal-side code and the server-side code for one or more target technologies – for instance a BPEL engine at server side and a S60 Nokia phone at terminal side,
- The service developer completes the generated code – in general, the body of the declared operations and, if necessary, the *glue code* implementing the invocation of a given service component. Manual completion of the glue code is not needed when dealing with standardized protocols like SOAP-based web services since, in that case, the invocation code is generated automatically based on the deployment descriptors,
- The service designer and service developer executes locally the service logic using the default web interface provided by the SPATEL native execution engine. This step is useful to debug the logic and the glue code,
- The service designer iterates over these steps until it obtains the required functionality,
- The service integrator deploys the service generated files into a remote server or into a specific terminal using the Service Deployer component.

4 Use Case: E-Tourism Dinner Planning Service

We describe here the definition and the implementation of a specific context-aware *Dinner Planning Service* example which has been developed as an illustration of the model-driven approach taken to develop services.

4.1 Scenario Definition

The E-tourism dinner planning scenario is as follow:

- An End User is on travel in a city. Because he does not want to waste time trying to find a good restaurant for his dinner he will delegate this task to a specialized dinner planning service. In the morning, he sends an SMS to the Service dinner planning requesting for finding a "recommended" restaurant at 20:00 near the location where he will at that time, and respecting some criteria (type of food),
- At dinner time (20:00) the Service locate a suitable restaurant list based on the end user geographic position,
- The Service sends a message to the End User containing the list of restaurants located in the surroundings including the contact points for reservation,
- The End User activates a call to the restaurant of choice using the restaurant contact point information.

The components that need to be in place for this scenario are:

- A Personal Agenda, to store from the user its willingness to be notified at dinner time,
- A Localization service, which will found the user's location relying on GSM network information,
- A SMS or Instant Messaging enabler to notify the user when the list of restaurants is found,
- A Yellow Pages service to found the restaurants near the location of the user,
- A Third Party Call component to activate the call to the selected restaurant.

The figure below shows the interaction between the different composed components and the orchestration engine:

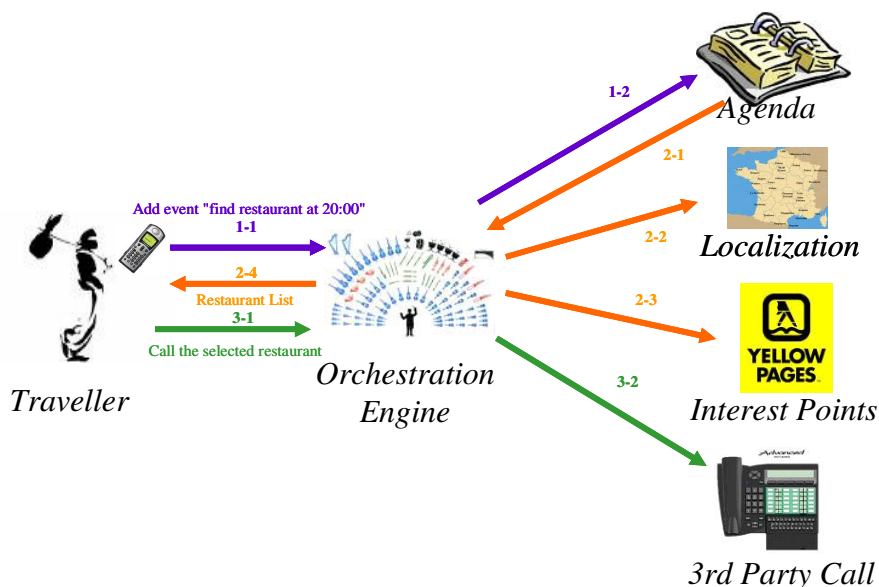


Fig. 2. Dinner planning scenario overview

From the point of view of the orchestrator, the scenario has three temporal phases:

- The orchestration engine receives the user request (1-1) and registers the event in the personal agenda (1-2),
- At dinner time, the orchestrator receives the reminder from the personal agenda (2.1) which invokes the localization services (2.2) to obtain the location information of the traveler. Then it request the interest points of the yellow pages services (2.3), collects the responses and sends the results to the traveler (2.4).
- Finally, if the user selects a restaurant, the orchestrator receives the request (3.1) and invokes the 3rd party call service to establish the communication.

4.2 Design of the Composite Service

In our experiment the SPATEL language, described in Section 2, has been used to develop the dinner planning service. In practice, following the SPATEL language philosophy this means:

- Declaring the interfaces for all the invoked components (Agenda, Localization, Yellow Pages, 3rd Party Call),
- Declaring the composite component – with a single 'orchestrate' operation – and defining the logic of this operation through a state machine.

All of the components to invoke already exist in some form. The Localization component is provided by Orange in the form of a web service, the Interest Points restaurant inspection can be obtained using an HTTP GET request on the French "Pages Jaunes" web site (after some filtering and parsing of the HTML output), the 3rd Party Call is another web service, and the agenda on line web component role can alternatively be played by Google Calendar application of a specific Orange Personal Calendar service.

So at this level, various questions arise, like:

- When a web service, is available should I directly derive the SPATEL interface from the WSDL interface or should I try to make some filtering to simplify it?
- When we have more than one candidate, should I try to define an interface that works for all the available possibilities?

Taking the WSDL file "as is" – through the WSDL to SPATEL importer – could be a comfortable solution but has some drawbacks. For instance, it could have an impact in the complexity of service logic definition, due to the fact that additional parameters - not really relevant to the designed composite service - may need to be constructed and passed anyway to have a valid service invocation.

Concerning the second issue, abstracting a common interface implies that there is the possibility to make the adaptation somewhere – maybe at deployment, when generating code from the model of the logic, or, at runtime, when executing the service through an intermediate object that performs the argument conversion. The best choice really depends on the target execution technology. When using the BPEL engine we tend to favor the first solution relying on code generator intelligence to perform the interface adaptation, since adding an intermediate web service would be costly. In the case of the Spatel Engine, for which an intermediate *local* proxy class is always generated, the second solution is much more convenient.

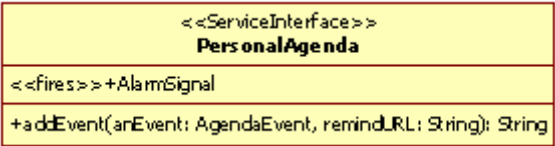


Fig. 3. Interface of the Personal Agenda component

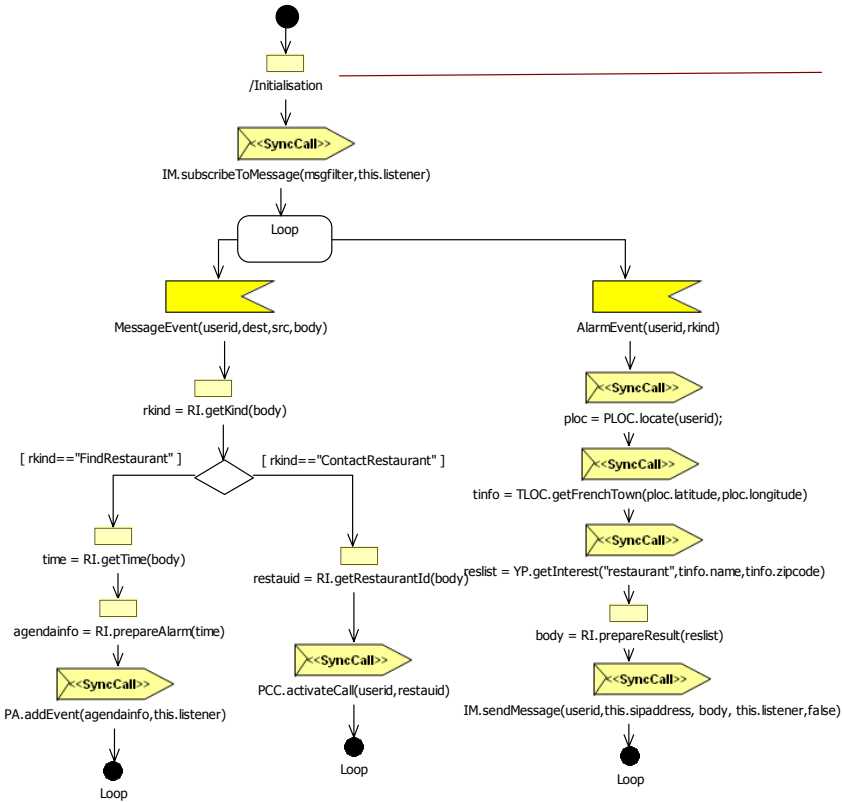


Fig. 4. Logic of the dinner planning service orchestration

In the case of the Dinner Planning service we followed the strategy of abstracting and simplifying as much as possible the interfaces of the invoked services. At the end, this had some implications in the design of the Service Repository: a unique SOAP web service may be associated to one or more registered SPATEL interfaces.

The figure below shows the interface of the Agenda component which abstracts a piece of functionality common to the Google Calendar and the Orange Personal Agenda component.

The following figure shows the modeling of the logic of the orchestration operation: we see the three threads of execution as described in the scenario definition

(in Section 5.1). On the left, we have the reception of the user initial request, on the right the treatment of the event triggered at dinner time and in the middle the final phone call. Note that this state machine uses the new UML2 transition centric view - in fact taken from ITU SDL - in which the list of actions executed during the triggering of a transition are explicitly represented as rectangles. In this diagram a specific icon is used to denote a remote service invocation, similar to an asynchronous signal sending symbol in UML. For the comprehension of this diagram, we should also mention that a Service Call in the SPATEL formalism is not an action but a State node, which gives the possibility for defining explicit exceptions transitions in case of invocation errors - overriding the default mechanism for handling errors.

4.3 Implementation and Deployment of the Composite Service

We generate two alternative implementations: one on top of the BPEL engine and the other on top of the SPATEL engine. In our development process, the implementation is the engineering phase where code generators are invoked and code completion is done when necessary. Because the state machines used in SPATEL have unambiguous execution semantics, the code corresponding to the state machine was completely generated. The part that required some manual code completion was the code related to the realization of "non standard" remote service operation calls, like the one performed to connect to Google Calendar [15] since this follows a proprietary protocol. Also all intermediate computations - like the formatting of the message containing the list of restaurants, which were modeled as invocations of local black-boxes operation calls - need to be completed, since only the skeletons were generated. The percentage of generated code in our dinner planning application was 80%. Notice however that in situations where all invoked components represent already existing components - registered as implemented components in the SPICE service repository - this generation factor may be of 100%! The richer is the catalogue of services, best are the chances to produce composite services without any code writing.

The client part for the Nokia N80 phone was generated using a simple description of the GUI in the form of Python code directly interpreted by the phone. The figure below represents the screen to activate the service.

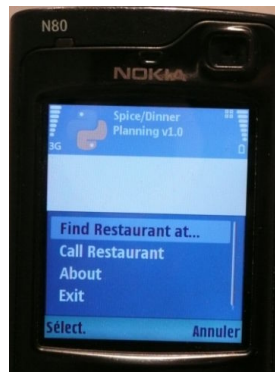


Fig. 5. Activation menu for the dinner planning service

5 Discussion

In this section, we will discuss an interesting question which concerns the legitimacy for using graphical notations like SPATEL for defining service logic. This is a controversial question and we will try to respond on the basis of our experience.

5.1 Does It Make Sense to Define Service Logic by Means of Graphic Models?

The target users of the SPATEL graphical language are professional service architects and service developers. The first population of users will probably not have to deal with the implementation tasks. However, for the second category of users, we can legitimately ask whether it make sense to develop the logic of a service using a graphical notation instead of using directly a general purpose programming language.

Our, experiments yields us to the observation that, for sure, for a programmer, using a graphical notation is much more time expensive than direct coding. However, if the time for providing an implementation is not a dramatic issue, there are clear advantages to make use of graphical notation to develop service logic that have good quality:

- Firstly, in formalisms like SPATEL, the designer is free to decide where to put the border between "graphical design" and "textual coding" of service logic: any intensive computation can be encapsulated by means of a black-box local operation. Also, some components may be completely implemented using opaque code and still have a well-defined SPATEL interface to allow its reference in other services. This emphasizes the fact that the choice between graphics and text is not black or white. The good balance between both is the responsibility of the service writer.
- Use of graphical notation helps in defining a "clean" logic, which can be understood by others, and hence facilitates the design of a logic that can be valid in different platforms. Quality of abstraction is an important feature for those that want to apply model-driven transformations to create multiple implementations from the same specification.

We believe the problem of the border between design and code will always exist. However we notice that model-driven technology is effectively pushing in the way of making more and more design and less coding and this is particularly true in the domain of service development.

6 Conclusions

Service composition has become a hot topic for all telecommunication players. The ability for professionals and, even more for end users, to compose efficiently running telecom components, depends a lot on the availability of tools capable of hiding the complexity to access the telecommunication network resources. Many initiatives are currently launched in the telecom arena to try to solve the complexity of distribution and heterogeneity, especially now that the operators tend to open their access to their network resources.

The SPICE is contributing to this challenge by developing a set of powerful inter-related tools which are integrated within its Service Creation Environment. The combined use of SOA and MDA is a distinguishing characteristic of the work conducted by this project.

In this paper we have limited the scope to the case of the service creation for developers. We presented a subset of UML named SPATEL that is designed to create services that are easily portable to different platforms at server and terminal side.

On the language side, future work will focus on a possible alignment of our formalism with the UPMS specification [21], which is still under construction at the OMG. In this respect, we tend to perceive SPATEL as a specialization of UPMS where only simple UML interfaces are used (no explicit notion of required interfaces) and where a composite service is represented by a default "service participant" containing the operation behaviours in the form of either an opaque implementation or a state-machine.

On the platform side, future work will focus on targeting new telecom oriented platforms like the Android environment from Google [22] or FlexLite from Adobe Technologies [23]. An interesting point for the future will be to see if the heterogeneity in mobile terminals will continue to be an issue with the emergence of a limited number of de facto Web 2.0 standards technologies in the mobile world.

References

1. OASIS, OASIS Reference Model for Service Oriented Architecture V 1.0 (August 2, 2006) <http://www.oasis-open.org/committees/download.php/19679/soa-rm-cs.pdf>
2. OMG, Model Driven Architecture, document ormsc/2000-11-05, (November 2000), <http://www.omg.org/mda/>
3. MG, Meta Object Facility V2.0, document formal/2006-01-01 (January 2006), <http://www.omg.org/spec/MOF/2.0>
4. OMG, Unified Modeling Language V 2.1.2, document: formal/2007-11-04 (November 2007), <http://www.omg.org/spec/UML/2.1.2/>
5. OMG, MOF 2.0 Query/Views and Transformations, document ptc/07-07-07 (July 2007), <http://www.omg.org/cgi-bin/doc?ptc/2007-07-07>
6. 3GPP, Service Requirements for the IP Multimedia System, Core Network Subsystem, release 5, document 3GPP TS 22.228 V5.6.0 (2002-2006)
7. Sun, Java 2 Micro Edition, Connected Limited Device Configuration 1.0, JSR 30, <http://java.sun.com/javame/index.jsp>
8. Nokia, Operating System Symbian S60, <http://www.s60.com>
9. Belaunde, M., Presso, J.M.: Vision for an industrial application of MDD in the Telecommunications Industry. In: ECMDA 2005 Conference. Springer, Heidelberg (2005)
10. W3C, Web Service Definition Language (WSDL), document (March 2001), <http://www.w3.org/TR/wsdl>
11. OASIS, Web Services Business Process Execution Language Version 2.0 (BPEL) (April 11, 2007), www.oasis-open.org/committees/wsbpel/
12. ITU-T, Specification Definition Language (SDL), <http://www.itu.int/ITU-T>
13. W3C: Semantic Annotations for WSDL and XML Schema, W3C Recommendation (August 28, 2007), www.w3.org/2002/ws/sawSDL/
14. W3C/VoiceXML Forum: Voice Extensible Markup Language, <http://www.w3c.org/TR/2007/REC-voicexml21-20070619/> www.voicexml.org/
15. Tool Google Calendar, <http://www.google.com/calendar>

16. Tool StarUML, <http://www.staruml.org>
17. Tool Graphical Modeling Framework (GMF), <http://www.eclipse.org/gmf>
18. Tool Eclipse Metamodeling Framework (EMF), <http://www.eclipse.org/emf>
19. Tool: Python Meta Object Facility framework (PYMOF) distributed with SmartQVT tool, <http://smartqvt.elibel.tm.fr/>
20. Tool SmartQVT, <http://smartqvt.elibel.tm.fr/>
21. OMG, Uml Profile And Metamodel for Services RFP (September 2009), <http://www.omg.org/cgi-bin/doc?soa/06-09-09>
22. Tool : Google Android <http://code.google.com/android/>
23. Tool: Adobe Technologies FlexLite, <http://www.adobe.com/fr/products/flex/>
24. Venezia, C., Falcarin, P.: Communication Web Services Composition and Integration. In: Proceedings of International Conference on Web Services (ICWS 2006), Chicago, USA, pp. 523–530. IEEE press, Los Alamitos (2006)

A Survey about the Intent to Use Visual Defect Annotations for Software Models

Jörg Rech¹ and Axel Priestersbach²

¹ Fraunhofer IESE, Fraunhofer Platz 1, 67663 Kaiserslautern, Germany
+49 (0) 631 6800 2210,
Joerg.Rech@iese.fraunhofer.de

² SAP Research, Vincenz-Prießnitz-Str. 1, 76131 Karlsruhe, Germany
+49 (0) 6227-752525,
Axel.Priestersbach@sap.com

Abstract. Today, many practitioners have consolidated their experience with software models in collections of design flaws, smells, antipatterns, or guidelines that have a negative impact on quality aspects (such as maintainability). Besides these quality defects, many compilability errors or conformance warnings might occur in a software design. Programming IDEs typically present problems regarding compilability in or near the code (e.g., icons at the line or underlining in the code). Modeling IDEs in MDSD follow a visual paradigm and need a similar mechanism for presenting problems in a clear, consistent, and familiar way. In this paper, we present different visualization concepts for visualizing quality defects and other problems in software models. These concepts use different dimensions such as color, size, or icons to present this information to the user. We used a survey to explore the opinions held by practitioners showing that 89.9% want to be informed about potential defects and prefer icon-, view- and underscore-based concepts to other types of concepts.

Keywords: Visual Annotations, Software Models, Intelligent Assistance, Quality Defects, Software Diagnostics, MDSD.

1 Introduction

Model-driven software development (MDSD) drastically alters the software development process, which is characterized by a high degree of innovation and productivity. MDSD focuses on the idea of constructing software systems by designing visual models that are translated into executable software systems by generators. These characteristics enable designers to deliver product releases within much shorter periods of time compared to the traditional development methods. In theory, this process makes it unnecessary to worry about an executable system's quality, as it is "optimized" by the generators.

However, just as in current software programming, people make mistakes that result in defects of the software model. These defects might prevent the compilation/transformation of the model, deteriorate its quality aspects such as its

maintainability, or violate conformance to a modeling standard. In programming errors and warnings are usually presented to the programmer in the textual editor by means of problematic parts being underlined or marked with little icons. In MDSD, problems are typically not annotated directly in the visual models.

In this paper, we present several concepts for annotating software models in order to provide architects, designers, and modelers with additional information about problems regarding the compilability, quality, or conformance of a software model. Our primary research goal was to identify the acceptance of our concepts by practitioners and explore their intent to use these visualization concepts.

After presenting related work in section 1, we describe details of the annotation concepts in section 2. The instrument for the evaluation of visual mockups, used to evaluate the acceptance of our annotation concepts, as well as the findings of our survey is described in section 3. Finally, we conclude our contribution in section 4

2 Related Work

In order to inform the modeler about the quality defects in his software model (e.g., the PIM), we need to annotate the visual diagrams presented to him. However, annotating UML with information about quality defects is not a straightforward task. UML is intended to describe the structural (and, with Action Semantics, the behavioral) elements of a software system. Nevertheless, in UML (2.1) [13] several mechanisms exist to store additional (non-standard) information in the software model. However the additional information are either not shown in the diagrams (e.g., UML annotations) or would flood the diagrams (e.g., UML comments).

2.1 Defects in Programming IDEs

In programming IDEs, icons are typically used to pinpoint problems such as compiler warnings or errors. For example, the Eclipse IDE (V3.3) [5] uses markers to annotate problems in the source code and icons are anchored directly beside the respective line, while underlines are used to pinpoint the exact position. A problem view is used to list all problems in all projects and decorators are used to annotate problems in a class or other files shown in the project file tree. Netbeans (V5.5) [7] also presents defects as icons in the source code directly beside the respective line and underlines the exact position in the text. In Visual Studio .Net 2003 (V7.1) [15], defects are listed in the Task List after the build process is started.

Another source for error or defect visualization are tools used for bug tracing and error detection. A couple of such tools exist for the Java language. Most of them are extensions to the Eclipse environment. During our survey, we looked at the most commonly used tools: Findbugs [6], PMD [9] and CheckStyle [2]. All tools are based on Eclipse and use the features Eclipse provides, such as the “problem view”, to report problems to the user. While these annotations are very useful in textual IDEs, they are not sufficient for visual IDEs in MDSD. Here we need to annotate elements in 2D graphs with multiple defects of various severity, priority, and effects.

2.2 Defects in Modeling IDEs

In software modeling, the visual annotation of defects in software models is scarcely explored. Current modeling tools can be classified as either being based on the Eclipse framework or being standalone tool. Those tools integrated into Eclipse usually use the known Eclipse features to visualize errors:

- The *problem view* lists errors and warnings from multiple sources and aggregates them into one list. The list usually supports icons to indicate the level of seriousness, such as error, warning, or info.
- The *outline view* is a representation of the model tree and usually contains information on errors, which are indicated by mini-icons (i.e., so-called decorators).
- Errors in the *textual editor* area are usually indicated by an icon directly beside the line of code and the text block containing the error is highlighted by underscores. To facilitate navigation, errors are marked right besides the scrollbar.

Typical examples of model editors built on top of the Eclipse functionalities are Topcased and OmondoUML. Topcased [12] displays errors in the UML models as tasks in the Eclipse Problem List and also little icons directly in the diagram as well as the tree view of the UML model. The icons are rather small. Only one error type is used. Errors in hidden parts of the tree view are not shown in collapsed nodes. One example that combines all methods is OmondoUML [8] that is also based on Eclipse and used within Java projects. It attaches Java coding directly to the model. Errors in the Java coding are shown in multiple locations such as the UML diagram, the code, the Eclipse Problem list, and the project (resp. package) tree.

Examples of tools that are not based on Eclipse are Poseidon [10] and Enterprise Architect [4]. Poseidon [10] is the commercial variant of ArgoUML [1], which presents some errors in the UML diagram using icons and most others in a list. Enterprise Architect [4] supports model validation and detects errors and warnings found in the model. They are visualized as lists similar to the Eclipse Problem list. However, these tools do not use differentiating icons in the error list or in the diagram.

2.3 Safety Signs and Defect Pictograms

Beyond our own domain people in the field of technical documentation are using icons, signs, and pictograms to warn about dangerous substances or situations. ISO 3864-2 2002 and ANSI Z535.2-2002 are standards for environmental & facility safety signs that includes plain warning symbols, detailed message panels, and short header sections (see Figure 1a). It is intended to establish “the safety identification colors and design principles for safety signs to be used in workplaces and in public areas for the purpose of accident prevention, fire protection, health hazard information and emergency evacuation”. Other standards include ISO 7010 (see Figure 1c), IEC 61310, ISO 16069, IEC/TR 60878, (see Figure 1b), ISO 7000, or IEC 60417. Similar to the passenger/pedestrian symbols developed in the 1970s by the AIGA and the U.S. Department of Transportation (DOT), they represent uniform and consistent visual languages.

However, while these pictograms can be used in the context of MDSD to annotate defects in software models, they are not tailored to represent different types, severities, priorities, or effects of these defects.

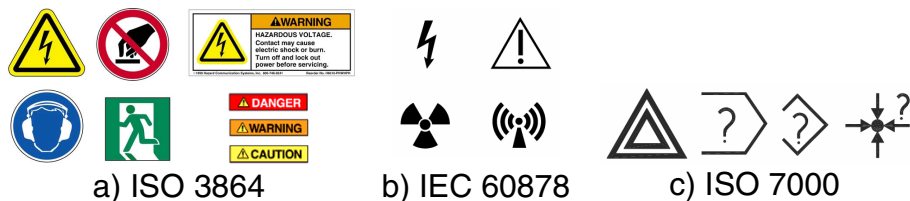


Fig. 1. Warning Symbols and Pictograms

3 Visualization Concepts

For the annotation concepts, we used an exemplary UML diagram contaminated with multiple defects. This example was also used in the survey as the guiding scenario. As presented in Figure 2, the UML class diagram contains a package (Opportunity) and two class declarations (Opportunity and SalesForecast). The class Opportunity is meant to represent an opportunity for a sale (e.g., during project acquisition) and SalesForecast the prognosis / forecast about the chances and benefits of this opportunity.

Since the focus of this survey was on the defect annotation of the software models, we introduced five defects on different levels that are marked with numbers in the following diagram:

1. **Relation defect:** Defect in the relations between the two classes. In this case, a circular inheritance was introduced.
2. **Attribute defect:** Defect in the attributes of the class SalesForecast. In this case, one identifier is specified twice.
3. **Method defect:** A defect in the method declaration of the class SalesForecast. In this case, too many parameters (i.e., the code smell "Long parameter list").
4. **Class defect:** Defect in the Opportunity class itself: The class has too few methods and attributes (i.e., the code smell "Lazy Class").
5. **Diagram defect:** Defect in the diagram: The diagram has too few elements and might be superfluous.

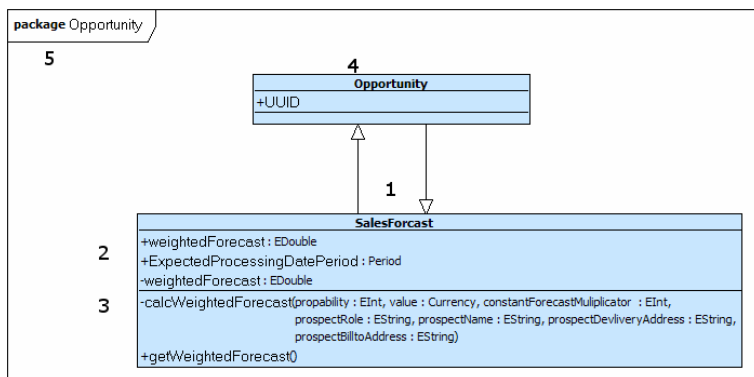


Fig. 2. Scenario Mock-up

We collected several kinds of annotations during a brainstorming session. We identified 11 techniques applicable to two-dimensional software models listed in Table 1. The first five (Icons to Aura) including the last one (Views) were selected for evaluation as they appeared to be the most promising. The concepts are characterized, as to whether they can be applied in entities (e.g., boxes for classes or packages), relations between entities, the connectors of relations (e.g., the diamond shaped ends of an aggregation), or additional notes or comments. Furthermore, the positioning of the annotation is differentiated in the body (e.g., within a box or connector), the line, the frame (e.g., borderline of a box or connector), and the aura (e.g., directly outside the box or connector).

Table. 1. Visualization Concepts and Affected Elements

	Entities			Relations		Connectors			Notes		
	Body	Frame	Aura	Line	Aura	Body	Frame	Aura	Entity	Relation	Connect.
Icons	●	-	●	-	●	○	-	●	●	●	●
Color	●	●	●	●	●	●	●	●	●	●	●
Bold	○	●	-	●	-	-	●	-	●	●	○
Dashed	-	●	-	●	-	-	●	-	○	●	●
Aura	●	●	-	●	-	●	●	-	●	●	●
Form	○	●	●	●	●	○	●	●	○	●	●
Size	○	●	●	●	●	○	●	●	○	●	●
Pattern	●	○	●	○	●	●	○	●	●	○	●
Opaque	○	●	○	●	-	○	●	-	●	●	●
Tilting	○	●	○	-	○	-	●	○	○	-	○
Views	-	-	-	-	-	-	-	-	-	-	-

In order to annotate software models in MDSD with information on defects, we developed the previously described concepts that should visually present defects. As we developed these concepts, we identified the following constraints:

- They should be *integrated into the UML* (Version 2.1) and not change the meaning and standard representation of the language’s elements (e.g., by changing the form of class-boxes).
- They should be easily *integratable into today’s UML modeling environments* (i.e., this excludes animated or 3D visualizations).
- They should *pinpoint the location* of the defect as exactly as possible.
- They should enable *differentiating between different defects* (e.g., based on type).
- They should enable the annotation of one modeling element with *multiple defects*.
- They should *not distract modeler* from his work (i.e., excluded large annotations).

Finally, the annotations should be presented to today’s software developers and modelers in a familiar way.

The icon-based concept is centered around the idea of representing every defect by a distinct icon positioned at a package, class, method, attribute, or relation in a UML model. As depicted in Figure 3, the five previously mentioned defects are positioned very close to the defective element. However, only one icon per element can be attached (except for packages/diagrams, classes, and possibly (long) relations) and, if more defects were diagnosed, they have to be hidden (i.e., stacked) below the first one. The amount of icons positioned at packages/diagrams, classes, and relations is basically constrained by their individual size.

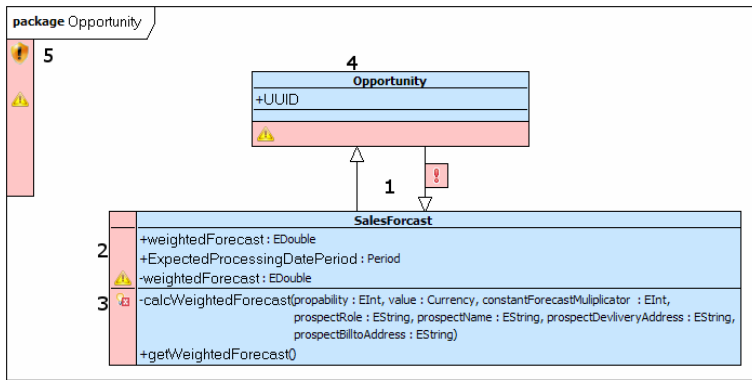


Fig. 3. The Icon-based Concept

We varied the icons in order to represent different defects, since in reality, several different kinds of defects (e.g., compiler errors, maintainability defects, security flaws, etc.) would be shown. However, as several hundreds of these defects are known [11] it is probably not possible to represent every individual defect by one distinct icon (esp. considering the size limitations of 16 x 16 pixels). Nevertheless, groups of defects regarding one specific quality aspect (cf. ISO 9126) might be represented by one icon.

The color-based concept is centered around the idea of representing every defect with a distinct color for a package, class, method, attribute, or relation in a UML model. Here the five previously mentioned defects are indicated by different colors.

As with icons, the concept allows only one color to be assigned for each individual element and therefore only one defect/error type. Multiple colors can represent several different kinds of defects (e.g., compiler errors, maintainability defects, security flaws, etc.), but if multiple defects need to be assigned to one model element, only the color of the defect with the highest priority can be presented.

However, as color can and is used in UML tools to distinguish entities such as classes, this concept has to be handled with care. Furthermore, if color is used it should affect the readability of the diagram (e.g., a red text on a red background would be hard to read).

Boldness-Based Concept

The boldness-based concept is very similar to the color-based concept but represents defects in boldface text for package, class, method, attribute names and with thicker/bold lines for relations and boxes in the UML model. The concept is expected to be less intrusive or distracting, however, the drawback is that the visualization cannot distinguish different types of defects.

Underscore-Based Concept

The underscore-based concept adapts a concept for error visualization from code editors as well as from spellcheckers in word processors. Since VIDE visual syntax goes beyond textual, this concept has been extended to diagrams where needed. While package, class, method, and attribute names are underlined, relations are *overlaid* with a dashed line. The concept may be extended to other diagrams and connectors as needed. To distinguish different defect types, different colors can be used similar to the color-based concept and as depicted in Figure 4.

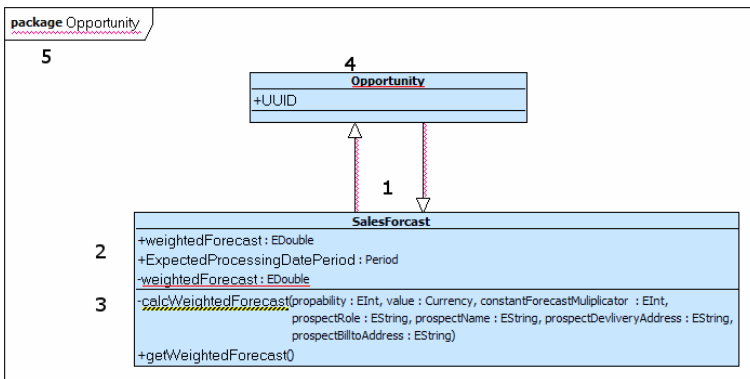


Fig. 4. The Underscore-based Concept

Aura-Based Concept

The aura-based concept represents a fusion of the underline-based concept with the color-based concept in order to maintain consistency of the annotated diagram. In this concept, diagrams, entities, relations, and text are enriched with a colored aura or halo. The aura-based concept surrounds all elements (package, class, method, attribute, and relation) with defects/errors in a consistent way.

The Views Concept

A separate view that lists all errors, defects, warnings etc. is the most common and accepted concept. Today, the majority of development, bug tracing, and diagram modeling tools feature a separate view that shows identified defects. The views are organized as simple lists that often support (hierarchical) categories and sorting.

Icons are often used to facilitate understanding. These icons should be the same (also semantically) as those used in other views (i.e., a diagram or the coding/textual view.).

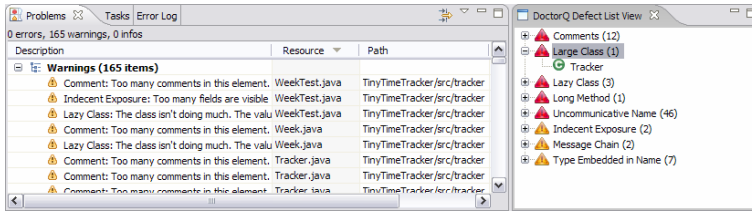


Fig. 5. Concept Using Views

4 Evaluation

In order to evaluate these annotation concepts for architectural models, we conducted a survey using an electronic questionnaire. The survey was aimed at eliciting which annotation concept practitioners prefer and when, what kind, why, and where they prefer the annotations.

We conducted the survey between 24 September and 8 October 2007. The main target groups were architects, designers, as well as programmers and testers in software organizations who are involved in daily software development activities. Our respondents consisted of a total of 292 individuals, of whom 78 completely finalized the questionnaire – 48.3% from SMEs and 51.7% from large enterprises.

To develop the survey pages and make them available on the Internet, a commercial tool called OPST from the company Globalpark (<http://www.globalpark.de/>) was used. The questionnaire was designed using multiple choice questions (mostly based on a 7-point semantic differential or five-point Likert scale) wherever possible, as these are more likely to be answered, and it is easy to statistically analyze the answers. To allow unexpected answers, most concepts had an open question with some extra space for comments.

Figure 6 summarizes the respondent profile of our survey. The respondents had 6-10 years of experience on average and consisted of 54.2% architects, 18.6% developers, and only 27.1% other (of which 1.5% stated to be project managers).

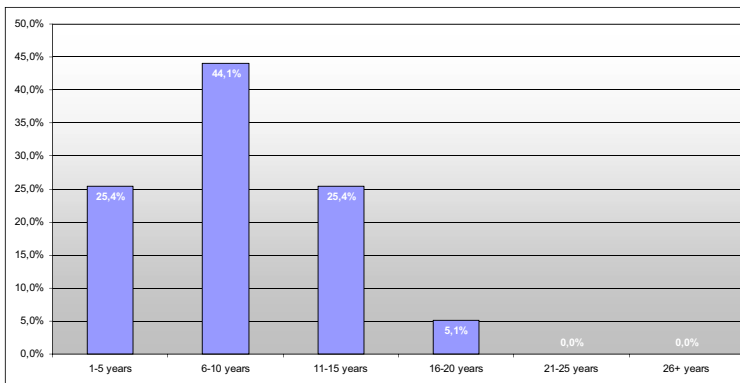


Fig. 6. Experience with Software Modeling

Furthermore, 67.9% of their employers had been using modeling techniques for over 2 years for their commercial software, 66.1% for their internal software, and 50.9% were applying the model-driven approach (i.e., code generation).

The respondent profile obtained met our prior expectations, considering the basic user group of assistance in software engineering tools. Non-management employees and project managers are the group that is supposed to have the most contact with tools in this domain.

4.1 Findings

The following results are extracted from the answers to the survey and are provided in graphical format for reasons of brevity. The main feedback on the evaluated visualization concepts is depicted in Figure 7. To compare the concepts we defined eight factors that are targeted to explore the intent to use a visualization concept. We asked the following questions using a semantic differential between 7 and 1, with 4 representing the neutral center:

- *Useful vs. useless*: To identify if the participants perceive the visual concept as useful or not.
- *Unobtrusive vs. distracting*: To identify if the participants perceive the visual concept as distracting or not.
- *Appealing vs. repelling*: To identify if the participants perceive the visual concept as appealing or not.

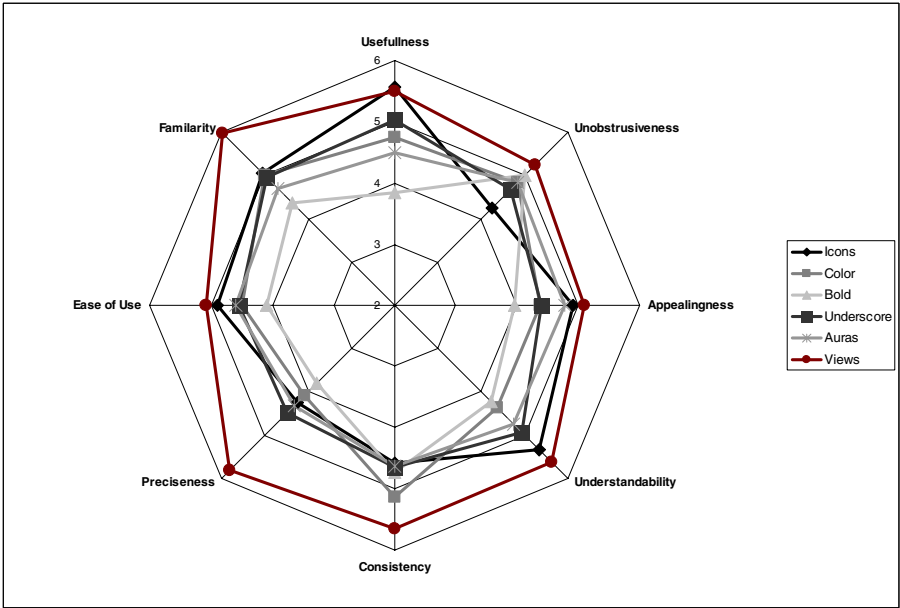


Fig. 7. Kiviati Graph Overview

- *Understandable vs. incomprehensible*: To identify if the participants perceive the visual concept as understandable or not.
- *Consistent vs. inconsistent*: To identify if the participants perceive the visual concept as consistent or not.
- *Precise vs. imprecise*: To identify if the participants perceive the visual concept as precise or not.
- *Easy to use vs. hard to use*: To identify if the participants perceive the visual concept as easy to use or not.
- *Familiar vs. strange*: To identify if the participants perceive the visual concept as familiar or not.

In order to compare all annotation concepts and rank them, we mapped the scores from all questions onto one value. As shown in Figure 8 the concepts with the highest overall scores are the views, icons, and underscore concepts. While no concept was found to be inadequate, the participants were almost undecided regarding the bold-based concept. The color- and aura-based concepts were slightly accepted and are only marginally behind the underscore-based concept.

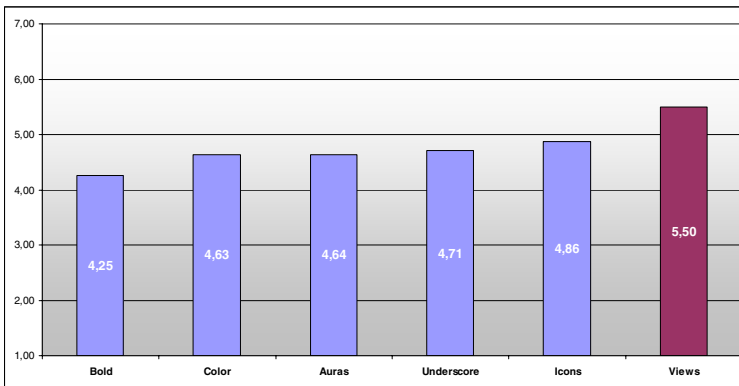


Fig. 8. Average Score for Concepts

The general preference of visual in-model presentation concepts (plus additional ones such as form or opaqueness) is shown in Figure 9 question was: “Assuming I had access to a modeling system (e.g., an UML tool) with a defect annotation extension, I would use it if it presented defects with the following types of annotations”. While the participants had no picture that described the concept the results seem to indicate that they are interested in the “Pattern (e.g., the background of a box)” concept. Concepts based on size (e.g., larger boxes and font sizes) or forms (e.g., deformed boxes) seem to be very undesirable.

4.2 Discussion

In this paper, we present different visualization concepts and used a survey to explore the practitioners’ opinions. In this section, we discuss the strengths and weaknesses of

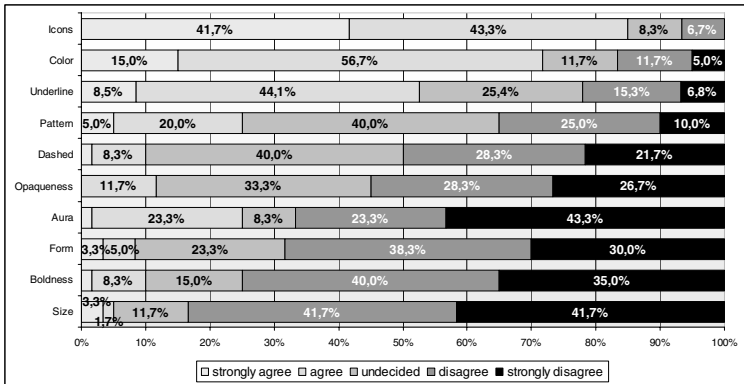


Fig. 9. General Preference of Concepts

our work. Beside the data presented in section 3, we have observed that the participants were very interested in the matter and gave long comments (3-4 sentences) to each concept.

Strengths

The survey described above has enabled us to undertake a transparent collection of opinions regarding the perceived usefulness and acceptance based on the technology acceptance model (TAM). While the original TAM (as well as UTAUT [14]) questions are not aimed at evaluating mockups, Davis and Venkatesh analyzed TAM for the user acceptance testing of pre-prototypes/mockups [3].

This survey was constructed to record information on visualization concepts for quality defects in software models. In retrospective, this approach and our implementation had the following strong points:

- The survey is *replicable* due to the approach described, the search terms used, and the selection process.
- It is *transferable* to other, new visualization concepts and can be used to compare them to the concepts we selected for the review.

Whether the results really show the intended use, i.e., whether the results of Davis and Venkatesh will hold, has to be investigated in the future.

Weaknesses & Threats to Validity

However, besides these strong points, we are aware that there may be weak points to our survey. From our point of view, it had the following weak points:

- The participants only evaluated a static representation of the visualization concept and could not “work” with it. However, we follow the research done by Davis and Venkatesh [3] that the evaluation of mockups can be used to judge the real usage behavior.
- We should have used more “obviously” bad concepts in order to find more explicit differences between the concepts. However, due to the constraints regarding the time participants are willing to invest, we assume that only few (i.e., 2-4 more for a

total of 8-10) concepts could be added, as one concept requires approx. 3-5 minutes to evaluate.

Finally, it is unclear if the results of the survey are truly representative, as no information on the basic population in the field of MDSD is documented. However, as we have elicited answers from people in organizations of various sizes and in different domains, as well as from different experience levels we assume that there was no large systematic error.

5 Conclusion

The findings of this survey provide a general characterization of the preferences of practitioners regarding the annotation of software models in MDSD. The findings helped us to identify the most promising candidate for quality defect annotation and might be used as a starting point for people interested in the development of intelligent assistance systems and annotation languages.

The survey results provided the following observations about visual annotations of software models as perceived by the participants:

- Almost all participants (89.8%) want to be informed about defects in their software models.
- Of the visual concepts presented, the icons-based concept was preferred above all others.
- Views that present a list of defects are clearly preferred by many people and should not be replaced by purely visual concepts.
- The most useful, appealing, understandable, easy to use, and familiar concept seems to be icons.
- The most unobtrusive concept seems to be bold.
- The most consistent concept seems to be color.
- The most precise concept seems to be underscore.

As MDSD is becoming more and more getting productive and enables software engineers to visually develop software systems, we expect to see our or similar concepts integrated into visual IDEs. Therefore, we are currently working on the implementation of defect annotations into an open source IDE for MDSD and are constructing a visual language for the differentiation of individual defects.

References

1. ArgoUML, ArgoUML IDE (last accessed on 27 November 2007), [\url{http://argouml.tigris.org/}](http://argouml.tigris.org/)
2. Checkstyle, Checkstyle Defect Detector (last accessed on 27 November 2007), [\url{http://checkstyle.sourceforge.net/}](http://checkstyle.sourceforge.net/)
3. Davis, F.D., Venkatesh, V.: Toward preprototype user acceptance testing of new information systems: implications for software project management T2 - Engineering Management. IEEE Transactions on, Engineering Management~51(1), 31-46 (2004)

4. EA, Sparx Enterprise Architect IDE (last accessed on 27 November 2007), [\url{http://www.sparxsystems.eu/default.asp?nav=3x6\&lid=32}](http://www.sparxsystems.eu/default.asp?nav=3x6\&lid=32).
5. Eclipse, Eclipse IDE (last accessed on 27 November 2007) [\url{http://www.eclipse.org/}](http://www.eclipse.org/)
6. Findbugs, Findbugs Defect Detector (last accessed on 27 November 2007), [\url{http://findbugs.sourceforge.net/}](http://findbugs.sourceforge.net/)
7. Netbeans, Sun Netbeans IDE (last accessed on 27 November 2007), [\url{http://www.netbeans.org/}](http://www.netbeans.org/)
8. Omondo, Omondo UML IDE (last accessed on 27 November 2007), [\url{http://www.omondo.com/}](http://www.omondo.com/)
9. PMD, PMD Defect Detector (last accessed on 27 November 2007), [\url{http://pmd.sourceforge.net/}](http://pmd.sourceforge.net/)
10. Poseidon, Gentleware Poseidon IDE (last accessed on 27 November 2007), [\url{http://www.gentleware.com/products.html}](http://www.gentleware.com/products.html)
11. Rech, J., Spriestersbach, A.: Quality Defects in Model-driven Software Development, Deliverable, Fraunhofer Institute for Experimental Software Engineering (IESE), Kaiserslautern, D4.1 (2007)
12. TopCased, Topcased IDE (last accessed on 27 November 2007) [\url{http://www.topcased.org/}](http://www.topcased.org/)
13. UML-2.1.1, Unified Modeling Language (UML), version 2.1.1, Object Management Group, Inc (OMG), Needham, MA, USA (2007)
14. Venkatesh, V., Morris, M.G., Davis, G.B., Davis, F.D.: User acceptance of information technology: Toward a unified view. *MIS Quarterly*~27(3), 425--478 (2003)
15. Visual-Studio, Microsoft Visual Studio IDE (last accessed on 27 November 2007), [\url{http://msdn2.microsoft.com/en-us/vstudio/default.aspx}](http://msdn2.microsoft.com/en-us/vstudio/default.aspx)

MDA-Based Methodologies: An Analytical Survey

Mohsen Asadi and Raman Ramsin

Department of Computer Engineering,
Sharif University of Technology, Tehran, Iran
mohsenasadi@mehr.sharif.edu, ramsin@sharif.edu

Abstract. Model-Driven Development (MDD) has become a familiar software engineering term in recent years, thanks to the profound influence of the Model Driven Architecture (MDA). Yet MDD, like MDA itself, defines a general framework, and as such is a generic approach rather than a concrete development methodology. Methodology support for MDA has been rather slow in coming, yet even though several MDA-based methodologies have emerged, they have not been objectively analyzed yet. The need remains for a critical appraisal of these methodologies, mainly aimed at identifying their achievements, and the shortcomings that should be addressed. We provide a review of several prominent MDA-based methodologies, and present a criteria-based evaluation which highlights their strengths and weaknesses. The results can be used for assessing, comparing, selecting, and adapting MDA-based methodologies.

Keywords: Model Driven Architecture, Software Development Methodology, Evaluation Criteria.

1 Introduction

The Model-Driven Architecture (MDA) proposed by the Object Management Group (OMG) defines an approach to information systems specification that separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform. The primary goals of MDA are portability, interoperability, and reusability of software. To achieve these goals, MDA raises the level of abstraction and strives to automate the software generation process.

There are a number of important OMG standards at the core of MDA: The Unified Modeling Language (UML), Meta Object Facility (MOF), XML Metadata Interchange (XMI), and Common Warehouse Metamodel (CWM) [1]. These standards define the core infrastructure of the MDA, and have greatly contributed to modern systems modeling. The core standards of the MDA (UML, MOF, XMI, and CWM) form the basis for building coherent schemes for authoring, publishing, and managing models within a model-driven architecture.

MDA provides an approach for specifying systems in terms of models; system requirements are specified in the *Computation-Independent Model* (CIM); the *Platform-Independent Model* (PIM) is the model that describes the system design independent of the implementation platform; the *Platform-Specific Model* (PSM), on

the other hand, describes the system design in the form of a platform-dependent model. Through its multi-layered modeling approach, MDA raises the abstraction level of traditional platform-dependent design approaches.

A Software Development Methodology (SDM) is a framework for applying software engineering practices with the specific aim of providing the necessary means for developing software-intensive systems [2]. A methodology consists of two main parts: a set of modeling conventions comprising a *modeling language* (syntax and semantics), and a *process* which provides guidelines as to the order of the activities and specifies the artifacts developed using the modeling language. According to the above definition, MDA is not a methodology, but rather an approach to software development. This fact forces organizations willing to adopt the MDA to either transform their software development methodologies into Model-Driven Development (MDD) methodologies, or use new methodologies that utilize MDA principles and tools towards the realization of MDA standards.

This research presents an analytical review and evaluation of a select set of existing MDA-based methodologies. The research has been performed in three main steps: information gathering and methodology selection, development of Evaluation Criteria (EC), and criteria-based evaluation of the selected MDA-based methodologies – with results and observations presented in tabular form. The results can be used by software developers to select the MDA-based methodology best suited to their needs, and by method engineers to create MDA-based methodologies through making use of the strengths identified and addressing the deficiencies observed.

The information gathering step involves studying relevant MDA literature and identifying prominent MDA-based methodologies. An initial set of evaluation criteria is then defined; this initial set is refined and completed to satisfy a predefined set of suitability Meta-Criteria (criteria to evaluate evaluation criteria). The last step involves performing the evaluation based on the set of criteria, and tabulating and analyzing the results.

The rest of the paper is organized as follows: Section 2 provides a review of existing MDA-based methodologies; we present our evaluation results in section 3, and provide an analysis of the results in Section 4; conclusions and areas for furthering this research are presented in section 5.

2 Review of MDA-Based Methodologies

In this section, we present a review on MDA-Based methodologies using the process-centered template introduced in [2], which accentuates the processes of the methodologies. The main factor influencing the selection of these particular methodologies was the availability of proper resources and documentation on their processes.

2.1 ODAC Methodology

ODAC is an MDA based methodology specifically targeted at distributed systems. It provides a set of concepts and structure rules to create systems. The "viewpoint" is the main concept used in this methodology. A viewpoint is a subdivision of the complex specification of the system [3], used for organizing the modeling activities. ODAC

considers five viewpoints: *enterprise, information, computational, engineering, and technology*. It uses these concepts to define development steps by identifying the correspondences between analysis, design, and implementation activities and the viewpoints. ODAC identifies three categories of specifications for each system: *behavioral, engineering, and operational* [4]. ODAC phases are as follows (Fig. 1):

- *Analysis*: produces the behavioral specifications (PIM) of the system.
- *Design*: establishes the engineering specifications (analogous to MDA's Platform Description Model–PDM) and uses it to produce the operational specifications (PSM) via projection of the PIM onto a target environment.
- *Implementation*: generates the execution code from the PSM.

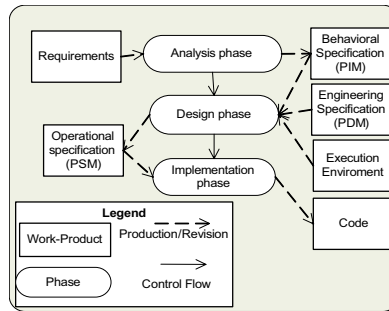


Fig. 1. The ODAC Process

2.2 MASTER Methodology

MASTER was developed as part of a European information project of the same name. The methodology includes an MDD process and a set of system family engineering methods to adapt the MDD process according to customer requirements [5]. The activities and roles of this methodology are defined based on the *Software Process Engineering Metamodel* (SPEM) [6]. MASTER phases are as follows (Figure 2):

- *Capture User Requirements*: covers requirements elicitation and documentation.
- *PIM Context Definition*: describes the domain scope of the software system to be developed. The output of this phase is a clear definition of the system, its goals, and its domain.
- *PIM Requirements Specification*: develops a clear and complete requirements model. The main activity of this phase includes specifying *capabilities* (use cases) and *enforcers* (nonfunctional requirements) of the system.
- *PIM Analysis*: models the internal view of the system regardless of the technological constraints.
- *Design*: models the detailed structure and behavior of the system.
- *Coding and Integration*: develops and verifies the execution code. The code can be generated from the platform-specific model by means of MDA tools.
- *Test*: verifies and validates the final system.
- *Deployment*: transitions the system to the user environment.

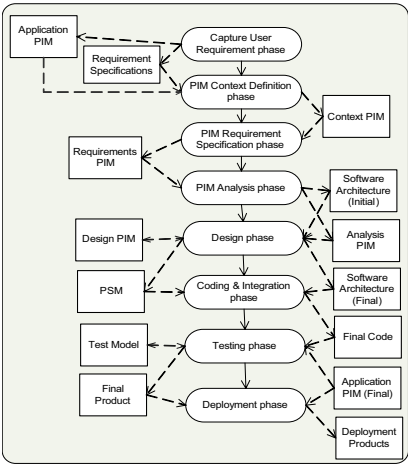


Fig. 2. The MASTER Process

2.3 C³ Methodology

The C³ methodology uses principles of Business Object Oriented Software Technology for Enterprise Reengineering (BOOSTER) to develop business applications [7]. The name C³ is derived from the three concepts of inter-organization *Collaboration*, *Concurrent* software engineering and *Component* development. Concurrent software engineering for both system architectural design and component design is realized through Model-Driven Development and XMI-based techniques.

The phases of this methodology are as follows (Figure 3):

- *Standardization*: downloads the required model elements needed to develop the target business software from the project repository.
- *Software Development*: defines the application’s overall architecture.

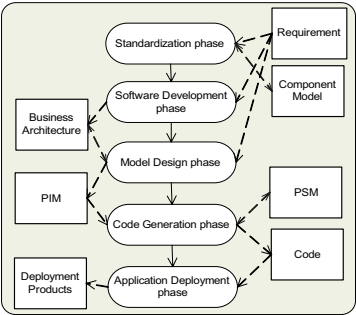


Fig. 3. The C³ Process

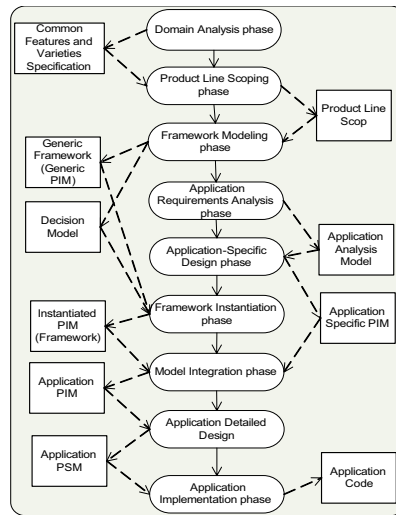


Fig. 4. The DREAM Process

- *Model Design*: refines the business application architecture. The PIM is the output of this phase.
- *Code Generation*: transforms the PIM to PSM and deployable components.
- *Application Deployment*: prepares the software for deployment into the operational environment based on the architectural framework.

2.4 DREAM Methodology

The DRamatically Effective Application development Methodology (DREAM) combines the key activities of Product Line Engineering (PLE) with the model transformation features of MDA [8]. DREAM phases are as follows (Fig. 4):

- *Domain Analysis*: captures the features of several organizations in the same domain, and analyzes the Commonality and Variability (C&V). The output is the specification of common features and differences between organizations.
- *Product Line Scoping*: determines the scope of the target product line.
- *Framework Modeling*: realizes the C&V in a framework, presented as a PIM. The framework defines the general architecture for the desired members of the product line, together with the relationships and constraints.
- *Application Requirements Analysis*: analyzes the application requirements and identifies the features related to the application at hand. The output of this phase is the application analysis model.
- *Application-Specific Design*: realizes the application analysis model as a platform-independent design model. The output is the application-specific PIM.
- *Framework Instantiation*: instantiates the framework for the specific application by setting the variants accordingly. The output of this phase is the instantiated framework PIM.
- *Model Integration*: integrates the specific application PIM and the instantiated framework PIM into one model.

- *Application Detailed Design*: refines the integrated model by considering platform-specific issues, thereby producing the PSM.
- *Application Implementation*: generates the execution code and its related implementation complements – such as the database – from the PSM.

2.5 MODA-TEL Methodology

The MODA-TEL methodology is mainly targeted at distributed applications [9]. The activities and roles of this methodology are defined based on SPEM [6].

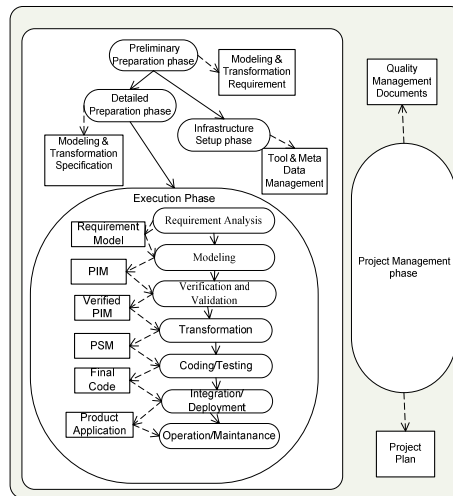


Fig. 5. The MODA-TEL Process

As shown in Fig. 5, the MODA-TEL process consists of five phases:

- *Project Management*: manages and monitors the project.
- *Preliminary Preparation*: identifies modeling and transformation requirements.
- *Detailed Preparation*: determines modeling and transformation specifications.
- *Infrastructure Setup*: provides the tool support and metadata management facilities to be used in the execution phase.
- *Execution*: aims at developing the software artifacts and executable code. The activities include: Requirement Analysis, Modeling (producing the PIM), Verification/Validation, Transformation (PIM to PSM), Coding/Testing, Integration/Deployment, and Operation/Maintenance.

2.6 DRIP-Catalyst Methodology

DRIP-Catalyst is an MDA-based methodology for developing complex, fault-tolerant distributed families of software [10]. DRIP stands for Dependable Remote Interacting Processes. The methodology makes use of the notion of “Atomic Action” as a recovery technique that permits programmers to apply backward and forward error

recovery. A Coordinated Atomic Action (CAA) consists of distributed transactions and an atomic action. The DRIP framework embodies the CAAs in terms of a set of java classes. It builds on the notion of Dependable Multiparty Interaction (DMI). DRIP Catalyst includes a process, a UML profile and a set of transformations, all of which have been integrated into a tool. DRIP-Catalyst phases are as follows (Fig. 6):

- *Problem to Solution Transition*: maps the requirements to the solution through sketching nested CAA diagrams.
- *Platform-Independent Architectural Design*: categorizes the CAAs generated in the previous phase in a coherent package of UML class diagrams.
- *Platform-Independent Detailed Design*: details the modeling elements related to each CAA identified in the previous phase, using UML activity diagrams.
- *Formal Verification*: automatically checks dependability properties using formal methods, and verifies that the models satisfy the requirements, thus producing a verified Platform-Independent Detailed Design Model (PIM2DM).
- *PIM to PSM Transition*: maps the PIM2DM to a platform-specific model through transformation provided by MDA tools, producing the PSM.
- *PSM to Code*: maps the PSM to execution code.
- *Completion*: produces code that can be compiled.
- *Deployment*: defines a configuration set that realizes the deployment of the application, through producing deployment guides and configuration files.

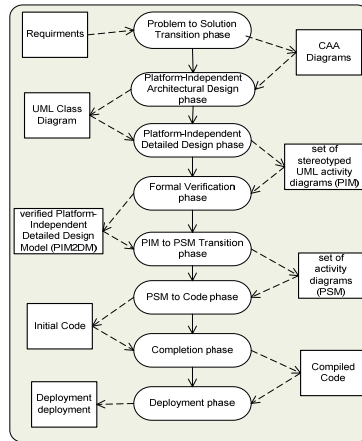


Fig. 6. The DRIP-Catalyst Process

3 Evaluation of MDA-Based Methodologies

We have evaluated the MDA-based methodologies reviewed in the previous section using a method similar to the Feature Analysis approach [11]. The Feature Analysis approach was developed in 1996 under a collaborative project between academia and industry. The outcome of this project was a method to evaluate software engineering

methods and tools. Feature Analysis provides two ways for evaluating any product in terms of results: a simple form and a scale form. In its simple form, the approach presents a list of “yes/no” responses against the existence of some feature in a product. In the scale form, instead of “yes/no” responses, a number between -1 to 5 is used which represents the degree of conformance of the product to a feature.

Since the selected MDA-based methodologies are evaluated with a set of Evaluation Criteria (EC), the development of the criterion set is an important feature of our research. The collected criteria are of two types: (a) Scale type, where a scale represents the degree of presence of a criterion in the methodology (we use scales with three levels for each such criterion); and (b) Narrative type, where the degree of the implementation of the criterion in the methodology is described in narrative form.

The criterion set used for evaluation was developed through gradual refinement: an initial set of general criteria – addressing software development processes and MDD-related issues – was compiled through studying relevant resources, such as official MDA specifications and survey/analysis reports on software development methodologies [2]; the set was then refined, using a set of meta-criteria (criteria to evaluate the EC set) to guide the refinement process towards a reasonably complete and precise set of criteria. The following meta-criteria were defined for this purpose:

- (I) *Existence of tool-related criteria*: used to ensure that the EC set provides tool evaluation, as most MDA practices are enacted through specialized tools. This meta-criterion ensures the existence of criteria that measure how much of a task is governed by tools and how much by the methodology itself; that is, whether the methodology participates in such activities or leaves them to tools.
- (II) *Existence of MDA-related criteria*: used to evaluate the completeness of the EC set as pertinent to MDA aspects. This meta-criterion ensures that the EC set covers the MDA aspects of methodologies. MDA-related criteria are applied to the methodologies only in an MDA-related context.
- (III) *Existence of general criteria*: used to evaluate the completeness of the EC set from general aspects. This meta-criterion ensures that the EC set covers the general aspects of the methodologies; general criteria can be applied to all methodology types: plan-driven, agile and MDA-based methodologies alike.

The refinement process proceeded by categorizing the initial set of criteria into *tool-related*, *MDA-related*, and *general* criteria (according to the above meta-criteria). For each category, relevant resources were then searched iteratively for new criteria and ideas for refining the existing ones. For instance, in striving to complete the criteria belonging to the general category, general software engineering resources and existing documentation on methodologies (such as plan-driven, agile and component-based) were consulted; as an example, since most methodologies (especially agile methods) include activities for customizing and adapting their processes, adaptability was added as a criterion in order to cover this need in MDA-based methodologies; Reusability was also added, based on the observation that MDA and most component-based methodologies consider it as essential. Tables 1, 2 and 3 show the resulting evaluation criteria; the three tables correspond to meta-criteria I, II and III respectively. We have strived to produce a useful, relevant, and meaningful criterion set, while keeping it small and practical. We have therefore focused on addressing features that are particularly important and significant in MDA and MDD.

As a basic requirement, evaluation criteria targeted at software development processes are expected to satisfy certain validity meta-criteria; one such set has been defined in [12]. Our evaluation criteria satisfy the four validity meta-criteria of [12], in that they are: 1) *general* enough to be applicable to all MDA-based methodologies; 2) *precise* enough to help discern the similarities and differences among MDA-based methodologies; 3) *comprehensive* enough to cover all significant features of MDA-based methodologies; and 4) *balanced*, i.e. adequate attention has been given to all three major types of features in a methodology: *technical*, *managerial* and *usage* [12].

Table 1. Tool-related evaluation criteria (satisfying meta-criterion I)

Criterion Name	Criterion Type	Description of Levels
PIM to PSM Transformation	Narrative	Involved: The Methodology explicitly participates in the activity and provides precise techniques/guidelines.
PSM to Code Transformation	Narrative	
Metadata Management	Narrative	Devolved: The activity is devolved to the tools and the methodology does not prescribe the steps that should be performed by the tools.
Automatic Test	Narrative	
Traceability between Models	Narrative	

Table 2. MDA-related evaluation criteria (satisfying meta-criterion II)

Criterion Name	Criterion Type	Description of Levels
Tool Selection/Implementation	Scale Form	A: The methodology does not provide a specific tool and there are no explicit guidelines as to how to select an appropriate alternative tool. B: The methodology does not provide a complete toolset, or only general guidelines are provided for selecting alternative tools. C: The methodology provides a complete toolset, or provides precise guidelines for selecting appropriate alternative tools.
CIM Creation	Scale Form	
PIM Creation	Scale Form	
PSM Creation	Scale Form	
Verification/ Validation	Scale Form	A: The activity is not defined and is devolved to the developers. B: The activity is defined by the methodology, but not in detail. C: The methodology provides explicit and detailed guidelines and techniques for performing the activity.
Extension of Rules	Scale Form	
Round-trip Engineering	Scale Form	
Source Model and Target Model Synchronization	Scale Form	
Use of UML Profiles	Narrative	

Tables 4, 5, and 6 show the results of applying the evaluation criteria to the selected set of MDA-based methodologies. It should be noted that the interpretation of the results is largely dependent on the usage context. The evaluation results can be used for selecting a suitable process from the set of surveyed ones based on a set of predefined requirements, or for identifying shortcomings in these processes in order to improve them. The evaluation framework (criteria) and the results can also be used in a Method Engineering (ME) context; i.e. for guiding the adaptation, extension, meta-modeling/instantiation, and decomposition/assembly of MDA-based processes.

4 Analysis of the Results

The following subsections contain analyses of the evaluation results shown in tables 4, 5, and 6. Of the methodologies reviewed herein, MODA-TEL and MASTER are the methodologies that satisfy most of the criteria.

Table 3. General evaluation criteria (satisfying meta-criterion III)

Criterion Name			Criterion Type	Description of Levels
Coverage	Generic Life Cycle	Requirements Engineering	Scale Form	A: The methodology does not provide coverage for the phase. B: The methodology provides general guidelines for the phase. C: The methodology provides detailed directives for the phase.
		Analysis	Scale Form	
		Design	Scale Form	
		Implementation	Scale Form	
		Test	Scale Form	
		Deployment	Scale Form	
		Maintenance	Scale Form	
	Umbrella Activities	Project Management	Scale Form	A: The methodology does not provide coverage for the activity. B: The methodology provides general guidelines for the activity. C: The methodology provides detailed directives for the activity.
		Quality Assurance	Scale Form	
		Risk Management	Scale Form	
Problem Domain Analysis			Scale Form	A: Problem Domain Analysis has not been addressed. B: Problem Domain Analysis is implicit and confined to requirements engineering. C: Problem Domain Analysis is explicitly addressed by the methodology, and traceability is maintained.
Reusability			Scale Form	A: The task is devolved to the developers; the methodology does not prescribe techniques/guidelines. B: The methodology explicitly prescribes techniques to create potentially reusable artifacts. C: In addition to B, the methodology prescribes techniques to record syntactic/semantic features of reusable aspects for future reuse.
Adaptability			Scale Form	A: No techniques are prescribed for adapting the methodology. B: The methodology provides extensible notations. C: In addition to B, the methodology prescribes explicit techniques for configuring the process and/or modeling language.
Completeness of Definition			Scale Form	A: Some phases of the methodology are not completely specified. B: All phases are completely specified (in breadth) but details are lacking in some phases. C: All phases are completely specified at an adequate level of detail.
Methodology Type			Scale Form	<u>Extended</u> : The methodology is the result of extending an existing methodology to support MDA-based development. <u>MDA-based (Genuine)</u> : The methodology has been created from scratch aimed at supporting MDA-based development.
Application Scope			Narrative	

4.1 Tool-Related Evaluation Results

The results seem to show that most of the methodologies examined do not offer any guidelines as to how MDA tools should be used in coherence with the methodology, thus leaving all tool-related issues to the tools themselves. The only counterexamples are MODA-TEL and MASTER, and even these do not provide full coverage.

4.2 MDA-Related Evaluation Results

Since tools have a key role in MDD, MDA-based methodologies are expected to incorporate activities aimed at selecting or implementing appropriate tools. While DRIP-Catalyst and MASTER incorporate suitable tools themselves, MODA-TEL provides guidelines for selecting the tool from existing commercial and open source MDA toolsets. DREAM, C³ and ODAC are at the other end of the spectrum: they do not even offer any guidelines as to how an appropriate alternative tool can be selected.

All of the MDA-based methodologies reviewed incorporate activities for creating the PIM and PSM; creation of the CIM, however, is only addressed by MASTER and C³. Due to the model-centric nature of MDD, syntactic and semantic accuracy of the models is essential, as is their traceability to requirements; however, most of the processes reviewed do not provide adequate support for model verification/validation.

All the methodologies reviewed (except for MASTER) are weak in providing other important MDA features; i.e., support for extension of rules, round-trip engineering, and source-model and target-model synchronization.

4.3 General Evaluation Results

Most of the methodologies reviewed cover the analysis, design, and implementation phases of the generic software development life cycle, either by prescribing specialized techniques, or through making use of existing object oriented techniques; however, the requirements engineering, test, deployment, and maintenance phases are not adequately supported in most of them. For instance, only MODA-TEL supports maintenance, whereas MDA-based maintenance requires special techniques that cannot be simply borrowed from existing methodologies. Another area where MDA-based processes need improvement is support for umbrella activities; of the processes reviewed, only MODA-TEL and MASTER provide support for project management and quality assurance, while risk management is not supported by any methodology.

Table 4. Results of applying the Tool-related evaluation criteria

Methodology Criterion	MODA-TEL	MASTER	C ³	ODAC	DREAM	DRIP-Catalyst
PIM to PSM Transformation	Involved	Involved	Devolved	Devolved	Devolved	Devolved
PSM to Code Transformation	Involved	Involved	Devolved	Devolved	Devolved	Devolved
Metadata Management	Involved	Involved	Involved	Devolved	Devolved	Devolved
Automatic Test	Devolved	Involved	Devolved	Devolved	Devolved	Devolved
Traceability between Models	Involved	Devolved	Devolved	Devolved	Devolved	Devolved

Table 5. Results of applying the MDA-related criteria

Methodology Criterion	MODA-TEL	MASTER	C ³	ODAC	DREAM	DRIP-Catalyst
Tool Selection/ Implementation	B	C	A	A	A	C
CIM Creation	A	B	B	A	A	A
PIM Creation	B	C	B	C	B	C
PSM Creation	B	C	B	B	B	B
Verification/ Validation	B	A	A	A	A	B
Extension of Rules	C	B	A	B	B	A
Round-trip Engineering	B	A	A	A	A	A
Source Model and Target Model Synchronization	B	B	A	A	A	A
Use of UML Profiles	Used for Requirements Representation	Used for Annotating PIM with Management Information	Not Used	Used for Describing Development Steps.	Used for Defining Well-Structured Models	Used for Defining Fault-Tolerant Transactions

Table 6. Results of applying the General criteria

Methodology		MODA-TEL	MASTER	C ³	ODAC	DREAM	DRIP-Catalyst	
Criterion								
Coverage	Generic Life Cycle	Requirements Engineering	B	C	B	A	B	A
		Analysis	B	C	A	C	B	C
		Design	B	C	B	C	B	C
		Implementation	B	B	B	B	B	B
		Test	B	C	A	A	A	A
		Deployment	B	B	B	A	A	B
		Maintenance	B	A	A	A	A	A
	Umbrella Activities	Project Management	B	C	A	A	A	A
		Quality Assurance	B	B	A	A	A	A
		Risk Management	A	A	A	A	A	A
	Problem Domain Analysis		A	B	B	A	B	A
	Reusability		B	B	B	A	B	A
Adaptability		B	B	A	A	A	A	
Completeness of Definition		B	C	A	B	B	A	
Methodology Type		MDA-Based	MDA-Based	Extended	Extended	Extended	Extended	
Application Scope		Distributed Applications	Information Systems	Business Software	Agent-Oriented Systems	Product Line Engineering	Distributed Fault-Tolerant Applications	

Most of the MDA-based methodologies reviewed provide techniques for creating and applying reusable artifacts. Support for reusability, however, is not comprehensive enough: Methodologies do not prescribe techniques for recording the syntactic- and semantic features of reusable artifacts in order to facilitate future reuse.

Developers prefer methodologies which lend themselves to customization and adaptation; but of the methodologies reviewed, only MODA-TEL and MASTER provide adaptability (in the form of extensible notations). Furthermore, methodologies need to be properly defined in order to be usable; however, some of the methodologies reviewed herein (e.g., C³) suffer from cursory definitions of activities.

5 Conclusions and Future Work

MDA cannot be useful without software development methodology support and the tools that implement its main concepts and standards. We have surveyed several prominent MDA-based methodologies and have evaluated them using a predefined set of evaluation criteria. According to the evaluations results, we can conclude that:

- The MDA-based methodologies studied herein are not mature enough, especially as pertaining to providing support for standard software engineering activities.
- Definitions of methodologies are not complete.
- Umbrella activities are not adequately addressed in most of these methodologies.
- Most of the methodologies do not participate in the activities that are supported by tools, and do not even provide guidelines for tool usage.
- PIM and PSM production is supported by the majority of these methodologies; the CIM, however, is mostly neglected.
- Most of the methodologies use conventional OOA and OOD techniques to produce PIMs.

We aim to further this research by identifying a set of process patterns showing recurring activities in different MDA-based methodologies, thereby producing a generic and instantiable process for such methodologies.

References

1. Mukerji, I., Miller, J.: MDA Guide Version 1.0.1. OMG (2003)
2. Ramsin, R., Paige, R.F.: Process-Centered Review of Object-Oriented Software Development Methodologies. *ACM Computing Surveys*~40(1), 1--89 (2008)
3. Gervais, M.: Towards an MDA-Oriented Methodology. In: 26th Annual International Computer Software and Applications Conference (COMPSAC 2002), pp. 265--270. IEEE Press, Oxford (2002)
4. Gervais, M.: ODAC: An Agent-Oriented Methodology Based on ODP. *Journal of Autonomous Agents and Multi-Agent Systems*~7(3), 199--228 (2003)
5. Larrucea, X., Diez, A.B.G., Mansell, J.X.: Practical Model Driven Development process. In: Second European Workshop on Model Driven Architecture (MDA), UK (2004)
6. Object Management Group: Software Process Engineering Metamodel v1.0 (SPEM). OMG (2002)

7. Hildenbrand, T., Korthaus, A.: A Model-Driven Approach to Business Software Engineering. In: 8th World Multi-Conference on Systemics, Cybernetics and Informatics, Florida. Information Systems, Technologies and Applications, vol.~IV, pp. 74--79 (2004)
8. Kim, S., Min, H.G., Her, J.S., Chang, S.H.: DREAM: A practical product line engineering using model driven architecture. In: ICITA 2005, Australia, pp. 70--75 (2005)
9. Gavras, A., Belaunde, M., Ferreira Pires, L., Andrade Almeida, J.P.: Towards an MDA-based development methodology. In: Oquendo, F., Warboys, B.C., Morrison, R. (eds.) EWSA 2004. LNCS, vol.~3047, pp. 71--81. Springer, Heidelberg (2004)
10. Guelfi, N., Razavi, R., Romanovsky, A., Vandenberg, S.: DRIP Catalyst: an MDE/MDA Method for Fault-tolerant Distributed Software Families Development. In: OOPSLA \& GPCE workshop on best practices for Model Driven Development, Portland (2004)
11. Kitchenham, B., Linkman, S., Law, D.: DESMET: a methodology for evaluating software engineering methods and tools. *Computing and Control Engineering Journal*~8, 120-126 (1997)
12. Karam, G.M., Casselman, R.S.: A cataloging framework for software development methods. *IEEE Computer*~26(2), 34--45 (1993)

Where Is the Proof? - A Review of Experiences from Applying MDE in Industry

Parastoo Mohagheghi and Vegard Dehlen

SINTEF, P.O. Box 124- Blindern
N-0314 Oslo, Norway
{Parastoo.Mohagheghi,Vegard.Dehlen}@sinetf.no

Abstract. Model-Driven Engineering (MDE) has been promoted as a solution to handle the complexity of software development by raising the abstraction level and automating labor-intensive and error-prone tasks. However, few efforts have been made at collecting evidence to evaluate its benefits and limitations, which is the subject of this review. We searched several publication channels in the period 2000 to June 2007 for empirical studies on applying MDE in industry, which produced 25 papers for the review. Our findings include industry motivations for investigating MDE and the different domains it has been applied to. In most cases the maturity of third-party tool environments is still perceived as unsatisfactory for large-scale industrial adoption. We found reports of improvements in software quality and of both productivity gains and losses, but these reports were mainly from small-scale studies. There are a few reports on advantages of applying MDE in larger projects, however, more empirical studies and detailed data are needed to strengthen the evidence. We conclude that there is too little evidence to allow generalization of the results at this stage.

Keywords: Model-driven engineering, quality, productivity, evidence.

1 Introduction

The model-driven approach has received considerable attention this decade. The OMG's Model-Driven Architecture (MDA) initiative, Model-Driven Development (MDD) or Model-Driven Engineering (MDE)¹ has been hailed as the solution to handle the key problem facing the software development industry; increasing complexity, by (1) providing better abstraction techniques and (2) facilitating automation. By switching to a MDE approach, businesses are promised to reap benefits through increased productivity and software quality [26].

The motivation behind this paper is that even though many promises are made, these are in most cases poorly, if at all, supported by evidence. During recent years we have witnessed the surfacing of attempts to evaluate practices and benefits of MDE through empirical studies; including experiments and industry experience

¹ In the remainder of the paper we use MDE to refer to a model-driven software development approach, also where MDD is used in the papers.

reports. This paper, the result of an extensive literature review, contributes to the state of evidence in MDE by gathering the individual evaluations and providing a detailed overview of industry's experiences with MDE.

The remainder of the paper is organized as follows. Section 2 presents the review framework and the three research questions leading the review, the strategy used for literature search, the publication channels, and an overview of the reviewed papers. Section 3 through 5 reports our findings, before Section 6 summarizes and concludes the paper.

2 The Review Process and an Overview of Papers

2.1 The Review Framework and Research Questions

We follow the review framework presented in [19], adopted to this review and depicted in Figure 1. The formulation of the review questions follows recommendations by Dybå et al. for collecting evidence as answer to questions. Questions should be well-partitioned into *intervention*, *context* and *effect* [9]. In this review, the intervention is “MDE” (vs. non-MDE approaches), the context is “industrial settings” and the effects are “changes in productivity and quality, or cost savings”.

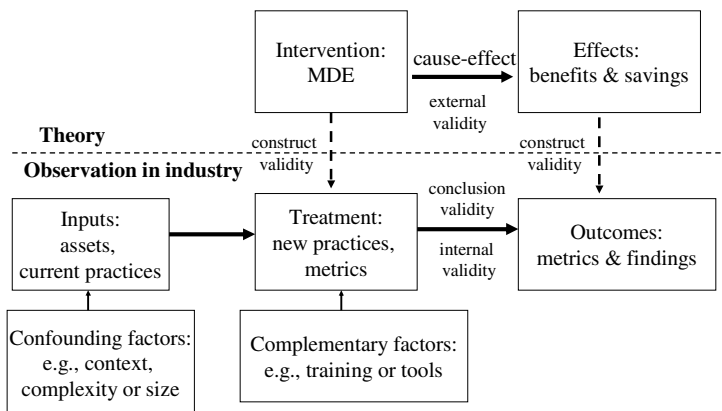


Fig. 1. The review process

To understand the intervention and context, we ask the following Research Questions (RQs):

- *RQ1. Where and why is MDE applied?*
- *RQ2. What is the state of maturity of MDE?*

And to evaluate the effects, we ask:

- *RQ3. What evidence do we have on the impact of MDE on productivity and software quality?*

2.2 An Overview of the Reviewed Papers

We searched the following publication channels for industrial studies related to MDE:

- The Software and Systems Modeling (SoSyM) journal from 2002 (the first issue).
- The Empirical Software Engineering journal since 2000.
- Proceedings of the UML conference from 2000 to 2004, succeeded by the MODELS conference to 2006.
- Proceedings of The European Conference on MDA- Foundations and Applications (ECMDA-FA) started in 2005, and to 2007.
- Proceedings of the DSM workshops at OOPSLA since start in 2002.

We also performed a search by keywords in the IEEE Xplore, the ACM digital library and the Internet. A few additional papers were discovered through references in the detected papers. The review identified 33 papers and reports (generally called papers). From these, we excluded 8 papers with claims on industrial application but no description of the application (a list can be provided by the authors). This left 25 papers for the review.

It was not possible to extract information on the size of projects from the majority of papers. For appraising the evidence, we asked what types of studies were performed (see [19] for a definition of study types). We concluded that:

- 20 of papers are experience reports from single projects with description of a project or development method and discussion of experiences [1-7, 8, 10, 11, 13, 20-24, 27-29 and 31]. Of these, only two include some quantitative data from the projects (both from Motorola).
- Three papers have used interviews and questionnaires in addition to observations [24-26].
- Three papers describe comparative studies (comparing projects or development of components with each other) [12, 14 and 16]. From these, [12] provides no quantitative data.
- One paper describes three (quasi)experiments [16].

Only seven papers report experiences from completed projects [1, 3, 6, 13, 25, 27 and 29], while the others are from pilot studies or ongoing projects at the time of reporting, and one is from a terminated project [ABB Robotics in 26].

When it comes to publication channels, 13 papers are published in the proceedings of conferences (especially the ECMDA-FA conference), 9 papers in workshops and satellite activities of conferences, two are online reports and only one is published in a journal.

3 Where and Why Is MDE Applied (RQ1)?

A broad range of companies in various domains report their experience from investigating or applying MDE. To name some, the papers cover:

- Telecommunications domain [2, 3, 16, 21, 26, 28 and 29].
- Business applications and financial organizations [1, 7, 8, 16 and 24].

- Defense / aerodynamics / avionic systems [5 and 11].
- Web applications [6 and 14].

We found examples of safety-critical and trustworthy systems [5, 11 and 27] and embedded systems [23 and 27]. MDE approaches are also applied to software product lines as in [2, 10 and 27]. In connection with legacy systems, Bloomfield reports successful remodeling of a component [5] and Raistrick reports developing new components that were integrated with existing components [22]. On the other hand, ABB Robotics refrained from adopting MDE due to the base of legacy code [26].

Regarding motivations for evaluating or applying MDE, the papers discuss:

- *Increasing productivity and shortening development time*: as in [12, 14, 16, 25, Ericsson in 26 and 29].
- *Improving quality*: improving the quality of the generated code [25, 27 and 29], improving the quality (assurance) of system requirements [4] and managing requirement volatility [22], improving the quality of intermediate models [4], and earlier detection of bugs [12, 27 and 29].
- *Automation*: generating code and other artifacts and introducing automation into the development process [1-3, 6, 7, 8, 11-13, 16, 21, 23 and 27], and model-based simulation and testing [3].
- *Standardization and formalism*: providing a common framework for software development across the company and phases of the lifecycle [2, 24 and 25], formalize and organize software engineering knowledge at a higher level of abstraction [29], and common data exchange format [20].
- *Maintenance and evolution concerns*: maintaining the architecture intact from analysis to implementation [25], evolution of legacy systems [12], concerns over software method and tool obsolescence [5], verification of system by producing models from traces [28] and that PIMs have long lifespan [14].
- *Improved communication and information sharing*: between stakeholders [18 and 24] and within the development team [12, 26 and 27] and ease of learning [27 and 29].

Additional motivations are *traceability throughout software development artifacts* [17 and 26], *early assessment* [22 and 26], *promoting reuse* [2, 18, 24 and 29], *porting of solutions to new platforms* [12 and 13], and *the ability to estimate costs based on the models* [22 and 26].

On the above list, increasing productivity (and shortening development time) and improving quality may be regarded as the ultimate reasons for applying MDE. The other items, on the other hand, are basically means towards these two ends.

4 What Is the Experienced Maturity of MDE (or, the-State-of-MDE) (RQ2)?

In this section we present findings related to the current state of practicing MDE. It covers automation as a key means to achieve the MDE benefits. We also discuss the state of software development processes and tools for MDE.

4.1 Level of Automation

By using transformations the MDE approach emphasizes generating models, code and other artifacts from models, in addition to verification and validation on the model level. In this section we analyze to what extent this is possible in the presented contexts and with the current state of tools

Automatic Generation of Code. While some papers report generating all or most of the code from the models [5 and 6], others report that only part of the code could be generated. Motorola evaluates the potential of MDE in generation to be between 65 to 96 percent depending on the type of the code (low level code is not captured in the design and is unlikely to be generated), and perceives the status of code generators as satisfactory in producing code with no introduced defects [3 and 29]. Automatic generation of code required developing Domain Specific Languages (DSLs) or UML profiles and own code generators in several cases, as in [1, 3, 7, 10, 21 and 29].

Generating XML Schemas. In [20] a metamodel was implemented as a UML profile and the needed XML schemas were generated directly from the marked PIM models. [2]'s toolset also includes an XML schema generator, a code generator using the schemas and other outputs. In the case of [10], the developed framework included a XML schema generator, HTML documentation generator and a model browser.

Automation of Testing. In Motorola, by using TTCN scripts, 90% of the tests are automated which has led to a 30% reduction in box-test cycle time [29].

Executable Models. A few papers have discussed that developing executable models is still a challenge. Deng et al. write that they used Visio as a static design tool, while a dynamic provisioning tool is desired to make the blocks executable [8]. MacDonald et al. report difficulties in specifying behavior using Telelogic Tau and that they could not develop executable models [12].

4.2 Software Processes

The importance of utilizing a defined process in software engineering has been known for several years. However, most “tried and tested” processes are not tailored for MDE, which does not make any assumptions on the software development process or the design methodology. Baker et al. report that many teams in Motorola encountered major obstacles in adopting MDE due to the lack of a well-defined process, lack of necessary skills and inflexibility in changing the existing culture [3]. Also, MacDonald et al. write that there is no well-defined process for developing non-trivial MDE components, especially when these are part of legacy systems [12]. Staron means that there are two reasons for why they currently find it unrealistic to purely use a MDE process [26]:

1. Software engineering methods are not fitted to use models as main artifacts, i.e. activities such as analysis and evaluation is still largely done at the code level.
2. Software engineering environments are not mature enough.

Some have attempted to apply pre-existing software processes to MDE, such as using a modified version of the Rational Unified Process (RUP) [24], and combining agile methods and MDE [23, 27 and 30]. Others have attempted defining processes for MDE. Firstly, THALES has defined a MDE process by extending the IEEE 1471 standard [11]. Secondly, Biffl et al. propose an iterative software development lifecycle, which includes creating models with explicit stakeholder requirements, a first quality assurance (QA) step with type checking and semantic validation and transforming these into intermediate models, and a second QA step with static validation of models [4]. Thirdly, Staron et al. discuss that raising the abstraction level and employing automatic code generation moves the complexity of software development to transformations [25]. An MDE process should consequently prioritize defining transformations before defining profiles, since profiles are considered a means of making the transformations automated. The importance of developing transformations early is further supported by [21]. None of the studies report using any of the already existing – although few – model-based methodologies, e.g. Kobra² or COMET³.

4.3 Tools

Supporting MDE with a comprehensive tool environment is crucial, as many of the techniques promoted as necessary in MDE strongly depend on proper tool support. A survey performed among industry participants (presented in [26]) showed that, when considering whether or not to adopt MDE, the availability of tools was perceived as the most influential factor. However, a tool chain has to integrate the various tools for software development (e.g., requirements management, modeling, model transformations, traceability, simulation, validation and testing [15]), support multiple platforms and domain-specific design [12] and the possibility to generate *correct* code by adding constraints and rules [1, 13, 27 and 31].

Integrating a tool suite that satisfies these requirements into a coherent environment is evidently a challenge. In the MODELWARE project, a wide range of tools were used, but all partners experienced problems with instability of the tools and their integration [15 and 17]. Also according to Motorola, third-part MDE tools do not scale well to large system development [3]. Safa writes that using third-party tools raises questions of suitability for the product, adaptability to new platforms, availability over time, and loss of differentiation factors since competitors may use the same tools [23].

The vendor lock-in problem persuades some users to use open source tools such as the Eclipse framework. Others combine third-party products with self-developed tools [27 and 29], or develop their own tools [2, 4 and 10]. Having to invest time and effort into the development and maintenance of an MDE tool chain raises issues of cost. France Telecom calculated that the cost for creating their tool chain in MODELWARE was approximately one person-year in terms of resources, in addition to approximately 0.4 person-year for maintenance [15].

² www.old.netobjectdays.org/pdf/02/papers/node/0308.pdf

³ http://www.uio.no/studier/emner/matnat/ifi/INF5120/v05/undervisningsmateriale/COMET_Method_v2-4.pdf

5 What Evidence Do We have on the Impact of MDE on Productivity and Software Quality? (RQ3)

Productivity and software quality gains are often given as main motivations for selecting new technologies, and most papers in this review include discussion of either one or both of the aspects. In this section, we present the reported data, observations and explanations on observations.

5.1 MDE Impact on Productivity

Three of the papers in the review report results from comparative studies on productivity (i.e., developing a product twice or comparing with company baseline data), although the studies are of small-scale.

Firstly, in a report from 2003, the Middleware Company, on behalf of Compuware, conducted a comparative case study on the productivity of MDA [14]. Two teams developed the same application, one using MDE and the other using a non-MDE approach. The result was that the MDE team developed their application 35% faster than the other team – needing 330 hours compared to 507,5. It is worth noting that the MDE team used a tool with pre-made transformation mappings, which relieved them of potential work. On the other hand, this was the developers' first experience with MDE and related tools, which would presumably hamper their productivity. Issues like application performance and maintenance were not evaluated.

Secondly, we have the results of the EU IST project MODELWARE⁴ [16]. In September 2006, results from six small-scale case studies and (quasi)experiments performed by five industrial partners were disseminated. When it comes to productivity, the results are differing:

- In WM-Data (desktop business applications), two developers re-implemented a subset of requirements and the effort was compared to some baseline data. The productivity gain was on average 24% using MDE.
- WesternGeco (oil and gas exploration) performed an experiment with 24 developers who were given four tasks – two involving a traditional development process and two involving MDE. Only eight subjects finished the experiment due to problems with the MDE tooling and complexity of the tasks. The results show no difference in productivity between the two approaches.
- A team of two developers from Enabler (specialist in creation and integration of IT solutions for retailers) developed a module twice over a period of approximately 300 hours. The results show an overall loss in productivity when using MDE by 27%. When discounting the problems with the use of immature tools, the loss in productivity was 10%.
- France Telecom measured the effort needed to specify, implement and change five different functional units, normalized by the weight of their complexity, and compared to the data on effort spent in a non-MDD approach. A productivity gain of 20% was measured during design activity and 69% during coding. This observed productivity gain does not take into account the cost of the development of tool chain.

⁴ <http://www.modelware-ist.org>

The third paper reports redevelopment of a small component of a legacy system using MDE [12]. The authors report that there is no proof that development speed is improved, especially with the workarounds required to integrate with legacy systems.

A few other papers have reported productivity gains in single projects when applying MDE, without having a clear baseline or providing detailed data. Firstly, Motorola has employed a MDE approach for more than 15 years and has shipped millions of lines of code based on MDE [3]. All in all, they have experienced a 2X–8X productivity improvement when measured in terms of equivalent lines of source code. These numbers are all approximates, as Motorola is lacking a common baseline. Also, an experience report by Trask et al. deals with the application of a combination of software product line and MDE techniques to the “software defined radio” domain [27]. The programmers reportedly experienced a 500% productivity gain, minimum, by utilizing their domain-specific modeling tool. These results are based on experiences and are not validated by data or experiments. And finally, Thales Air Traffic Management (TATM) in the MODELWARE project estimated 5 to 25% productivity gains based on the assumption that a certain type of defect (interface mismatch) cannot occur because of the MDE process.

The industrial papers that reported productivity gains accredited the improvement to automatic code generation [3, 14, 27 and 29], model-based simulation and testing [3, 15 and 29], automatic test generation [3, 29], avoiding defects [27 and TATM in 16], domain-specific languages [27], and reuse of design and test between platforms or releases [29].

As discussed above, there are also reports of productivity loss. The main reasons are mentioned to be immature tools and high start up costs [Enabler in 18], and that modeling can be at least as complex as programming with a traditional third generation language [12].

5.2 MDE Impact on Software Quality

Among industry adopters discussing improvements in software quality due to MDE, the key experienced benefit is a drastic reduction in the number of software defects. However, there are not much quantitative data presented in the papers.

Firstly, we discuss the Motorola case. Weigert and Weil write that with MDE, there are fewer inspections required to ensure the quality of the developed code than using conventional development. In addition, inspection rates are higher and have increased from 100 source lines per hour to in between 300 and 1000 source lines per hour [29]. Motorola data also shows that simulation is about 30% more effective in catching defects than the most rigorous inspections, and that defects are detected earlier in the software development lifecycle. They expect a 3X reduction in defects, which is backed up by an earlier Motorola study, experiencing “a 1.2X–4X overall reduction in defects and a 3X improvement in phase containment of defects”. Baker et al. write that it is not unusual to see a 30X–70X reduction in the time needed to correctly fix a defect by detecting and correcting the problems at the model level [3].

That models are verified through simulation (or other techniques) and checked for completeness also improves quality significantly according to [15 and 29]. In [15], France Telecom writes that being able to validate the specification using simulation, allows them to “to eliminate uncomfortable ergonomics that would be difficult to detect otherwise”.

6 Summary and Conclusions

This review examined experiences of applying MDE in industry published since 2000, showing the status as it is and identifying gaps for future research. Validity threats are identified to be:

- The low number of studies is the main threat to the external validity of the results (i.e., generalization to a population or theory).
- Success cases are more likely to be published than failures.
- Some companies may refrain from publishing their results to keep their competitive advantage.
- Projects with external financing, such as EU projects, may report biased results. However, in the case of the MODELWARE project, we know the details of the studies and do not consider this as a threat to the validity of the results.
- There are few results of large-scale studies and the scalability of MDE to large system development should be evaluated in more cases.
- There is a lack of baseline data in most companies, which results in subjective evaluations.
- Most studies do not include enough quantitative data or the metrics are not properly defined.

Due to the low number of experiments, we do not discuss experimentation validity threats in more details. Finally, we mainly searched journals and conferences that have a review process and are considered relevant to our subject, in addition to including two on-line reports [14 and 16]. Additional search in other publication channels may add new papers which can extend the results of this review.

We asked three research questions and the findings are summarized here:

- *RQ1-Context and motivation.* MDE is applied in a wide range of domains; including safety-critical systems and product lines. MDE is assumed to lead to higher productivity (by increased automation in the development process), increased standardization and formalism, and improved communication within development teams and with external stakeholders, to name the most frequently given benefits. Labor-intensive and error-prone development tasks are automated and best-known solutions can be integrated in code generators, resulting in reducing defects and improving software quality.
- *RQ2-State-of-the-MDE.* The current state of MDE is far from mature. There is a varying degree of automation and it is mostly applied for code generation. Examples of using models for simulation and test generation are also given. Tools are improved during the recent years but several papers still discuss the lack of a coherent MDE environment and tool chain. Tools should scale to large-scale development and support the domain-specific approach more effectively. Software processes should also be adapted to MDE. Other challenges in adopting MDE are the complexity of modeling itself, developing PIMs that are portable to several platforms and using MDE together with legacy systems.
- *RQ3-MDE impact on productivity and software quality.* We found some quantitative evidence on productivity gains in the Motorola context [3 and 29], from a domain-specific environment [27], and three small-scale comparative

studies and quasi-experiments described in [14 and 16]. The Motorola studies are the only ones providing some quantitative data on software quality improvements. Software quality benefits are discussed in several papers but are not backed up with data.

Modeling should be easier and faster than code writing to promote MDE. Appropriate tools and processes and increased expertise on modeling are areas for improvement in most cases. Combining MDE with domain-specific approaches and in-house developed tools has played a key role in successful adoption of the approach in several cases. One of promises of MDE in increasing portability of solutions to multiple platforms has not often been feasible, mainly due to the fact that tools are bound to specific platforms. However, most papers evaluate models as useful for improving understandability and communication among stakeholders.

It is a challenge to collect convincing proof on any technology – MDE included. Future work for evaluation of MDE should focus on performing more empirical studies, improving data collection and analyzing MDE practices so that success and failure factors and appropriate contexts for MDE can be better identified. Future research should also cover evaluating Return-On-Investment (ROI) of MDE in various contexts and for different project scales. We only found an estimation of ROI in France Telecom which provided an estimation based on costs related to the training and tool chain setup and the measured productivity gain [15]. High initial investment and unsure benefits were one of the issues influencing the decision of the non-adopters [26]. In the MODELPLEX project⁵, we continue the MODELWARE approach in combining research with industrial application and evaluation and will report the results of research on applying MDE in large and complex system development on the project website and in future publications.

Acknowledgments. This research was done in the “Quality in Model-Driven Engineering” project⁶ at SINTEF. We thank Dr. Arnor Solberg and Mr. Tor Neple for their comments and constructive criticism.

References

1. Anonsen, S.: Experiences in Modeling for a Domain Specific Language. In: Jardim Nunes, N., Selic, B., Rodrigues da Silva, A., Toval Alvarez, A. (eds.) UML Satellite Activities 2004. LNCS, vol. 3297, pp. 187–197. Springer, Heidelberg (2005)
2. Bahler, L., Caruso, F., Micallef, J.: Experience with a Model-Driven Approach for Enterprise-Wide Interface Specification and XML Schema Generation. In: Seventh IEEE International Enterprise Distributed Object Computing Conference (EDOC 2003), pp. 288–295 (2003)
3. Baker, P., Loh, P.S., Weil, F.: Model-Driven Engineering in a Large Industrial Context - Motorola Case Study. In: ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems (MoDELS/UML 2005). LNCS, vol. 3713, pp. 476–491. Springer, Heidelberg (2005)

⁵ European IST-34081, <https://www.modelplex.org/>

⁶ <http://quality-mde.org/>

4. Biffl, S., Mordinyi, R., Schatten, A.: A Model-Driven Architecture Approach Using Explicit Stakeholder Quality Requirement Models for Building Dependable Information Systems. In: 5th International Workshop on Software Quality (WoSQ 2007) at ICSE 2007, p. 6. IEEE, Los Alamitos (2007)
5. Bloomfield, T.: MDA, Meta-Modeling and Model Transformation: Introducing New Technology into the Defense Industry. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 9–18. Springer, Heidelberg (2005)
6. Brambilla, M., Ceri, S., Fraternali, P., Acerbis, R., Bongio, A.: Model-Driven Design of Service-Enabled Web Applications. In: ACM SIGMOD International Conference on Management of Data, pp. 851–856 (2005)
7. Burgstaller, B., Wuchner, E., Fiege, L., Becker, M., Fritz, T.: Using Domain Driven Development for Monitoring Distributed Systems. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 19–24. Springer, Heidelberg (2005)
8. Deng, G., Lu, T., Turkay, E., Gokhale, A., Schmidt, D., Nechypurenko, A.: Model Driven Development of Inventory Tracking System. In: 3rd OOPSLA Workshop on Domain Specific Modeling (DSM 2003), p. 6 (2003)
9. Dybå, T., Kitchenham, B.A., Jørgensen, M.: Evidence-Based Software Engineering for Practitioners. *IEEE Software* 22(1), 58–65 (2005)
10. Jonkers, H., Stroucken, M., Vdovjak, R.: Bootstrapping Domain-Specific Model-Driven Software Development within Philips. In: 6th OOPSLA Workshop on Domain Specific Modeling (DSM 2006), p. 10 (2006)
11. Jouenne, E., Normand, V.: Tailoring IEEE 1471 for MDE Support. In: Jardim Nunes, N., Selic, B., Rodrigues da Silva, A., Toval Alvarez, A. (eds.) UML Satellite Activities 2004. LNCS, vol. 3297, pp. 163–174. Springer, Heidelberg (2005)
12. MacDonald, A., Russell, D., Atchison, B.: Model-Driven Development within a Legacy System: an Industry Experience Report. In: Australian Software Engineering Conference (ASWEC 2005), pp. 14–22. IEEE, Los Alamitos (2005)
13. Mattsson, A., Lundell, B., Lings, B., Fitzgerald, B.: Experiences from Representing Software Architecture in a Large Industrial Project using Model Driven Development. In: 2nd Workshop on SHaring and Reusing architectural Knowledge Architecture, Rationale, and Design Intent (SHARK-ADI 2007) at ICSE 2007, p. 6. IEEE, Los Alamitos (2007)
14. Middleware Company. Model Driven Development for J2EE Utilizing a Model Driven Architecture (MDA) Approach. Productivity Analysis. Report by the Middleware Company on behalf of Compuware (2003), http://www.omg.org/mda/mda_files/MDA_Comparison-TMC_final.pdf
15. MODELWARE D5.3-4 France Telecom ROI, Assessment, and Feedback. Revision 1.1 (2006), <http://www.modelware-ist.org>
16. MODELWARE D5.3-1 Industrial ROI, Assessment, and Feedback- Master Document. Revision 2.2 (2006), <http://www.modelware-ist.org>
17. MODELWARE D5.3-5 Western Geco ROI, Assessment, and Feedback. Revision 0.3 (2006), <http://www.modelware-ist.org>
18. MODELWARE D5.3-2 Enabler ROI, Assessment, and Feedback. Revision 1.1 (2006), <http://www.modelware-ist.org>
19. Mohagheghi, P., Conradi, R.: Quality, Productivity and Economic Benefits of Software Reuse: a Review of Industrial Studies. *Empirical Software Engineering Journal* 12(5), 471–516 (2007)
20. Pagel, M., Brörkens, M.: Definition and Generation of Data Exchange Formats in AUTOSTAR. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 52–61. Springer, Heidelberg (2006)

21. Presso, M.J., Belaunde, M.: Applying MDA to Voice Applications: an Experience in Building an MDA Tool Chain. In: Hartman, A., Kreische, D. (eds.) ECMDA-FA 2005. LNCS, vol. 3748, pp. 1–8. Springer, Heidelberg (2005)
22. Raistrick, C.: Applying MDA and UML in the Development of a Healthcare System. In: Jardim Nunes, N., Selic, B., Rodrigues da Silva, A., Toval Alvarez, A. (eds.) UML Satellite Activities 2004. LNCS, vol. 3297, pp. 203–218. Springer, Heidelberg (2005)
23. Safa, L.: The Practice of Deploying DSM, Report from a Japanese Appliance Maker Trenches. In: 6th OOPSLA Workshop on Domain Specific Modeling (DSM 2006), p. 12 (2006)
24. Shirtz, D., Kazakov, M., Shaham-Gafni, Y.: Adopting Model Driven Development in a Large Financial Organization. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA 2007. LNCS, vol. 4530, pp. 172–183. Springer, Heidelberg (2007)
25. Staron, M., Kuzniarz, L., Wallin, L.: Case Study on a Process of Industrial MDA Realization: Determinants of Effectiveness. *Nordic Journal of Computing* 11(3), 254–278 (2004)
26. Staron, M.: Adopting Model Driven Software Development in Industry- a Case Study at two Companies. In: MoDELS 2006. LNCS, vol. 4199, pp. 57–72. Springer, Heidelberg (2006)
27. Trask, B., Paniscotti, D., Roman, A., Bhanot, V.: Using Model-Driven Engineering to Complement Software Product Line Engineering in Developing Software Defined Radio Components and Applications. In: ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA 2006), pp. 846–853 (2006)
28. Ulrich, A., Petrenko, A.: Reverse Engineering Models from Traces to Validate Distributed Systems- an Industrial Case study. In: Akehurst, D.H., Vogel, R., Paige, R.F. (eds.) ECMDA-FA 2007. LNCS, vol. 4530, pp. 185–193. Springer, Heidelberg (2007)
29. Weigert, T., Weil, F.: Practical Experiences in Using Model-Driven Engineering to Develop Trustworthy Computing Systems. In: IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC 2006), pp. 208–217 (2006)
30. Wegener, H.: Agility in Model-Driven Software Development? Implications for Organization, Process, and Architecture (2002), <http://www.softmetaware.com/oopsla2002/wegenerh.pdf>
31. Wegener, H.: Balancing Simplicity and Expressiveness: Designing Domain-Specific Models for the Reinsurance Industry. In: 4th OOPSLA Workshop on Domain Specific Modeling (DSM 2004), p. 12 (2004)

Author Index

- Akehurst, David 137
Asadi, Mohsen 419
- Barbier, Franck 338
Becker, Steffen 169
Belaunde, Mariano 393
Bender, Darlam Fabio 121
Berbers, Yolande 17, 231
Bercovici, Avivit 381
Berthomieu, Bernard 121
Blaschek, José Roberto 110
Bork, Manuel 33
Braga, Christiano 326
Brown, John 369
- Carrington, David 349
Champeau, Joel 361
Chevillat, Cédric 349
Clavel, Manuel 326
Combemale, Benoît 121
Crégut, Xavier 121
- da Silva, Viviane 326
de Souza, Jano Moreira 110
Dehlen, Vegard 432
Duchien, Laurence 48
Dustdar, Schahram 246
- Egea, Marina 326
Engels, Gregor 94
Espinazo-Pagán, Javier 185
- Falcarin, Paolo 393
Farines, Jean Marie 121
Fondement, Frédéric 200
Fournier, Fabiana 381
Fritzsche, Christoph 369
Fritzsche, Mathias 369
- García-Molina, Jesús 185
Geiger, Leif 33
Gilani, Wasif 369
Goknil, Arda 310
Goldschmidt, Thomas 169
- Gotzhein, Reinhard 278
Grønmo, Roy 262
- Haugen, Øystein 215
Hoeltzener, Brigitte 361
Holmes, Ta'ïd 246
Hovsepyan, Aram 231
Howells, Gareth 137
- Iqbal, Muhammad Zohaib Z. 79
- Jaozafy, Fabre 361
Jézéquel, Jean-Marc 361
Joosen, Wouter 231
- Kilpatrick, Peter 369
Kleppe, Anneke 94
Kolovos, Dimitrios S. 1
Krogdahl, Stein 262
Kuhn, Thomas 278
Kurtev, Ivan 310
- Lima, Divany Gomes 110
- Malik, Zafar I. 79
Marchalot, Gabriel 361
Marzullo, Fabio Perez 110
McDonald-Maier, Klaus 137
Menárguez, Marcos 185
Mohagheghi, Parastoo 432
Møller-Pedersen, Birger 262
Monperrus, Martin 361
- Noguera, Carlos 48
- Oldevik, Jon 215
- Paige, Richard F. 1
Polack, Fiona A.C. 1
Porto, Rodrigo Novo 110
- Ramsin, Raman 419
Rech, Jörg 406
Rensink, Arend 94
Rose, Louis M. 1
- Sadilek, Daniel A. 63, 294
Scheidgen, Markus 153

Schneider, Christian 33
Schulz-Gerlach, Immo 17
Semenyak, Maria 94
Soltenborn, Christian 94
Sørensen, Fredrik 262
Spence, Ivor 369
Spriestersbach, Axel 406
Strooper, Paul 349
Süß, Jörn Guy 349

Tran, Huy 246

Uhl, Axel 169

Van Baelen, Stefan 231
van den Berg, Klaas 310

Vanhooff, Bert 17
Vernadat, François 121
von Pilgrim, Jens 17

Wachsmuth, Guido 63
Waheed, Tabinda 79
Wecker, Alan J. 381
Wehrheim, Heike 94
Weißleder, Stephan 294
Wildman, Luke 349
Wood, Stephen 137

Zdun, Uwe 246
Zündorf, Albert 33