Richard F. Paige
Alan Hartman
Arend Rensink (Eds.)

Model Driven Architecture – Foundations and Applications

LNCS 5562

5th European Conference, ECMDA-FA 2009
Enschede, The Netherlands, June 2009
Proceedings

Springer

Lecture Notes in Computer Science 5562

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

Richard F. Paige Alan Hartman
Arend Rensink (Eds.)

Model Driven Architecture - Foundations and Applications

5th European Conference, ECMDA-FA 2009
Enschede, The Netherlands, June 23-26, 2009
Proceedings

Springer

Volume Editors

Richard F. Paige
University of York
Department of Computer Science
Heslington, York, U.K.
E-mail: paige@cs.york.ac.uk

Alan Hartman
IBM India Research Laboratory
Bangalore, India
E-mail: alan.hartman.gm@gmail.com

Arend Rensink
University of Twente
Department of Computer Science
Enschede, The Netherlands
E-mail: rensink@cs.utwente.nl

Library of Congress Control Number: Applied for

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 3-642-02673-7 Springer Berlin Heidelberg New York
ISBN-13 978-3-642-02673-7 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 12706525 06/3180 5 4 3 2 1 0

Preface

The fifth edition of the European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009) was dedicated to furthering the state of knowledge and fostering the industrialization of Model-Driven Architecture (MDA) and Model-Driven Engineering (MDE). MDA is an initiative proposed by the Object Management Group for platform-generic systems development; MDA is one of a class of approaches under the umbrella of MDE. MDE and MDA promote the use of models in the specification, design, analysis, synthesis, deployment, and evolution of complex software systems.

It is a pleasure to be able to introduce the proceedings of ECMDA-FA 2009. ECMDA-FA 2009 addressed various MDA areas including model transformations, modelling language issues, modelling of behavior and time, traceability and scalability, model-based embedded systems engineering, and the application of model-driven development to IT and networking systems.

ECMDA-FA 2009 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 and MDE. Particularly, it is one of the few venues that engages both leading academic researchers and industry practitioners, with the intent of creating synergies.

The Program Committee accepted, with the help of additional reviewers, research papers and industry papers for ECMDA-FA 2009. We received 72 submissions. Of these, a total of 23 were accepted: 16 research papers and 7 industrial papers. We thank the Program Committee for their thorough reviews and detailed discussion during the selection process.

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 MDE/MDA community led to a successful conference: ECMDA-FA 2009 obtained excellent results in terms of submissions, program size, and attendance.

The Steering Committee members helped with various issues and we enjoyed continuous and constructive interactions with them. Arend Rensink, the Conference Chair, led a highly responsive and very organized local team. In addition, ECMDA-FA 2009 was complemented by a varied and successful set of workshops and tutorials: for this we would like to thank the Workshops and Tutorials Chair, Luís Ferreira Pires.

We would like to thank the University of Twente, MODELPLEX, the Centre for Telematics and Information Technology (CTIT) and the European Association of Software Science and Technology (EASST) for their support.

Finally, we would like to thank all the authors who spent valuable time in preparing and submitting papers to ECMDA-FA 2009, our keynote speakers – David Harel, Tim Trew, Akira Ohata and Laurent Balmelli – and the sponsors of ECMDA-FA 2009.

June 2009

Richard Paige
Alan Hartman

Organization

ECMDA-FA 2009 was organized by the University of Twente, Enschede, The Netherlands, with the support of the IBM Corporation and the Software and Systems Modelling Team at the University of York, UK.

Conference Committee

Conference Chair	Arend Rensink (Twente, The Netherlands)
Program Chairs	Richard Paige (University of York, UK)
	Alan Hartman (IBM, India)
Workshop and Tutorial Chair	Luís Ferreira Pires (Twente, The Netherlands)
Tools and Demos Chair	Regis Vogel (IHG, USA)
Organization Chair	Ivan Kurtev (Twente, The Netherlands)
Publicity Chair	Andrey Sadovykh (Softeam, France)

Steering Committee

Terry Bailey	Vicinay Cadenas, Spain
Philippe Desfray	Softeam, France
Alan Hartman	IBM, India
Richard Paige	University of York, UK
Arend Rensink	University of Twente, The Netherlands
Andy Schürr	T.U. Darmstadt, Germany
Regis Vogel	IHG, USA

Program Committee

Jan Aagedal	Telenor
David Akehurst	Thales
Terry Bailey	Vicinay Cadenas
Mariano Belaunde	France Telecom
Xavier Blanc	Universite de Paris-6
Marc Born	IKV
Phil Brooke	University of Teesside
Zhen Ru Dai	Philips
Jean-Luc De Keyser	LIFL
Miguel De Miguel	Polytechnical University of Madrid
Birgit Demuth	Technical University Dresden
Juergen Dingel	Queens University, Canada
Gregor Engels	University of Paderborn
Miguel Angel Fernandez	Telefonica

Luís Ferreira Pires	University of Twente
David Frankel	SAP
Mathias Fritzsche	SAP
Alan Hartman	IBM
Andreas Hoffmann	Fraunhofer Fokus
Frederic Jouault	INRIA and EMN
Gabor Karsai	Vanderbilt University
Olaf Kath	IKV
Ingolf Krueger	University of California at San Diego
Ivan Kurtev	University of Twente
Ralf Laemmel	University of Koblenz
Tiziana Margaria	University of Potsdam
Erhan Mengusoglu	IBM
Parastoo Mohagheghi	SINTEF
Nanjagud Narendra	IBM
Tor Neple	SINTEF
Bjorn Nordmoen	Western Geco
Richard Paige	University of York
Christoph Pohl	SAP
Arend Rensink	University of Twente
Laurent Rioux	Thales
Tom Ritter	Fraunhofer Fokus
Julia Rubin	IBM
Bernhard Rumpe	RTWH Aachen
Andrey Sadovykh	Softeam
Houari Sahraoui	University of Montreal
Ina Schieferdecker	Technical University Berlin
Doug Schmidt	Vanderbilt University
Andy Schürr	Darmstadt University of Technology
Bran Selic	Malina
Bikram Sengupta	IBM
Alin Stefanescu	SAP
Juha-Pekka Tolvanen	Metacase
Tim Trew	NXP
Andreas Ulrich	Siemens
Pieter van Gorp	University of Eindhoven
Marten van Sinderen	University of Twente
Regis Vogel	IHG
Jos Warmer	Ordina
Jules White	Vanderbilt University
Steffen Zschaler	University of Lancaster

External Reviewers

R. Bendraou
L. Daniele
B. Demchak
C. Dumoulin
A. Etien
D. Kolovos
M. Meisinger
M. Menarini
J.-M. Mottu
A. Muller
J. Polowinski
S. Richly
L. Santos
D. Zhou

Table of Contents

Applications

Creating Embedded Platforms with MDA: Where's the Sweet Spot?

Tim Trew

NXP Semiconductors, Eindhoven, The Netherlands
Tim.Trew@nxp.com

Abstract. MDA is often promoted to enable the portability of applications across platforms, but what of the development of platforms themselves? Today's consumer electronics products, such as TVs and mobile phones are based on complex integrated circuits to support their many functions. Cost and power consumption is crucial and semiconductor suppliers have to deliver substantial amounts of software to deliver the best performance from their hardware designs. This low-level software might be exposed to customers as device drivers for the operating system of their choice, and their development gives a different perspective on "platform specific models". Given the sensitivity of this software to changes in both customer requirements and their overall software architecture, and to changes in the underlying hardware, software development and maintenance effort becomes an increasing proportion of the overall engineering effort. The presentation will describe the software development challenges faced by a semiconductor supplier and some of our experiments in meeting these challenges with MDA. It will reveal the technical limitations encountered and the greater problems of deploying unfamiliar technology across large development teams. This guides us in the hunt for the "sweet spot", where MDA delivers benefits with an acceptable learning curve and project risk.

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, p. 1, 2009.

Comparison of Three Model Transformation Languages

Roy Grønmo[1,2], Birger Møller-Pedersen[1], and Gøran K. Olsen[2]

[1] Department of Informatics, University of Oslo, Norway
[2] SINTEF Information and Communication Technology, Oslo, Norway
{roygr,birger}@ifi.uio.no, goran.k.olsen@sintef.no

Abstract. In this paper we compare three model transformation languages: 1) Concrete syntax-based graph transformation (CGT) which is our emerging model transformation language, 2) Attributed Graph Grammar (AGG) representing traditional graph transformation, and 3) Atlas Transformation Language (ATL) representing model transformation. Our case study is a fairly complicated refactoring of UML activity models. The case study shows that CGT rules are more concise and requires considerably less effort from the modeler, than with AGG and ATL. With AGG and ATL, the transformation modeler needs access to and knowledge of the metamodel and the representation in the abstract syntax. In CGT rules on the other hand, the transformation modeler can concentrate on the familiar concrete syntax of the source and target languages.

1 Introduction

In model-driven engineering, the graphical models are the primary assets, and model-to-model transformations define mappings between models. The leading model-to-model transformation languages, such as Atlas Transformation Language (ATL) [6], are textual-based programming languages even though the source and target models are graphical models. It is a paradox that the model-driven community promotes the usage of models instead of textual code, while the same community dominantly uses textual code to define the model transformations. Model-to-model transformation languages can handle arbitrary source and target modeling languages, as long as they can be defined in a highly generic metamodeling language, such as Meta-Object Facility [9].

As opposed to ATL, graph transformations (such as Attributed Graph Grammar (AGG) [16]) provide graphical languages to define model-to-model transformations. Graph transformations are defined upon metamodel elements and visualized with a generic layout, called *abstract syntax*, where nodes are visualized as rectangles and edges as directed arrows.

The concrete syntax of a modeling language uses a tailored visualization with icons and rendering rules depending on the element types. Concrete syntax-based graph transformation (CGT) is our own emerging model-to-model transformation language that uses graph transformation principles, and where the rules

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, pp. 2–17, 2009.

are defined with concrete syntax. In this paper we investigate one configuration of CGT, with UML 2 activity models [11] as the source and target language (described previously in [3]). In general CGT can be configured with a different source and target language, and is thus comparable with the general model-to-model transformation languages ATL and AGG.

We illustrate the benefits of CGT by investigating a complicated model refactoring problem, which we will refer to as *removeGoto*. The refactoring task is to translate an activity model with arbitrary goto-like control flow into an activity model with structured loops (the refactoring task is taken from Koehler et al. [7]). We have implemented three alternative solutions for the removeGoto task using: 1) CGT, 2) AGG, and 3) ATL. Our findings can be summarized as follows:

- CGT can handle complex model transformations such as the removeGoto problem.
- The CGT solution to the removeGoto problem is more concise and was made with considerably less effort than the corresponding solutions in AGG and ATL.

2 Remove Unstructrured Cycles

Our running example is a business process model of a Web-based shopping application also taken from Koehler et al. [7]. The example is modeled with UML 2 activity models and we only show a submodel of the full business process model. The first model in Figure 1 (labeled `1`) shows our source model of the transformation, and represents a business process model with four activities. A Web shopper is allowed to select items (`Select` activity), configure the chosen items (`Configure` activity), put chosen items into the shopping cart (`Put` activity), and to finalize the shopping by leaving with an empty cart or with items to buy (`Finish` activity). Each control flow between two activities has a two letter guard that reflects the user choice to move from one activity to the next. The two letters are the first letters in the involved activity names.

The transformation task explained below requires that there is no explicit parallelism (no forks), and no implicit parallelism resulting from multiple outgoing control flow edges from the same activity. With no parallelism we can simplify the model, as we have in the figure, by not using explicit decision nodes and interpret multiple outgoing control flow as XOR-behavior. This interpretation is different from the activity model semantics, but is unproblematic since a complete transformation would isolate this simplification to the intermediate models. All the three transformation languages benefit from the simplification.

There are several approaches where the business process models are used to automatically generate BPEL code [13,12] that can be used to execute the business process in a BPEL engine. However, BPEL does not support unstructured cycles, meaning that we need to remove all the unstructured cycles of the activity model before generating BPEL.

In an *unstructured cycle* there is more than one entry or exit point into or out of the cycle. For instance, model 1 in Figure 1 contains the cycle

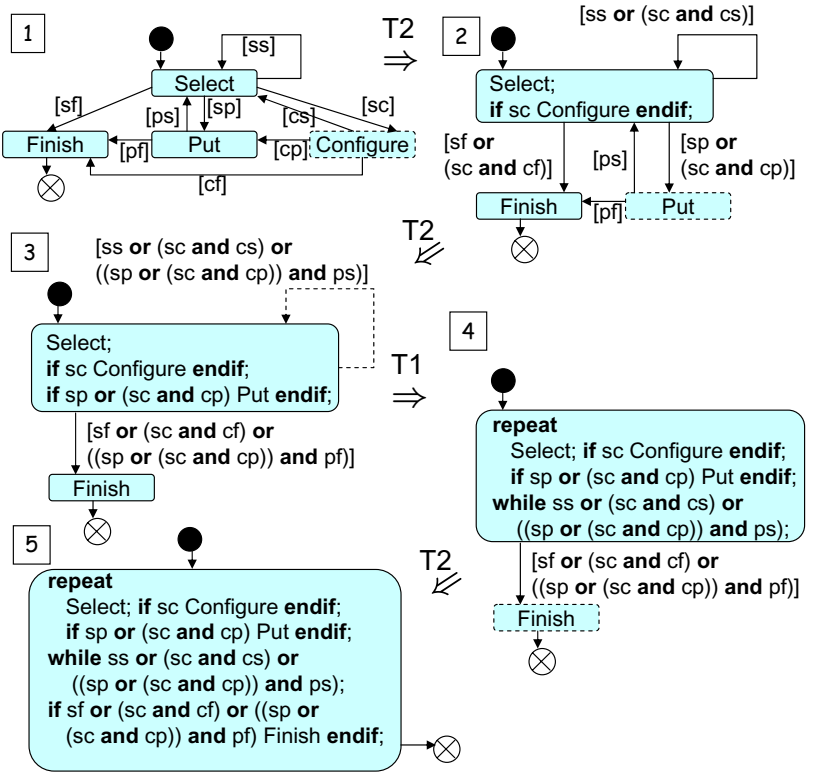


Fig. 1. From unstructured cycles to structured loops

`Select-Configure-Put-Select` which can be exited to the `Finish` activity from all three activities in the cycle. Another example is the cycle `Select-Put-Select` which can be entered from both the initial node and the `Configure` activity, and exited from both activities in the cycle.

Forcing the business process designer to not use unstructured cycles, but structured loops instead, is a heavy burden to put on the business process designer. Avoiding unstructured cycles, as in our starting model, is often a non-trivial and complex task.

Fortunately, an automatic transformation of graphs, like the shopping business process model (Figure 1), into a graph with structured loops (and no unstructured cycles), is well-known from compiler theory. Two tasks, called T1 and T2, can be applied non-deterministically until neither is applicable.

The tasks T1 and T2 will reduce the number of activities and control flow, while expanding the activity nodes from plain activities to become structured activities. We will use the name property of the activities to represent structured activities with arbitrarily many `repeat-while` and `if` expressions. At the end we have a single structured activity with no explicit control flow, only hidden

control flow in the name attribute value (the model with label 5 in Figure 1). It is straight forward to translate from the hidden control flow of `repeat-while` and `if` expressions in the name attribute of an activity, into plain activities with explicit control flow, by introducing decision and merge nodes. The end result is then guaranteed to be unstructured cycle free.

In order to apply a transformation based on the tasks T1 and T2, the source model must have the following three characteristics: 1) the model contains at least one unstructured cycle, 2) there is no parallelism, and 3) the model represents a *two-terminal region*. A source model is called a *two-terminal region* if it has a single initial node and a single final node. Our source model in Figure 1 satisfies all the three requirements. According to experience at IBM Zurich, subgraphs with all the three characteristics above, occur frequently in business process designs [7].

One possible transformation process with the tasks T1 and T2, over four steps, is shown in Figure 1. A model is displayed with dashed marking for elements that are replaced in the next transformation step, and $\overset{Tx}{\Longrightarrow}$ denotes the application of task Tx, where $x \in \{1, 2\}$.

The task T1 replaces cyclic control flow by `repeat-while` statements. The task T2 removes an activity with a single predecessor, moves its outgoing control flows to the predecessor activity, adds an `if` statement to the predecessor activity, and introduces a cyclic control flow to the predecessor activity if there is a "reverse" control flow from the successor to the predecessor. In the application of task T2 from model 1 to model 2, the `Configure` activity plays the role of a successor node with `Select` as the single predecessor activity. To ensure that there is at most one control flow in one direction between two activities, we make a combined control flow with `or` operators between the guards of the control flows.

In three following sections we describe an implementation of the tasks T1 and T2 by CGT rules (section 4), AGG rules (section 5), and ATL transformation modules (section 6). For all the three transformation languages our chosen strategy is to allow multiple control flows in the same direction between two activities in the intermediate models, while we define separate rules to combine multiple control flow edges into one. We do not consider nested activities due to limited space. The following section covers preliminary information about the three model transformation languages.

3 Preliminary

AGG uses graph transformation rules consisting of exactly one *left hand side* (LHS), a (possibly empty) set of *negative application conditions* (NACs), and exactly one *right hand side* (RHS). The LHS defines a subgraph for which we are looking for matches within the graph to be transformed. A NAC prevents application of a rule if the LHS combined with the NAC has a match. None of

the NACs can have a match in order to apply a rule. When a rule is matched by a LHS, then the matched LHS within the source graph is replaced by the RHS of the matched rule.

The dangling condition [4] ensures that a rule, involving node deletion, is only applied when there will be no dangling edges in the resulting graph. Identifiers, displayed with a number followed by a colon, e.g. `1:Activity`, are shown in the rule when there are shared elements between the LHS and the RHS/NACs. Elements that are shared between the LHS and the RHS are preserved by the rule. Elements where we have not displayed an identifier, occur either only in the LHS and will be deleted, or they occur only in the RHS and will be added.

CGT is basically the same as graph transformation, except that rules use concrete syntax instead of the abstract syntax. In addition CGT defines a *collection operator* which is explained below. In CGT, identifiers are displayed next to the elements(e.g. id=1), and attribute variables are prefixed with a question mark (e.g. ?guard1) and displayed in the same position as the attribute within the concrete syntax.

A collection operator can be used in a graph transformation rule to match a set of similar subgraphs. In the right part of Figure 2, we see how an activity model with a redundant decision node can be refactored by combining two guard expressions with an `and` operator [5,3]. With plain graph transformation it is not possible to express the removal of redundant decision nodes with a single rule. In the left part of Figure 2, a single CGT rule with the collection operator (dashed line frame) is sufficient to do the refactoring. The collection operator matches an arbitrary number of subgraphs (the `doA` and `doB` branches), removes a decision node and a merge node, and combines two guards for each subgraph match.

In the ATL code, transformation modules (hereafter called *modules* for short) declare the imported metamodel of the source and target models to be used in a transformation. *Rules* are used to implicitly match source elements and produce target elements, *lazy rules* are called explicitly to produce target elements based on source elements, and *helpers* represent user-defined functions with return values that does not produce target elements. ATL is built around the Object Constraint Language (OCL) [10] with some additional predefined functions.

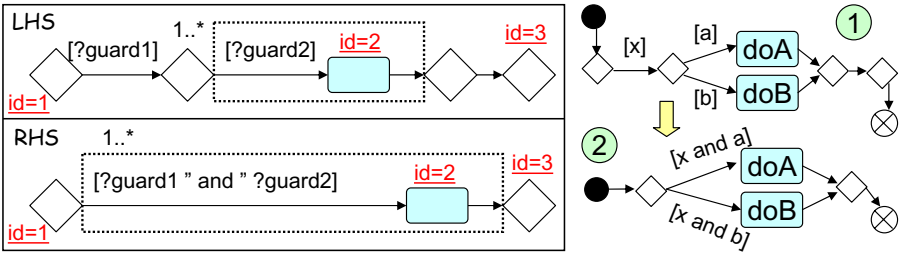


Fig. 2. Removing redundant decision nodes

4 RemoveGOTO by CGT

In Figure 3 we have defined five rules to simulate the tasks T1 and T2. Since we use the concrete syntax to define the rules, they resemble the transformation steps from Figure 1.

A single rule is sufficient to simulate the task T1. The LHS expresses that we are looking for matches of arbitrary activities with a cyclic control flow. id=1 is an identifier of the matched activity, and `?guard` is an identifier of the guard value of the cyclic control flow. The NAC ensures that the matched activity has exactly one cyclic control flow. The RHS expresses the replacement of the LHS match, implicitly meaning that the cyclic control flow is removed, and the activity name is extended with a `repeat-while` expression.

To simulate the task T2, we define two rules depending on the node type(s) following the successor activity. Either the next node(s) is the final node or activity node(s). In both cases a "reverse" control flow going from the successor activity back to the predecessor activity shall result in a cyclic control flow of the predecessor activity, where the guards are combined with an `and` operator. A collection operator with cardinality 0..1 expresses that such a "reverse" control flow is either present or not.

For both T2 rules the predecessor activity name is extended by the same `if`-expression. For the `T2-NextIsFinal` rule the predecessor activity gets an

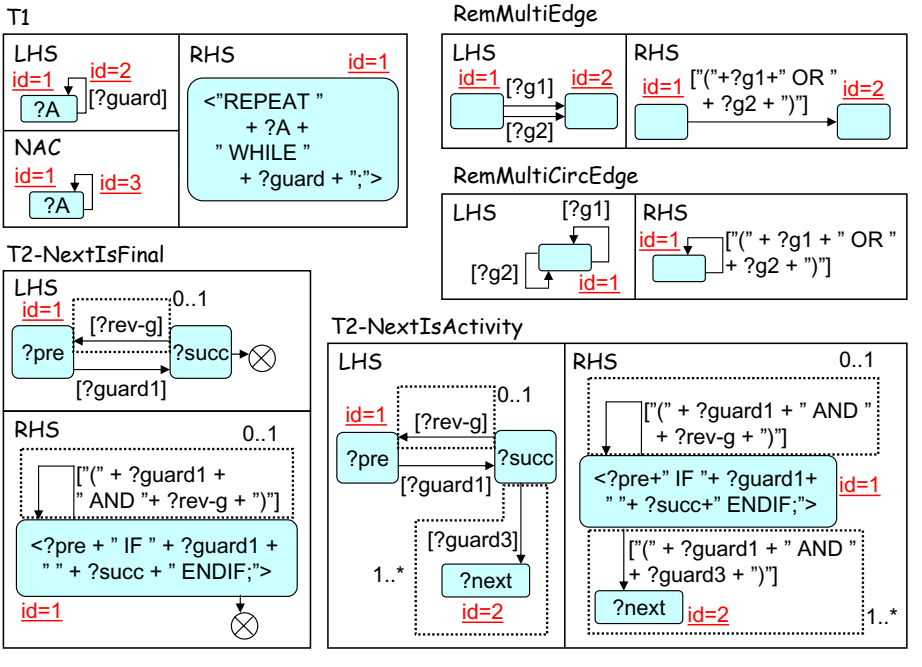


Fig. 3. *RemoveGoto:* Rules using concrete syntax (CGT)

outgoing control flow to the final node. The `T2-NextIsActivity` is a bit more complicated. Here we need to move each outgoing control flow of the successor activity over to the predecessor activity, and the guard of the new control flow is extended with an `and` operator between two successive guards. A collection operator with cardinality `1..*` expresses that there are arbitrarily many such outgoing control flows from the successor activity.

Note that we have not defined a NAC to ensure that the successor activity has exactly one predecessor activity, since this is ensured by the dangling condition. Otherwise an additional incoming control flow to the to-be-deleted successor activity would become a dangling edge.

Finally, we need two simple rules to define: 1) the merging of two cyclic control flows into one control flow (`RemMultiCircEdge`), and 2) the merging of two control flows in the same direction between two distinct activities (`RemMultiEdge`). The merged control flow uses an `or` operator to combine the guards of the joined control flows.

The two rules to merge multiple control flow rules should always be applied after each application of a T1 or a T2 rule. However, no specific control flow ordering of the rules is necessary due to 1) the dangling condition for the two T2 rules, and 2) the NAC of the T1 rule. The NAC of the T1 rule implies that the `RemMultiCircEdge` rule must be applied as long as possible first on the relevant activity, while the dangling condition on the to-be-deleted successor activity ensures that the `RemMultiEdge` rule is applied before the T2 rules.

5 RemoveGOTO by AGG

While the CGT transformations uses the concrete syntax, graph transformations (such as AGG) uses the abstract syntax. In addition, AGG does not have a collection operator. Thus, we get several rules for a single CGT rule using the collection operator. An automated mapping from a CGT rule to AGG rules is described in our earlier work [3].

We will use the AGG rules that result directly from our mapping of CGT rules into AGG rules. To be fair to AGG we manually investigated the generated rules to see if they could be optimized or further improved. No such improvements were found, and in fact the generated rules were fewer than a previous attempt where we coded the rules manually in AGG.

We get eight AGG rules (Figure 4 and Figure 5) corresponding to the five CGT rules. The three CGT rules without collection operators (Figure 5) are simply translated from concrete to abstract syntax.

The CGT rule, named `T2-NextIsActivity`, has two collection operators and is mapped to three transactional rules in AGG. The Iter-1 rule represents the `0..1` collection with the "reverse" control flow. The Iter-2 rule represents the `1..*` collection with the arbitrary number of outgoing control flow from the successor activity. The Final rule deletes the successor activity. For both the iteration rules we get autogenerated NACs to exclude matches when there are multiple predecessors for the successor activity.

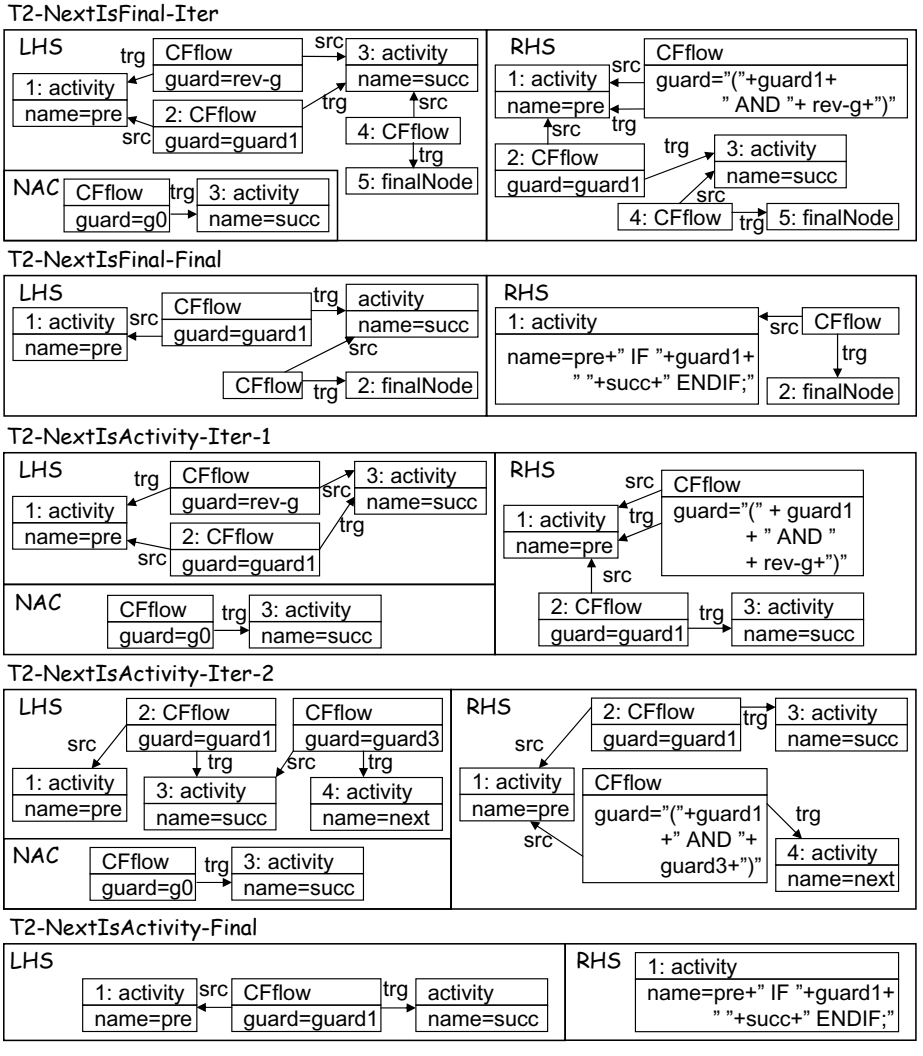


Fig. 4. *RemoveGoto:* Rules using abstract syntax - part 1 (AGG)

The CGT rule, named `T2-NextIsFinal`, has one collection operator and is mapped to two transactional rules in AGG. The `T2-NextIsFinal-Iter` rule will replace a "reverse" control flow by a cyclic control flow of the predecessor activity, and it gets a NAC to prevent multiple predecessors. The `T2-NextIsFinal-Final` rule deletes the successor activity.

We need additional Java code to control the rule application order. The set of rules (e.g. `T2-NextIsFinal-Iter` and `T2-NextIsFinal-Final`) corresponding to a single rule with collection operators shall be applied as one transactional group The Iter rules are applied first and at least the minimum cardinality

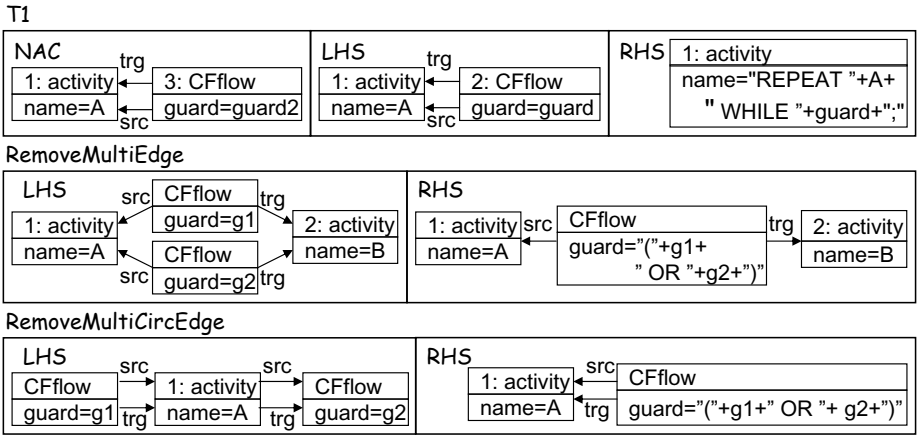


Fig. 5. *RemoveGoto:* Rules using abstract syntax - part 2 (AGG)

number of times, and as long as possible (or up to the maximum cardinality if it is different from $*$). The Final rule must then be applied.

6 RemoveGOTO by ATL

We use three ATL modules to accomplish the removeGOTO task: `T1`, `T2` and `RemMultiEdges`. The `T1`/`T2` module performs the task T1/T2 with the simplification also used by CGT and AGG to allow creation of multiple control flow in the same direction between two activities, and to allow creation of multiple circular control flows for the same activity. The `RemMultiEdges` module removes all such multiple control flow edges. Since the removeGOTO transformation is a refactoring task, the target model should keep all the unchanged parts from the source model. This is achieved by using the publicly available UML2Copy.atl module [15] that has rules to copy all UML source elements into the target model. Our three modules are then superimposed on the UML2Copy in order to override only the rules where things are changed.

Due to limited space we have made some abbreviations in the ATL code listings: `UML2!` is skipped as prefix on metamodel types, `Action` instead of `UML2!CallOperationAction`, `CFlow` instead of `UML2!ControlFlow`, `LString` instead of `UML2!LiteralString`, `Node` instead of `UML2!ControlNode`, and `thisModule` is skipped as prefix when calling helper functions. We have also left out a few rules, all the helpers implementation code in the `RemMultiEdges` and `T2` modules, and left out a large number of attributes that should be copied from the source to the target. We discuss some of these details after the code extracts of the modules are presented.

The `T1` module consists of one main rule, `Action`, where we produce a `repeat-while` statement in the Action name, for each circular control flow. The `T1` module will remove all circular control flow at once, and thus corresponds to

multiple applications of the T1 rule of CGT/AGG. A precondition for calling T1 is that there is at most one circular control flow of each Action element. This precondition is ensured by calling the RemMultiEdges module after each application of a T1 or T2 module.

Listing 1.1. ATL code extract from the T1 module

```
rule Action { from s: Action to t: Action (
  name <- if s.incoming->excludesAll(s.outgoing)
          then s.name
          else 'REPEAT ' + s.name + ' WHILE ' +
               loopGuard(s) + ';' endif) }

helper def: loopGuard(act: Action): String =
  act.incoming->asSet()->intersection(act.outgoing)->
       asSequence()->first().guard.value;
```

The RemMultiEdges module (not listed due to limited space) removes all multiple control flow occurrences, both circular and non-circular. In the main rule, Activity, we calculate all combinations of two activity nodes (n1,n2), where n1 and n2 may be the same node. If there exists a control flow edge between these two nodes, then a single, possibly combined control flow is produced from n1 to n2 by a call to the rule makeOneEdge. The combined control flow uses an or operator between the guards from the replaced multiple control flows.

The T2 module defines the Action rule to ignore successor nodes from the target model (guarded by not isSucc(s)). For the remaining Action elements, the name is kept unchanged if the Action source object is different from the predecessor node, and for predecessors the original name is replaced by an if-statement. If the "reverse" control flow exists from successor to predecessor, then an explicit call to the circEdge rule produces a circular control flow for the predecessor activity . All other outgoing control flows of the successor activity are moved over to the predecessor node, with extended guard values, by calls to the newNextEdge rule. Both the circEdge and newNextEdge lazy rules call on associated rules (circGuard and nextGuard) for combining guards by and operators.

The T2 module uses the helper functions for which the signatures are shown in the listing below. It is important to stress that we cannot safely apply the task T2 on multiple matches at the same time. We need to ensure that at most one node is treated as predecessor and at most one node is treated as successor. We use the predefined ATL method indexOf to get a unique index of each activity, and we choose only the successor with the lowest index. There is no rationale behind this choice except to ensure that we get only one successor activity, and then the rules also ensure that we get a single predecessor activity.

Since all the three modules deletes some of the source control flow, we need CFlow rules to override the default behavior of copying all the the CFlow. In the

Listing 1.2. ATL code extract from the T2 module

```
rule Activity { from s: Activity to t: Activity (
  name <- s.name, node <- s.node, edge <- s.edge,
  -- circular pre edge if edge from succ to pre
  edge <- s.edge->select(cf|isSucc(cf.source) and
                         role(cf.target) = 'pre')
                ->collect(cf| circEdge(cf)),
  -- outgoing edges of succ is moved to pre
  edge <- s.edge->select(cf|isSucc(cf.source)
          and role(cf.target) <> 'pre')
          ->collect(cf| newNextEdge(cf)))}

rule Action { from s:Action(not isSucc(s)) to t:Action (
  name <- if role(s) = 'pre' then s.name+' IF '
            +guardPreToSucc(s)+' '+succName(s)+' ENDIF;'
          else s.name endif ) }

lazy rule circEdge { from revCF: CFlow to t: CFlow (
  source <- revCF.target, target <- revCF.target,
  guard <- circGuard(revCF.target) ) }

lazy rule circGuard { from pre:Action to guard:LString (
  value <- '(' +  guardPreToSucc(pre)+' AND '
           + guardSuccToPre(pre) + ')')}

lazy rule newNextEdge { -- move edge from succ to pre
  from toNext : CFlow to t: CFlow (
    source<-toNext.source.incoming->first().source,-- pre
    target <-toNext.target, -- next
    guard <- if toNext.guard.oclIsUndefined()
             then OclUndefined
             else nextGuard(toNext) endif)}

lazy rule nextGuard { from next:CFlow to guard:LString (
  value<- '(' + next.source.incoming->first().guard.value
          + ' AND ' + next.guard.value + ')') }
```

listing below we show how the `T1` module adds a rule guard to ignore circular control flow. Similarly for the string value (called `LString`) of the corresponding control flow guard, we need to avoid producing target elements.

As a shortcut in our code we have assumed that all guard values are registered as literal strings, while in general there are a number of possibilities. While these are treated generically in CGT and AGG, we need additional rules (similar to `LString`) to handle all these other value types (e.g. integer, boolean, time).

Listing 1.3. Helpers from the T2 module

```
-- The role of a node is 'pre', 'succ' or 'other'.
helper def: role(a: Action): String = ...
-- match only the lowest indexed succ node
helper def: isSucc(a:Action): Boolean = ...
-- returns true for all candidate succ nodes
helper def: isWeakSucc(a:Action): Boolean = ...
helper def: guardPreToSucc(pre:Action): String = ...
helper def: succName(pre: Action): String = ...
helper def: guardSuccToPre(pre: Action): String = ...
-- each action element has a unique index
helper def: nodeIndex(a: Action): Integer = ...
helper def: noLowerIndexSucc(a: Action): Boolean = ...
```

For the module `RemMultiEdges` to combine multiple control flow edges into one, the guard of `CFlow` and associated `LString` is set to false. This is because no source control flow is preserved and all the resulting control flow is explicitly produced. The guard of the rule `CFlow` in the module `T2` ensures that control flow connected to the to-be-deleted successor activity is omitted in the target model.

Listing 1.4. CFlow and LString rules in the T1 module

```
rule CFlow{from s:CFlow (s.source <> s.target) to t:CFlow
    ( -- copy all attrs ) }

rule LString { from inGuard: LString
  (not (inGuard->parent().oclIsTypeOf(CFlow)) or
   (inGuard->refImmediateComposite()().source <>
    inGuard->refImmediateComposite()().target) )
  to outGuard: LString ( -- copy all attrs ) }
```

The ATL code listings are simplified because we need to copy all the source element properties (e.g. `Activity` has 46 properties and `ControlFlow` has 14 properties), just like `UML2Copy` already does. This could be avoided if there was a way to execute the existing body of the superimposed rule, but this is not possible. Inheritance can only be achieved on rules within the same transformation module. The same problem does not occur with CGT/AGG since unmatched elements and properties are preserved by default.

The three modules must be invoked in a controlled manner. The modules `T1` and `T2` must be applied as long as possible, but after each application of `T1` or `T2`, we have to call the module `RemMultiEdges` before we continue. This module application order is shown in Figure 6. The ATL transformation modules can be invoked from Java code, and it is trivial to translate the control flow in

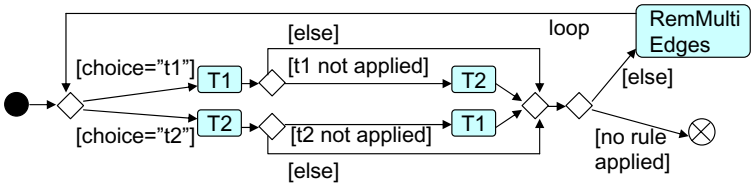


Fig. 6. Control flow of ATL transformations

Figure 6 into Java code by using `if` statements for the decision nodes, and a `do-while` statement for the outer loop. The data flow is omitted from the figure, but basically each transformation module takes an activity model as input and returns an activity model as output. In addition there will be some data objects involved to test if a transformation module has been applied or not. This can be achieved in many ways, for instance checking if the number of control flow edges are reduced by `T1`, or if the number of activities are reduced by `T2`.

7 Discussion

Our paper example can be compared with respect to the amount of code or models that a programmer or modeler needs to specify in order to make an executable solution. While the CGT solution is expressed with half a paper page of graphical models, the AGG solution needs one page of graphical models. In addition AGG needs rule application order code to ensure the transactional behavior of the iteration/final rules. The ATL solution (excluding the UML2Copy.atl) would use about 4,5 pages (160 code lines without empty lines and comments) of textual code without the rule application order code, and without the copying of unchanged attributes which would make the complete ATL solution several pages longer. So, CGT is much better than AGG, and AGG is much better than ATL with respect to the solution size.

The usage of the collection operator in the concrete syntax of CGT raises the level of abstraction compared to the collection free AGG rules. This is why we get fewer rules in CGT than in AGG. The ATL solution gets very large compared to CGT/AGG since we need: 1) to copy all the unchanged properties of elements that are modified, 2) several helper methods to ensure that the T2 module matches only one activity as a predecessor and only one activity as the successor for each application (otherwise the result may be corrupted).

CGT appears to be more intuitive since it uses the concrete syntax of activity models which is already familiar to the modeler. Even though AGG is graphical, it takes more time to understand what the diagrams express since they are defined on the abstract syntax. It is clearly a benefit for a modeler to define an activity model by using a concrete syntax-based editor. The same benefit applies when rules are defined as model extracts in LHS/RHS.

The transformation modeler does not need to know anything about the representation of the UML activity metamodel when defining CGT rules, while access to and knowledge of the metamodel and the abstract syntax is essential when using AGG or ATL. This largely increases the overall complexity of using AGG and ATL compared to CGT. While a control flow is simply drawn in CGT as within activity models, the modeler need to know how a control flow is represented in AGG and ATL. In AGG we represented a control flow with a node of type `CFlow`, and two outgoing edges labeled `src` and `trg` going to the source and target nodes of the control flow. In the UML 2 metamodel we used in ATL, the `incoming` and `outgoing` properties of the activity node provides the set of incoming and outgoing control flow references.

ATL has additional weaknesses compared to CGT and AGG for refactoring transformations like the paper example. While CGT and AGG defines only the changes to the source model, and preserves all the unchanged parts by default, this is very cumbersome in ATL. In the ATL 2006 version, a *refining mode* was introduced, with the intention to preserve unchanged parts. But it currently has limitations and does not behave as expected. We still have to define a copy module with default copying of all elements. With UML, one was publicly available (`UML2Copy.atl`), which saved a lot of effort, but this will generally not be the case. Upon this copy module, we need to superimpose our actual module. In ATL there is no way to change only a few properties of an element, and to keep the other attributes unchanged. All the properties must be explicitly listed. This is because we cannot use inheritance combined with superimposition.

With refactoring it is useful to apply a set of rules for as long as possible. This is directly supported by CGT and AGG, but not by ATL.

CGT is one configuration of concrete syntax-based graph transformation, and its implementation has been hard coded. Full support for concrete syntax-based graph transformation to support general model transformations requires more tool implementation and also some initial configuration where the user defines the relation between the concrete and abstract syntax of the source/target language. Thus, AGG and ATL have a benefit, compared to concrete syntax-based graph transformation, by having existing tools that support general model transformations.

We have developed a proof-of-concept Eclipse GMF-based [2] rule editor for the configuration of CGT where activity models is the source and target language [3]. The transformation from CGT to AGG rules has been implemented using the MOFScript language [8]. We have not implemented the transactional support needed to generally ensure a correct simulation of the collection operator. In the removeGOTO solution it was sufficient to add a few NACs, and to apply all the rules non-deterministically.

For the ATL code implementation of the removeGOTO solution, we did not implement the Java code to control rule application order, but tested several orders manually. The generated AGG rules, and the ATL rules were tested successfully on four different models including the source model of Figure 1.

8 Related Work

Strommer et al. [14] also aim at using the concrete syntax of the source and target languages to define model transformations. Our transformation definitions are completely defined upon the concrete syntax, while they generate a starting point that must be further modified and extended within the ATL language and the abstract syntax of the source and target.

The removeGOTO problem used in this paper is taken from Koehler et al. [7]. They have an OCL-based solution to the problem, which has many of the same drawbacks that ATL has. In addition pure OCL tends to be more complicated than ATL since the rule construct of ATL and additional functions is an improvement over pure OCL.

The TIGER tool [1] and the MATA tool [17] use concrete syntax in their transformation rules. These rules are then mapped to abstract syntax rules that are executed in a traditional graph transformation tool. As opposed to our approach, the focus with TIGER and MATA seems so far to be restricted to model transformations where the source and target languages are the same. Neither TIGER nor MATA support a similar concept as the collection operator.

9 Conclusions

In concrete syntax-based graph transformation, the transformation modeler can concentrate on rules directly within the familiar concrete syntaxes of the source and target modeling languages. With AGG and ATL, on the other hand, the transformation modeler must master several related languages: 1) the concrete syntax of source and target, 2) the metamodel of source and target, and 3) the abstract syntax of source and target.

This paper presents three solutions (in CGT, AGG, and ATL) to a fairly complicated refactoring example of activity models. The conclusion is that CGT in this case requires less effort and is the most concise solution of the three. In addition to the usage of concrete syntax, the usage of a collection operator is the reason why CGT outcompetes the other two. The collection operator can also be used on the abstract syntax of traditional graph transformation, which would reduce the disadvantage of using traditional graph transformation like AGG.

Concrete syntax-based graph transformation (with the collection operator) is at a higher level of abstraction than traditional graph transformation and model transformation. While this is a benefit for the users, it also requires more tools and infrastructure than is available today. As a simplified view, the missing infrastructure can be seen as a compiler that translates concrete syntax rules into abstract syntax rules. For each new combination of source and target, the user also needs to define the relationship between abstract and concrete syntax which is not needed in the traditional approaches. It is a future goal to implement full tool support for CGT.

Acknowledgment. The work reported in this paper has been funded by The Research Council of Norway, grant no. 167172/V30 (the SWAT project), and by the DiVA project grant no. 215412 (EU FP7 STREP).

References

1. Biermann, E., Ermel, C., Hurrelmann, J., Ehrig, K.: Flexible visualization of automatic simulation based on structured graph transformation. In: IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC (2008)
2. Eclipse Consortium. Eclipse Graphical Modeling Framework (GMF) (2007), http://www.eclipse.org/gmf
3. Grønmo, R., Møller-Pedersen, B.: Aspect Diagrams for UML Activity Models. In: Applications of Graph Transformations with Industrial Relevance. LNCS. Springer, Heidelberg (2008)
4. Habel, A., Müller, J., Plump, D.: Double-pushout graph transformation revisited. Mathematical Structures in Computer Science 11(5), 637–688 (2001)
5. Eder, J., Gruber, W., Pichler, H.: Transforming Workflow Graphs. In: Proceedings of the First Int. Conf. on Interoperability of Enterprise Software and Applications (INTEROP-ESA) (2005)
6. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
7. Koehler, J., Hauser, R., Sendall, S., Wahler, M.: Declarative techniques for model-driven business process integration. IBM Systems Journal 44(1) (2005)
8. Oldevik, J., Neple, T., Grønmo, R., Aagedal, 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)
9. OMG. OMG's MetaObject Facility, http://www.omg.org/mof/
10. OMG. UML 2.0 OCL Specification, OMG Adopted Specification ptc/03-10-14 (October 2003)
11. OMG. UML 2.0 Superstructure Specification, OMG Adopted Specification ptc/03-08-02 (August 2003)
12. Ouyang, C., Dumas, M., Breutel, S., ter Hofstede, A.H.M.: Translating Standard Process Models to BPEL. In: Dubois, E., Pohl, K. (eds.) CAiSE 2006. LNCS, vol. 4001, pp. 417–432. Springer, Heidelberg (2006)
13. Skogan, D., Grønmo, R., Solheim, I.: Web Service Composition in UML. In: IEEE Intl. Enterprise Distributed Object Computing Conf. (EDOC) (2004)
14. Strommer, M., Wimmer, M.: A framework for model transformation by-example: Concepts and tool support. In: Objects, Components, Models and Patterns (TOOLS). LNBIP. Springer, Heidelberg (2008)
15. System and Software Engineering Lab, Vrije Unversiteit Brussel, Belgium. MDE Case Studies, http://ssel.vub.ac.be/ssel/research:mdd:casestudies
16. Taentzer, G.: AGG: A Graph Transformation Environment for Modeling and Validation of Software. In: Applications of Graph Transformations with Industrial Relevance (AGTIVE) (2003)
17. Whittle, J., Jayaraman, P., Elkhodary, A., Moreira, A., Araújo, J.: MATA: A Unified Approach for Composing UML Aspect Models based on Graph Transformation. Transactions on AOSD - Special Issue on Aspects and Model-Driven Engineering (2008) (in press)

On the Use of Higher-Order Model Transformations

Massimo Tisi[1], Frédéric Jouault[2], Piero Fraternali[1], Stefano Ceri[1], and Jean Bézivin[2]

[1] Politecnico di Milano, Dipartimento di Elettronica e Informazione
Milano - Italy
{massimo.tisi,piero.fraternali,stefano.ceri}@polimi.it
[2] INRIA, Centre Rennes - Bretagne Atlantique
Nantes, France
{frederic.jouault,jean.bezivin}@inria.fr

Abstract. The level of maturity that has been reached by model transformation technologies is proved by the growing literature on transformation libraries that address an increasingly wide spectrum of applications.

With the success of the modeling and transformation paradigm, the need arises to address more complex applications that require a direct manipulation of model transformations.

The uniformity and flexibility of the model-driven paradigm allows this class of applications to make use of the same transformation infrastructure. This is possible because transformations can be translated into transformation models and given as objects to a different class of model transformations, called Higher-Order Transformations (HOT).

This paper provides an introduction to HOTs and a survey of the several application cases where their use is relevant. A number of possible future applications of HOTs is also proposed.

1 Introduction

The popularity of Model-Driven Engineering (MDE) is continuously growing and the reason behind this increasing success is mainly technological: a set of automation frameworks, built around model transformation technologies, are reaching a good level of maturity. This maturity is related to two important technological drivers. At first, common recognized formalisms (e.g. MOF, Ecore, KM3[23]) have allowed the explicit characterization of several metamodels; then more and more libraries of transformations have started to gather reusable model transformations expressed in declarative rule-based languages (e.g. QVT, ATL[24]).

The evolution of model-driven environments since its first steps can be outlined distinguishing three phases. In a first phase, early MDE tools were limited to assistance in drawing models, sometimes reverse-engineering them from existing code, and to generation of structural skeleton of programs from diagrams. Such limited approaches were relegating models to the role of mere program documentation, difficult to maintain, with a cost that was not clearly justified by a tangible improvement in software quality.

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, pp. 18–33, 2009.

The second phase saw an increasing success of model transformation technologies to automate forward engineering and several other software development activities. Automation is among the most appealing means to reduce costs and increase productivity. While the idea of automating the generation of a significant part of the program code accompanies MDE since its first steps, the success of code generation only started when it was able to produce code whose efficiency was comparable to hand-crafted code. Since that moment, the use of models in software development started to become much wider than their role as drivers of the implementation. Once models become an integral part of the software engineering process, there is no reason not to exploit the same transformation infrastructure to automate other tasks. This lead to the development of a vast library of transformations to accomplish several activities automatically, such as metrics evaluation, testing, generation of documentation.

Finally in a third phase, while models and transformations are still a central part of the software development process, they start to become also an integral part of the developed system. Model-based software systems appear, where models and model transformations are first-class elements of the runtime architecture, together with data structures and programs. In these systems, models are used to represent several heterogeneous types of information. They can be handled natively at runtime, and system logic can be represented by complex transformation workflows.

It is especially in this third phase that the idea of transformation manipulation naturally arises. As part of the developed system, transformations can be themselves generated and handled by model-driven development, exactly like traditional programs. While transformation manipulation can be performed by means of an independent methodology (e.g., program transformation, aspect orientation), the elegance of the model-driven paradigm allows again the reuse of the same transformation infrastructure. To achieve this objective, the concept of model transformation needs to be extended with that of transformation model [11]. The transformation is represented by a transformation model that has to conform to a transformation metamodel. Just as a normal model can be created, modified, augmented through a transformation, a transformation model can itself be instantiated, modified and so on. This uniformity is beneficial in several ways: especially it allows reusing tools and methods, and it creates a framework that can in theory be applied recursively (since transformations of transformations can be transformed themselves).

Once the boundary between development-time transformations and execution-time transformations is weakened, a wide set of application patterns appear that involve transformation models in the roles of both manipulation program and manipulated object. This paper provides a first survey and classification of these applications and the related transformation patterns.

The rest of the paper is organized as follows: Section 2 introduces definition and structure of higher-order transformations, i.e. transformations of transformations; Section 3 presents the outline of the classification and the main transformation types; Section 4, 5, 6 and 7 describe the four main areas in which

the survey is divided, namely transformation synthesis, analysis, composition and modification; Section 8 suggests other applications that can be addressed in future by higher-order transformations; Section 9 draws the conclusions.

2 Higher-Order Transformations

An essential prerequisite for fully exploiting the power of transformations is their treatment as objects. This demands the representation of the transformation as a model conforming to a transformation metamodel.

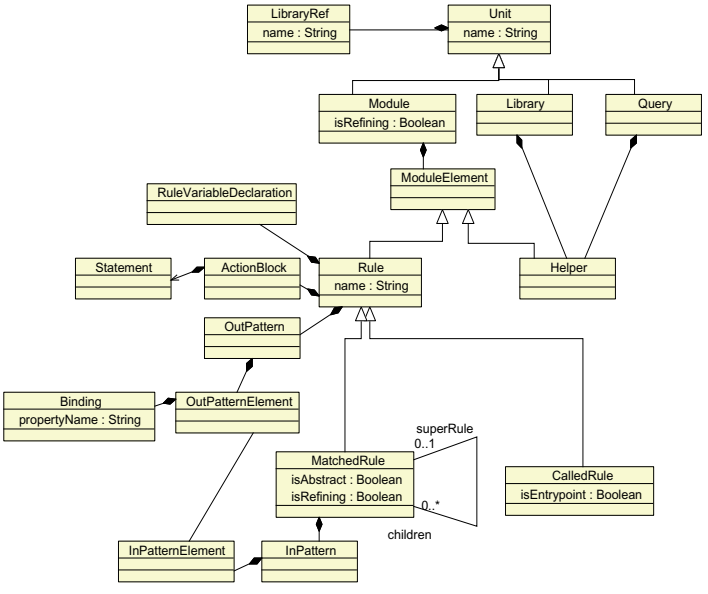


Fig. 1. Simplified version of the ATL Metamodel

Not all transformation frameworks provide a transformation metamodel. In this work we will mainly refer to the AmmA framework [27] that contains a mature implementation of the ATL transformation language. Within AmmA an ATL transformation is itself a model, conforming to the ATL metamodel. Figure 1 shows the main classes of the ATL metamodel. Besides the shown elements like the central classes of *Rule*, *Helper*, *InPattern*, and *OutPattern*, the ATL metamodel also incorporates the whole OCL metamodel to represent expressions on the manipulated models.

Once the representation of a transformation as a transformation model is available, a HOT can be defined as follows:

Definition 1 (Higher-order transformation). *A higher-order transformation is a model transformation such that its input and/or output models are themselves transformation models.*

According to this definition HOTs either take a transformation model as input, produce a transformation model as output, or both. An example of a HOT in the AmmA framework is shown in Figure 2. This example reads and writes a transformation, e.g. with the purpose of performing a refactoring. The three operations shown as large arrows at level M1 (Models) are:

- *TCS Injection.* The textual representation of the transformation rules is read and translated into a model representation. This example uses for this step a generic program that is parametrized with a model representing the concrete syntax of the ATL language. The Textual Concrete Syntax is described in AmmA by means of the TCS formalism [22]. The generated model is an instance of the ATL metamodel (Figure 1).
- *HOT.* The transformation model is the input of a model transformation that produces another transformation model. The input, output and HOT transformation models are all conforming to the same ATL metamodel.
- *TCS Extraction.* Finally an extraction is performed to serialize the output transformation model into a textual transformation program.

Note that the injection and extraction operations are not always used during a HOT. For instance, the source transformation model may come from a previous step, and already be in the form of a model. Similarly, the target transformation model is sometimes reused as a model without an immediate serialization.

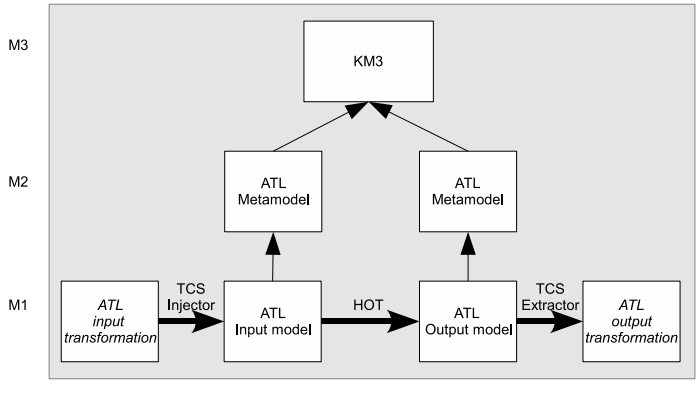


Fig. 2. A typical example of Higher-Order Transformation

3 A Survey of Higher-Order Transformations

The next sections are an overview of the literature on model transformations that involve the use of HOTs for specific tasks.

This survey comprises all publications known to us that are related to HOTs in the ATL language. ATL appears to be the preferred language for HOTs development to date, and we were able to gather a set of 44 transformations from

previous work. We add to this set other notable examples of HOTs in other frameworks. Most of the described model transformations are freely available and highly reusable. They constitute a first comprehensive library of HOTs.

The survey is organized in a two-level hierarchy. The first level is focused on the identification of base *transformation patterns*, each of them representing the usage pattern of a given class of HOTs. In the second level, each one of the base groups is further divided by considering different variants of the patterns.

We identified four transformation patterns that are shown in Figure 3. These patterns include the following models: the HOT, its input and output models and the input and output models of the transformations that are handled by the HOT. Models are included together with their respective metamodel and they are linked by the following associations: the *conforms-to* relationship between a model and its metamodel (represented as a thin arrow), and the *transforms* relationship between a transformation and its output and input models (represented as thick arrows). The *transforms* arrows of the HOTs are shown in a darker shade of gray. The four base patterns are:

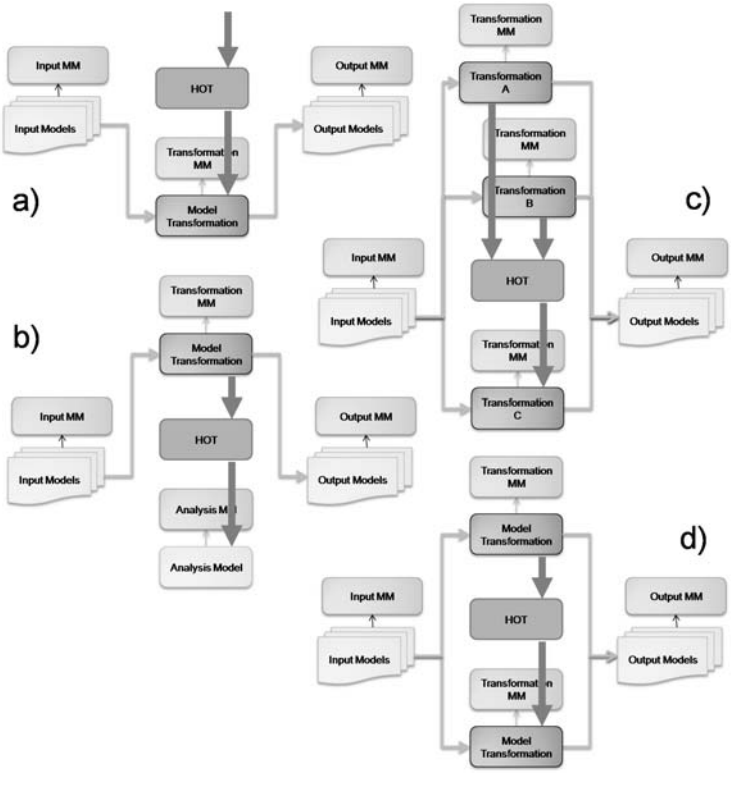


Fig. 3. Base HOT patterns: a) Transformation synthesis, b) Transformation analysis, c) Transformation composition (decomposition not shown), d) Transformation modification

Transformation Synthesis. (Section 4) is the common pattern for HOTs that generate transformations from different information sources. These HOTs are defined by two conditions: 1) the output model is a transformation; 2) the input models, if present, are not transformations;

Transformation Analysis. (Section 5) is the pattern for HOTs that take transformations as input, to generate different kinds of data, in the form of output models. More precisely: 1) they have a transformation as input model; 2) they do not have transformations as output models.

Transformation (De)composition. (Section 6) is the pattern for HOTs that take multiple transformations as input (composition) and/or output (decomposition). Three conditions define these HOTs: 1) at least one of the input models must be a transformation; 2) at least one of the output models must be a transformation; 3) the input and/or output models must contain more than one transformation.

Transformation Modification. (detailed in Section 7) is the pattern for HOTs that take a transformation as input and generate a modified version of the same transformation. These HOTs must have: 1) one transformation as input; 2) one transformation as output.

Additional input or output models are allowed for all the previous HOTs, with the condition that they are not transformation models.

4 Transformation Synthesis

All the cases of transformation synthesis in our survey can be inserted in one of the following sub-classes: 1) mapping implementation, i.e. the generation of an executable version of an abstract mapping; 2) generic metamodel, i.e. the construction of transformations that are generic with respect to the input or output metamodel.

4.1 Mapping Implementation

The semantics of a transformation language, while representing an abstract mapping between metamodels, needs to be concrete enough to allow a direct execution in a transformation engine. For a set of practical applications this level of abstraction is not sufficient and a higher-level specification is preferred. A more abstract mapping has advantages in terms of readability and manageability and can provide useful features that may not be in the transformation language, such as bi-directionality. However the non-executable representation needs to be translated into an executable transformation. This is a typical application of HOTs. We refer to this class of HOTs with the term *mapping implementation*, since the abstract representation can be considered as the higher-level specification of a mapping that needs to be implemented as a transformation.

In [15] a practical use case for mapping implementation HOTs is deeply studied. The motivating scenario is enabling tool interoperability between different bug-tracking systems. Two autonomous software development companies that

need to collaborate rely on different bug-tracking systems and they need an import/export mechanism between their bug-tracking metadata. A declarative correspondence model is semi-automatically generated by the analysis of the metamodels used by the two systems. The correspondence model represents a set of relationships between model elements. To address the problem of the possibly infinite kinds of relationships between source and target elements, the authors use an extension mechanism. The supported links are classified in three major groups: *similarity expressions* (e.g., equivalence of model elements), *mapping expressions* (e.g., one-to-many, many-to-one or many-to-many relationships between source and target model elements) or *data-value expressions* that involve values of related attributes (e.g., the relationship between the values of the *status* attribute for a bug request in the two systems). Finally a HOT translates this representation in an executable transformation.

The same authors provide another example of this approach in [7]. Using AMW (i.e. a model that represents generic correspondences between models [29]) and an ATL HOT it is possible to generate two model transformations that translate a KM3 metamodel in the SQL DDL and vice versa. In this example the HOT takes as input a AMW weaving model (i.e. a correspondence model) between a KM3 metamodel and a SQL DDL metamodel. The weaving model defines a correspondence between metamodel classes and tables on a database. This model is then used to produce the two implementation transformations for bidirectional translation.

HOTs to translate an AMW correspondence model into an implementation are provided also in [1]. This example contains an extension of the core weaving metamodel to specify correspondences from a metamodel with flat structures and foreign keys relationships (as in relational databases) to a metamodel that contains nested structures (as in XML). The two proposed HOTs generate respectively an ATL and an XSLT executable version of the AMW specification.

The same general schema is followed by:

- [25] that addresses the issue of adapting a model to an evolving metamodel;
- DUALLy [28], an automated framework that allows architectural languages and tools interoperability;
- the MML2MMR transformation in [20] for studying the integration of DoDAF [5] metamodels and models.

In all these cases an AMW mapping that correlates two different metamodels is translated into two transformations at model level (one for each direction of the mapping). Each AMW link corresponds to one or more rules or helpers in the generated transformations. [16] provides an abstract specification of several transformations. Among them, the *patchgen* transformation is the specification of the previous transformation group.

[14] presents an alternate representation for differences between models that conform to the same metamodel. In this proposal the difference model conforms to a new metamodel that is derived from the input metamodel using a model transformation. This transformation specializes the metaclasses of the input metamodel in three subclasses: for each class *ClassName* an 1) *AddedClassName*,

a 2) *DeletedClassName* and a 3) *ModifiedClassName* are introduced. The instances of these metaclasses represent all the differences between the analyzed models. In this case too, starting from the difference representation, the authors use a HOT to derive an ATL implementation (they call this implementation *difference animation*).

Ecore2RDF [19] is a bridge between the EMF and SemanticWeb technical spaces. A custom mapping metamodel, which extends the whole ATL metamodel by adding novel constructs, is the core formalism of this solution. Two HOTs generate the two directions of the mapping.

Mapping implementations are common also outside the ATL environment. For instance WebRatio[8] is a modeling environment for the WebML domain-specific language that allows the generation of a Web application from structure and navigation models. WebRatio includes a tool called EasyStyle to generate the presentation of Web pages. EasyStyle is based on a XSLT HOT which takes as input an HTML template annotated with custom tags, i.e. placeholders for the content elements of the page. The XSLT HOT translates this HTML file into another XSLT transformation that generate pages conforming to that template. This application is not different from the previous ones, since the HTML template can be considered as a mapping that relates the elements of the input model (the Web application model) with the elements of the output model (the page layout model).

4.2 Generic Metamodels

Several application cases require the development of generic model transformations that need to take as input or output a metamodel that is not known a priori. The typical solution in these cases is using HOTs for the on-the-fly generation of those model transformations that could not be easily developed manually with a sufficient generality.

For instance the proposal in [18] includes a HOT that takes a KM3 metamodel as source model and generates a transformation for conflict detection. The output transformation takes two source models (an implementation and an architecture model) that conform to the same metamodel and generates a target model that is the union of the two source models with each of the model elements labeled convergent, divergent or absent depending on their occurrence in one or both of the architecture and implementation models.

The need to use a HOT in this case is related to the limitations of the ATL language. The expressing power of ATL (at least in its declarative form) does not allow the direct development of a generic conformance checking transformation. For instance a specific input and output metamodel needs to be specified at development time. The reflectivity features of declarative ATL are not enough to outflank the problem (an alternative using imperative ATL is possible but not painless). This limitation is shared by several other transformation languages.

An analogous problem is presented in [6], where the HOT is used to generate a generic copier for models conforming to a metamodel that is only known at runtime. [12] presents an alternative implementation of the model copier as a

HOT based on the transformations in [36] and applied to product lines. In this case a so-called *configuration* is obtained by a copier transformation that selects only a subset of the elements of an extremely generic default model. To keep the genericity with respect to the configuration metamodel, a HOT dynamically generates the copier.

In [10] the authors develop a set of transformations to bridge models expressed in the Microsoft DSL framework to the Eclipse Modeling Framework. A first sequence of transformations performs the bridging at M2 level, translating metamodels between the two frameworks. A second set deals with the models. The last step of this process is a transformation that needs to build models conforming to the metamodel generated in the first step. Knowledge about this metamodel is available only at runtime and can not be embedded into the last transformation. In the proposed solution the last transformation is dynamically generated from a HOT that *instantiates* some general ATL rules for the elements of the input metamodels.

Generating test cases for model transformations is a task that can be easily performed by exploiting the syntactic description of the input and output domains of the transformation, given by the input and output metamodels. This kind of approach to test set generation is referred to as black-box generation, since the process does not involve an analysis of the internal structure of the transformation under testing. In [9] the last step of the test data generation process is the *MM2TM* transformation for the generation of test models according to a test criterion. In [9] the generation of models is guided by *model fragments* that are particular model chunks identified in previous steps: each model fragment should appear at least once in generated test models. To be implemented as a generic model transformation, applicable to any input metamodel, the *MM2TM* transformation has to be dynamically generated for each input metamodel. The generating HOT contains the logic of the coverage criteria (i.e., a different HOT has to be developed if we want to use a different criteria).

5 Transformation Analysis

The generation of an output model that represents a particular analysis of an input transformation is inherently a HOT.

An example of these HOTs is given in [4]. The ATL to Problem use case describes a transformation from an ATL model into a Problem model. The generated Problem model conforms to a single-class Problem metamodel and contains the list of non-structural errors and warnings that have been identified within the input ATL model. The transformation assumes the input ATL model is structurally correct, i.e. it conforms to the ATL metamodel.

[26] shows a complex example of transformation analysis in the GReAT framework. The HOT in this case reads a model transformation to derive a variability metamodel: for each mapping defined in the transformation, the types of the input and output elements are extracted and gathered into a model of the possible variabilities of the system.

A HOT included in the Topcased project [33] analyzes a transformation to address the problem of conformance checking. In [33] the authors describe a framework that translates abstract SimplePDL models into Petri nets. The SimplePDL2PetriNet transformation is strongly dependent on a set of implementation choices made by the user. When the author faces the task of checking the correctness of the Petri nets starting from SimplePDL specifications, they need to analyze also the SimplePDL2PetriNet transformations to retrieve the translation choices. The corresponding ATL HOT takes as input the SimplePDL2PetriNet transformation, the Petri net to check, and the high-level specifications. As output it returns a boolean value expressing the result of the conformance test.

6 Transformation Composition

There are two mechanisms to perform the composition of model transformations. External composition consists in chaining separate model transformations and in passing models from one transformation to another. Internal composition composes two model transformation definitions into one new model transformation, with a typically complex merge of the transformation rules. Internal composition, when performed by a model transformation, is a higher-order problem.

[36] provides an example of internal composition performed by a HOT. The paper uses *superimposition* as the composition mechanism to merge two ATL transformation modules into a single output transformation. Superimposition is a simple kind of internal composition in which a transformation module A is superimposed to a transformation module B obtaining a transformation module C, such that: 1) C contains the union of the sets of transformation rules and helpers of A and B; 2) C does not contain any rule or helper of B, for which A contains a rule or helper with the same *name* and the same *context*.

In [36] the HOT is split up in the external composition of two HOTs: ATLCopy.atl that is a simple copying transformation, and Superimpose.atl that provides the special transformation rules for superimposition.

7 Transformation Modification

In our survey the most common use of HOTs is related to the modification of existing transformations. The HOTs of this kind can be more precisely classified in one of the sub-classes described in the following sections.

7.1 Transformation Variants

The transformation variants approach is particularly useful when developing product lines, as in [32]. In this work HOTs are programmed using the model-to-text transformation language MOFScript. Two sets of HOTs are used to generate two kinds of variability: 1) *Platform Variability* is implemented by generic HOTs that are developed once and can be applied on every input transformation; *Intra-domain Variability* is implemented by ad-hoc HOTs that hard-code some domain knowledge to change the internal structure of the original transformation.

A particular kind of variants is produced during mutation analysis. Mutation analysis consists in systematically creating faulty versions of a program (called mutants) and in checking the efficiency of a test dataset to reveal the faults in these erroneous programs. The main interest of mutation analysis is to provide an estimate of the quality of a test dataset with the proportion of faulty programs it detects. To be effective, mutation analysis must create mutant programs that correspond to realistic faults. The faults are injected into the correct program by means of a set of *mutation operators*. The problem to identify a set of realistic mutation operators for model transformations, independently from a particular transformation language has already been studied in [30]. The authors distinguish four error classes, correspondent to the four main operations performed by model transformations, i.e navigation, filtering, output creation, input modification. For each one of the error classes, a set of mutation operators is defined to represent the most common mistakes in transformation development. For instance one of the simplest mutation operators is *Collection Filtering Change with Deletion (CFCD)* that represent the mistake of forgetting a needed filter from the left hand side of a transformation rule. [9] proposes a formalization of mutation operators as a set of HOTs and provides an implementation of the mutation framework in the EMF platform.

7.2 Feature Weaving

HOTs can be easily used to weave cross-cutting concerns into a model transformation. Examples of these concerns are related to debugging, traceability, program tracing. A HOT for adding a cross-cutting concern can usually be programmed with extreme generality, and complete independence from the logic of the original transformation. In this way the same HOT can be used as a general means to add that specific feature to any transformation. As a further consequence several features can be added to the same transformation by sequentially applying the corresponding HOTs.

The use of HOTs in this application case is especially convenient when the transformation language does not provide a native implementation of aspect orientation. An aspect oriented mechanism can be in fact considered as a particular type of higher-order transformation that is performed on-the-fly when the original transformation is executed. A further solution to attach cross-cutting concerns to existing rules could be based on rule inheritance. Such a solution is already programmable in the current ATL but it lacks generality, having a tighter coupling with the program logic. Finally a more general alternative could be implemented by using reflection, letting the developer dynamically plug-in code to existing rules. The coupling between the feature code and the program logic could be relatively loose in this case.

There are several cases in literature that exploit ATL HOTs for cross-cutting concerns. [3] describes an ATL2BindingDebugger HOT that adds a debug instruction to each attribute binding in an input ATL transformation. Each time that a value is assigned to an attribute using a binding in the target element of a rule, a line is printed in the logfile containing the names of the rule, of the target

element and of the assigned value. This feature is easily implemented augmenting each binding in the form $targetAttribute \leftarrow value$ with a print on a logfile using the syntax $targetAttribute \leftarrow value.debug('ruleName.targetName.value')$. The required ATL2BindingDebugger HOT is very concise and shows how nicely HOTs can address cross-cutting concerns.

[21] and [2] provide two different implementations for another application case, in which the addressed cross-cutting concern is traceability, i.e. the maintenance of a set of links between corresponding source and target model elements. In [21] traceability is implemented by adding to each original transformation rule the production of a traceability link in an external ad-hoc traceability model (conforming to a small traceability metamodel). The solution presented in [2] is analogous, with a slightly higher complexity, due to the fact that the traceability link is represented by an ad-hoc extension of the *core weaving metamodel*.

7.3 Changing the Engine Execution Mode

A higher-order transformation can be used to modify the execution mode of the transformation engine for particular model transformations. Some practical problems, in fact, would find a more natural solution if the transformation engine had a different execution semantics. This would allow for more readable and concise transformation code.

For example in [9] transformations are used to implement mutation operators. The most natural way to express a mutation operator is a single transformation rule, whose intended application would require the generation of a different output model (i.e.; a mutation) each time that the rule is applied to a single mutation point. In the chosen solution a HOT translates the simple mutation specification with the mutation semantics in an equivalent, much more verbose, version with the standard execution semantics.

As a side note the solution in [9] has another reason of interest in being the only second order HOT in this survey. It is a transformation that takes as input and output other HOTs, i.e. the mutation operators.

7.4 Transformation Language Extension

One of the means to simplify the specification of a transformation is the addition of new language features, e.g., a new operator. However the direct extension of a transformation language and engine may not be the optimal solution. Often the new feature would be useful in only a limited set of cases, not enough to justify the cost of making the engine and notation heavier.

HOTs provide a good alternative for language extension. The transformation language can be easily extended by adding new metaclasses to the transformation metamodel. Then a HOT can be developed to convert the new elements into a set of rules in the existing transformation language. In this way the extension does not require to change the actual engine and can be easily applied only to transformations that need it. Examples of this kind of transformations are described in [31]. [17] uses this approach to develop an extended version of ATL to address the specification of model matching strategies.

7.5 Parametric Transformation

Being faithful to the "everything is a model" ideal, ATL does not support an explicit parameter initialization mechanism but it requires to define a metamodel for the parameters and to add it as a further input to the transformation.

A common alternative among ATL developers is the use of HOTs to overwrite a parameter value directly defined within the original transformation. This solution is sometimes more concise (thanks to the ATL refining mode) and flexible of the standard one.

7.6 Transformation Adaptation

Within a model-driven system it is possible that a set of properties for a specific transformation is known only at runtime. Often such a problem is addressed by representing these properties as a *configuration model*. The configuration model is given as input to the transformation, guiding a set of choices that are hard-coded into a limited set of rules. This approach, however, requires to define, at development time, the kinds of variations that will be possible at runtime.

HOTs are the natural means to remove any constraint on the possible runtime adaptations of the base transformation. [34] describes an example of this kind, and provides a simple implementation that chooses at runtime among different variants of rule, based on runtime statistics.

8 Other Applications for HOTs

This section lists application areas for HOTs that have not yet been explored by works in literature. The list does not want to be exhaustive but it has the purpose of providing some directions for future works.

Transformation metrics. One of the most natural applications of HOTs is the analysis of model transformations for deriving metric values. Several works implement measurement transformations for generic models (e.g., [35]) and these transformations are of course applicable also to the analysis of transformation models. However, current research lacks a set of specifically higher-order transformations for the generation of transformation metrics.

Transformation refactoring. The building of model transformations could benefit from a set of transformation refactorings, encapsulating best practices for transformation development. These refactorings could be implemented as HOTs and executed by the users during the editing of the transformation. A theory of model transformation refactorings has not been investigated and refactoring HOTs have not been developed yet.

Transformation optimization. The execution of a model transformation could be improved by preceding the execution phase with a preprocessing step. In this step the transformation could be analyzed to detect common patterns and translated into an equivalent version that trades a speed or memory improvement with a loss in readability and manageability. This topic has not been addressed by previous work.

Table 1. Summary of HOTs

Name (# of cases)	Language	Source MM	Target MM	Type	Ref.
AMWtoATL_KM32SQL	ATL	AMW	ATL	Implementation	[7]
AMWtoATL_MantisBug	ATL	AMW	ATL	Implementation	[15]
AMWtoATL	ATL	AMW	ATL	Implementation	[1]
AMWtoXSLT	ATL	AMW	XSLT	Implementation	[1]
ATL2BindingDebugger	ATL	ATL	ATL	Weaving	[3]
ATL2Tracer	ATL	ATL	ATL	Weaving	[21]
ATL2WTracer	ATL	ATL	ATL	Weaving	[2]
ATL2Problem	ATL	ATL	Problem	Analysis	[4]
KM32CONFATL	ATL	KM3	ATL	Generic	[18]
KM32ATLCopier	ATL	KM3	ATL	Generic	[6]
AMWtoATL_Kelly	ATL	AMW	ATL	Implementation	[25]
MMD2ATL	ATL	KM3	ATL	Implementation	[14]
MMTtoMT	ATL	ATL, Ecore	ATL	Execution	[13]
MSDSL2EMF	ATL	KM3	ATL	Generic	[10]
Superimpose	ATL	ATL, ATL	ATL	Composition	[36]
ATLCopy	ATL	ATL, ATL	ATL	Composition	[36]
HITransform	MOFScript	MOFScript	MOFScript	Variants	[32]
Topcased	ATL	ATL	ATL	Analysis	[33]
Metamodel2Derivation	ATL	Ecore	ATL	Generic	[12]
DUALLyLeft2Right (2)	ATL	AMW, Ecore, Ecore	ATL	Implementation	[28]
Easystyle	XSLT	HTML	XSLT	Implementation	[8]
HOT4Tests	ATL	Ecore	ATL	Testing	[9]
Mutators (11)	ATL	ATL, Trace	ATL	Mutation	[9]
SingleApplication	ATL	ATL	ATL	Execution	[9]
patchgen	ATL	AMW	ATL	Implementation	[16]
propagate	ATL	ATL,INMM,INMM,AMW	ATL	Implementation	[16]
VariabilityMM_HOT	GReAT	GME, GReAT	GReAT	Analysis	[26]
MML2MMR (2)	ATL	AMW, Ecore, Ecore	ATL	Implementation	[20]
AML2ATL	ATL	AML	ATL	Extension	[17]
UITransReconfig	ATL	ATL, USRMM	ATL	Adaptation	[34]
Ecore2RDF (2)	ATL	Meo, Ecore, OWL	ATL	Implementation	[19]

Partial Evaluation. While a model transformation has often several input models, it is very common the case in which some of the input models are not subject to frequent changes. At every execution, the transformation needs to process all the input models, including the stable ones. Often it is possible to obtain a remarkable performance gain by removing the stable input models and hard-coding their information inside the transformation. This process, called *partial evaluation* has two benefits: 1) the new version of the transformation needs to process only the input models that actually change between different executions; 2) the hard-coding of some input models can allow ad hoc optimizations and a simplification of the transformation structure. The implementation of a general HOT for the partial evaluation of any transformation, with respect to any input model has never been addressed.

9 Conclusions

In this paper we have presented a categorized survey of HOTs. The categorization is based at its first level on the concept of transformation pattern, for which we have given an informal definition. Four base patterns are identified. Then HOTs are further divided based on some variants of the base patterns. Table 1 summarizes this list of HOTs showing their input and output metamodels and the transformation areas they belong to.

The main contributions of this paper are: 1) to provide an index of the previous work on HOTs, and to identify some important unexplored areas; 2) to provide a list of applications for HOTs proving the practical value of this approach and the necessity of a deeper research in this direction; 3) to identify a limited set of common transformation patterns that involve HOTs; 4) to provide a first categorization of existing and future HOTs based on their role within transformation patterns.

References

1. AMW to ATL, http://www.eclipse.org/gmt/amw/examples/#AMW_2ATL_XSLT
2. AMW Traceability, http://www.eclipse.org/gmt/amw/usecases/traceability
3. ATL to BindingDebugger, http://www.eclipse.org/m2m/atl/atlTransformations/#ATL2BindingDebugger
4. ATL to problem, http://www.eclipse.org/m2m/atl/atlTransformations/#ATL2Problem
5. DoDAF 2004 volume II: product description (4/2/2004), http://www.defenselink.mil/cio-nii/global_Info_grid.html
6. KM3 to ATL copier, http://www.eclipse.org/m2m/atl/atlTransformations/#KM32ATLCopier
7. Translating KM3 into SQL using AMW and ATL, http://www.eclipse.org/gmt/amw/examples/#AMW_KM32SQL
8. WebRatio, http://www.webratio.com/
9. WebRatio MD Framework, http://home.dei.polimi.it/mbrambil/legacytomda
10. Bézivin, J., Hillairet, G., Jouault, F., Kurtev, I., Piers, W.: Bridging the MS/DSL Tools and the Eclipse Modeling Framework. In: Proceedings of the International Workshop on Software Factories at OOPSLA (2005)
11. Bézivin, J., Büttner, F., Gogolla, M., Jouault, F., Kurtev, I., Lindow, A.: Model Transformations? Transformation Models! In: Model Driven Engineering Languages and Systems, pp. 440–453 (2006)
12. Botterweck, G., O'Brien, L., Thiel, S.: Model-driven derivation of product architectures. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, pp. 469–472. ACM, New York (2007)
13. Brambilla, M., Fraternali, P., Tisi, M.: A metamodel transformation framework for the migration of WebML models to MDA. In: MDWE at Models 2008 (2008)
14. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: A metamodel independent approach to difference representation. Journal of Object Technology 6, 165–185 (2007)
15. Del Fabro, M.D., Bezivin, J., Valduriez, P.: Model-Driven Tool Interoperability: An Application in Bug Tracking. LNCS, p. 863. Springer, Heidelberg (2006)
16. Didonet Del Fabro, M., Bézivin, J.: Generic model management: from theory to practice. In: First Intl. Workshop on Towers of Models (2007)
17. Garcés, K., Jouault, F., Cointe, P., Bézivin, J.: A domain specific language for expressing model matching. In: Proceedings of the 5ère Journée sur l'Ingénierie Dirigée par les Modèles (IDM 2009), Nancy, France (2009)
18. Graaf, B., van Deursen, A., Baudry, B., Faivrea, A., Ghosh, S., Pretschner, A.: Using MDE for generic comparison of views. In: Proceedings of the 4th International Workshop on Model Design, Verification and Validation (MoDeVVa 2007) (2007)

19. Hillairet, G., Bertrand, F., Lafaye, J.Y.: MDE for publishing data on the semantic web. In: Transf. and Weaving Ontologies in MDE (TWOMDE) at MODELS 2008 (2008)
20. Jossic, A., Del Fabro, M.D., Lerat, J.P., Bézivin, J., Jouault, F., Sodius, S.A.S.: Model integration with model weaving: a case study in system architecture. In: International Conference on Systems Engineering and Modeling ICSEM 2007 (2007)
21. Jouault, F.: Loosely coupled traceability for ATL. In: Workshop on Traceability at ECMDA 2005, Nuremberg, Germany (2005)
22. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a DSL for the specification of textual concrete syntaxes in model engineering. In: Proceedings of the 5th international conference on Generative programming and component engineering (2006)
23. Jouault, F., Bézivin, J.: KM3: A DSL for Metamodel Specification. In: Formal Methods for Open Object-Based Distributed Systems. LNCS. Springer, Heidelberg (2006)
24. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Satellite Events at the MoDELS 2005 Conference, pp. 128–138 (2006)
25. Cointe, P., Garcés, K., Jouault, F., Bézivin, J.: Adaptation of models to evolving metamodels. Technical report (2008)
26. Kavimandan, A., Klemm, R., Gokhale, A.: Automated Context-Sensitive dialog synthesis for enterprise workflows using templatized model transformations. In: EDOC 2008, pp. 159–168 (2008)
27. Kurtev, I., Bézivin, J., Jouault, F., Valduriez, P.: Model-based DSL frameworks. In: 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, Portland, Oregon, USA, pp. 602–616. ACM, New York (2006)
28. Malavolta, H.M., Pelliccione, P., Tamburri, D.A.: Providing architectural languages and tools interoperability through model transformation technologies. Technical report, TR 004-2008, Available at the DUALLY site (2008)
29. Marcos, D.D.F., Jean, B., Frric, J., Erwan, B., Guillaume, G.: AMW: a generic model weaver. In: 1res Journes sur l'Ingnierie Dirige par les Modles (2005)
30. Mottu, J.-M., Baudry, B., Le Traon, Y.: Mutation Analysis Testing for Model Transformations, pp. 376–390 (2006)
31. Muliawan, O.: Extending a model transformation language using higher order transformations. In: 15th Working Conf. on Reverse Engineering, WCRE (2008)
32. Oldevik, J., Haugen, O.: Higher-Order transformations for product lines. In: Proceedings of the 11th International Software Product Line Conference (SPLC 2007), pp. 243–254. IEEE Computer Society, Washington (2007)
33. Pantel, M.: ACADIE team, OLC team, and TOPCASED team: The TOPCASED project. In: Int. Conf. on Embedded Real Time Software (2006)
34. Sottet, J.S., Ganneau, V., Calvary, G., Coutaz, J., Favre, J.M., Demumieux, R.: Model-Driven adaptation for plastic user interfaces. In: Baranauskas, C., Palanque, P., Abascal, J., Barbosa, S.D.J. (eds.) INTERACT 2007. LNCS, vol. 4662, pp. 397–410. Springer, Heidelberg (2007)
35. Vépa, É., Bézivin, J., Brunelière, H., Jouault, F.: Measuring model repositories. In: Proceedings of the Model Size Metrics Workshop at the MoDELS/UML 2006 conference, Genova, Italy (2006)
36. Wagelaar, D.: Composition techniques for Rule-Based model transformation languages. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) ICMT 2008. LNCS, vol. 5063, pp. 152–167. Springer, Heidelberg (2008)

Managing Model Adaptation by Precise Detection of Metamodel Changes

Kelly Garcés[1,2], Frédéric Jouault[1], Pierre Cointe[2], and Jean Bézivin[1]

[1] AtlanMod, EMN-INRIA Rennes
[2] ASCOLA, LINA (UMR 6241)-INRIA Rennes
{kelly.garces,pierre.cointe}@emn.fr,
{frederic.jouault,jean.bezivin}@inria.fr

Abstract. Technological and business changes influence the evolution of software systems. When this happens, the software artifacts may need to be adapted to the changes. This need is rapidly increasing in systems built using the Model-Driven Engineering (MDE) paradigm. An MDE system basically consists of metamodels, terminal models, and transformations. The evolution of a metamodel may render its related terminal models and transformations invalid. This paper proposes a three-step solution that automatically adapts terminal models to their evolving metamodels. The first step computes the equivalences and (simple and complex) changes between a given metamodel, and a former version of the same metamodel. The second step translates the equivalences and differences into an adaptation transformation. This transformation can then be executed in a third step to adapt to the new version any terminal model conforming to the former version. We validate our ideas by implementing a prototype based on the AtlanMod Model Management Architecture (AMMA) platform. We present the accuracy and performance that the prototype delivers on two concrete examples: a Petri Net metamodel from the research literature, and the Netbeans Java metamodel.

Keywords: Model-Driven Engineering, Model Transformation, Adaptation.

1 Introduction

Software engineers usually have to adapt computer systems to technological and business changes. This need is rapidly increasing in systems built using the Model-Driven Engineering (MDE) paradigm. An MDE system basically consists of metamodels, terminal models, and transformations. The addition of new features and/or the resolution of bugs may change metamodels. The changes may break the consistency of related terminal models and transformations. In this work, we focus on terminal models consistency. Fig. 1 illustrates the problem: a metamodel *MM1* evolves into a metamodel *MM2* (see the dotted arrow). Our concern is to adapt any terminal model *M1* conforming to *MM1* to the new metamodel version *MM2* (see the dashed arrow).

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, pp. 34–49, 2009.

This paper proposes a three-step adaptation. Firstly, a matching process computes the equivalences and changes between *MM1* and *MM2* by executing a set of heuristics. Secondly, an adaptation transformation is derived from the discovered equivalences and changes. Finally, this transformation brings *M1* into agreement with *MM2*, and persists the result in *M2*.

The bulk of this work is devoted to the first step that discovers equivalences, as well as simple and complex changes. We explicitly distinguish two kinds of changes because complex changes need a more insightful adaptation that simple changes. Whereas a simple change describes the addition, deletion, or update of one metamodel concept, a complex change integrates a set of actions affecting multiple concepts[1]. The paper reports a family of heuristics responsible for figuring out both simple and complex changes, and representing them in a straightforward way.

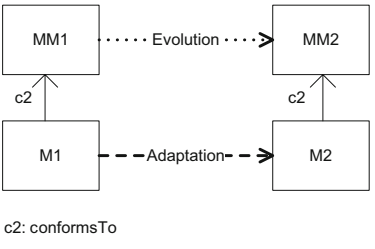


Fig. 1. Metamodel evolution and model adaptation

We have implemented a proof-of-concept prototype based on the AtlanMod Model Management Architecture (AMMA) platform [2]. We have evaluated its performance and accuracy on two examples: a Petri Net metamodel from the research literature, and the Netbeans Java metamodel. The Petri Net metamodel is selected because it is simple enough to be analyzed in a paper, and includes complex changes. The concrete choice of the Netbeans Java metamodel is driven by three main reasons: 1) it is a "real-life" problem, 2) experimental data is widely available (open-source), and 3) the metamodel and terminal models are significantly larger than those of Petri Net, which illustrates the scalability of our approach.

We investigate 6 versions of the Petri Net metamodel (containing between 10 and 20 elements), and 8 versions of the Java metamodel (containing approximately 250 elements). Using this prototype, we are able to analyze on a desktop machine any pair of the Petri Net metamodels in under 1 second, and any pair of the Java metamodels in under 10 seconds. Moreover, our tool always discovers the changes, and only fails by identifying simple changes when in truth there is an equivalence (in 1% of the cases).

This paper is organized as follows: Section 2 compares our contributions to other known solutions. Section 3 presents a running example. Section 4 presents

[1] The reader interested on examples of simple and complex changes may consult [1].

our solution to adapt models to evolving metamodels. Section 5 describes the results of applying our approach on the examples. Finally, Section 6 concludes the paper.

2 Related Works

We may divide the related approaches according to which of the two main issues they deal with: 1) discovery of equivalences and differences, or 2) derivation of adaptation transformations.

We now describe the related works closer to the first problem. In the context of relational and object-oriented data bases, the production of equivalences between two schemas/ontologies has been invested in [3] [4]. In the MDE domain, the approaches of [5] [6] [7] [8] present algorithms for detecting changes between UML models. Sriplakich et al. [9] identify simple changes in terminal models conforming to any metamodel. Wenzel et al. [10] present an approach which discovers fine-grained traces between versions of modeling languages, e.g., UML models, schemas, web service description languages, and domain specific languages. The EMF Compare tool [11] reports simple changes between terminal model pairs or metamodel pairs. Finally, Falleri et al. [12] automatically detect equivalences between two metamodels using the algorithm *Similarity Flooding* described in [13].

In contrast to the first issue, the second one has been addressed by some recent approaches. The works described in [14] [15] [16] [17] [18] assume traces of changes are available, and derive adaptation transformations from them. In particular, [14], [18], and [17] apply stepwise automatic transactions on *MM1* to obtain *MM2*. These approaches then reuse the logs of applied transactions to derive adaptation transformations. Cicchetti et al. [16] use difference models provided by external tools.

The following six items position our approach in comparison with the solutions mentioned above:

1. Similarly to [10] [18], our approach computes equivalences and differences between any pair of metamodels (e.g., representing schemas, UML models, ontologies, grammars) .
2. Our solution overlaps the solutions presented in [14] [16] [17] in the sense of considering both simple and complex changes.
3. As in [12], in our matching process the robust algorithm *Similarity Flooding* can be executed. Falleri's main contribution is to provide 6 graph representation configurations. These configurations generate graphs that contain some metamodel information or all the metamodel information. Although light metamodel representations benefit the algorithm performance, they point out the matching process accuracy increases when all the metamodel information is represented in the graphs. This is what our approach exactly does.

4. Unlike existing approaches [14] [15] [16] [17] [18], we do not suppose that the changes are already known. We consider a more general case where the evolution of metamodels is done without someone explicitly keeping track of the applied changes.
5. The matching step executes modularized heuristics that discover the differences between the metamodels. While most of the previous approaches (with the exception of [4]) execute all the heuristics, each of our heuristics may be plugged or unplugged on demand, which may mean a considerable performance increase.
6. An experimentation shows that our approach scales to larger metamodels and models. This is an improvement on other techniques developed to date.

To sum up, most of the listed works solve the two main issues in an isolated fashion. Some of them are in contexts different to metamodel evolution. In contrast, we propose a solution that addresses all the described model adaptation issues in a consistent and integrated way.

3 Running Example

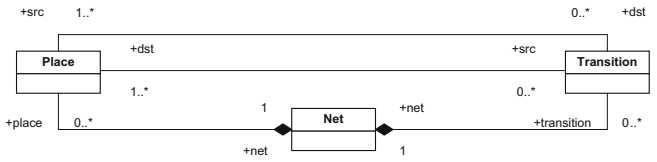
This section describes a running example, i.e., the Petri Net metamodel, and how to represent it using the KM3 metametamodel. We omit describing the Netbeans Java metamodel to save space, this is fully depicted in our technical report [1]. We choose the KM3 notation because it is simple but expressive enough to represent metamodels [19].

3.1 A Sample Petri Net Metamodel

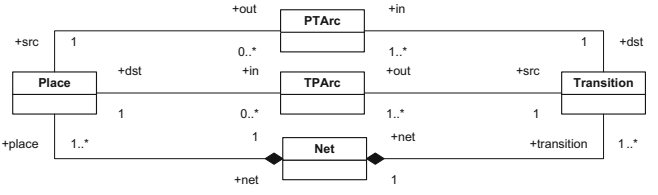
This example is based on the six versions of the Petri Net metamodel provided by [14]. Fig. 2 illustrates versions 0 (*MM1*) and 2 (*MM2*). *MM1* represents simple Petri Nets. These nets may consist of any number of places and transitions. A transition has at least one input and one output place. *MM2* represents more complex Petri Nets. The principal changes between *MM1* and *MM2* illustrated in Fig. 2 are:

- References `place` and `transition` change their multiplicity from 0-* to 1-*.
- Classes `PTArc` and `TPArc` as well as references `in` and `out` are added.
- References `src` and `dst` are extracted from classes `Place` and `Transition`.

Remark 1. The extraction of the reference `dst` illustrates a complex change named *Extract class*. This implies to add and remove a reference, add a class, and associate classes. In considering these actions as isolated simple changes, we may skip changes without migrating involved data from *M1* to *M2*. In contrast, when we distinguish the complex change, we infer (for instance) that the added property (e.g., `dst`), contained in the new class `PTArc`, actually corresponds to the property `dst` removed from the class `Place`. Since we know the relationship between the properties we can migrate the data. We thus need to explicitly distinguish complex changes in order to properly derive adaptation transformations.



(a) Petri Net *MM1* (version 0)



(b) Petri Net *MM2* (version 2)

Fig. 2. Petri Net metamodels

3.2 The KM3 Metametamodel

Fig. 3 shows the basic concepts of the KM3 metametamodel. The `ModelElement` class denotes concepts that have a `name`. `Classifier` extends `ModelElement`. `DataType` and `Class` in turn specialize `Classifier`. `Class` consists of a set of `StructuralFeatures`. There are two kinds of structural features: `Attribute` or `Reference`. `StructuralFeature` has `type` and `multiplicity` (`lower` and `upper` bound). `Reference` has `opposite` which enables to get the owner and target of one reference. `Class` may extend zero or more other classes and may be abstract. In the Petri Net metamodel, `Place` conforms to the `Class` concept. The reference `dst` conforms to the `Reference` concept, this has the attributes `lower` and `upper` with value `0` and `*`.

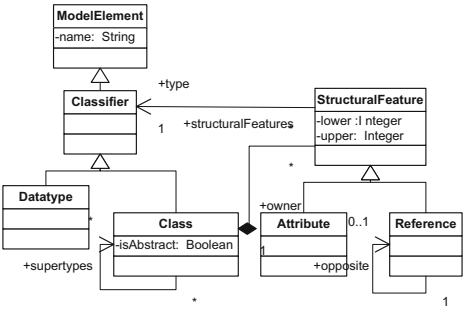


Fig. 3. KM3 concepts

4 A Model Adaptation Approach

As we introduced earlier, our model adaptation approach adapts terminal models in three steps (Fig. 4). In the first step, a *matching strategy* computes

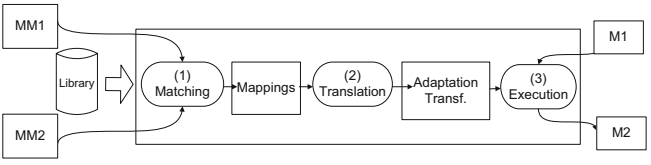


Fig. 4. Approach Overview

equivalences and differences between the metamodels *MM1* and *MM2* by executing a set of heuristics (available in a library). Equivalences and differences are represented by a *matching model*. In the second step, the matching model is translated into an adaptation transformation by using a Higher-Order Transformation (HOT). Finally, the adaptation transformation is executed. Below we discuss the three steps in detail.

4.1 Matching Equivalences and Differences

Matching model. Before describing the matching step, we explain how matching models represent equivalences and differences. A matching model conforms to the *Matching metamodel*[2] illustrated in Listing 1.1. The main concept is `Equal` which describes a mapping (or correspondence) between an element of *MM1* (`leftElement`) and an element of *MM2* (`rightElement`). `Equal` has a similarity value (between 0 and 1) that represents the plausibility of the correspondence. An equivalence with similarity value 1 represents that the *MM2* element is an identical copy of the *MM1* element. An equivalence with similarity value 0.7 describes that the *MM2* element is a copy of the *MM1* element including simple modifications. An equivalence with similarity value 0 link elements different to each other. Other basic concepts are `Added` and `Deleted` which mark a metamodel element as deleted/added from/into *MM1*.

Listing 1.1. Excerpt of the matching metamodel

```
1   class LeftElement extends WLinkEnd {}
2   class RightElement extends WLinkEnd {}
3   class Link extends WLink {
4       reference left[0-1] container : LeftElement;
5       reference right[0-1] container : RightElement;
6   }
7   class Equal extends Link {
8       attribute similarity : Double;
9   }
10  class Added extends Link {}
11  class Deleted extends Link {}    ...
12  class AssociatedClassExtracted extends EqualStructuralFeature {
13      reference associatedReference container : RightElement;
14  }
```

[2] This metamodel extends the core weaving metamodel proposed by [20].

`Equal`, `Added`, and `Deleted` can be extended to describe more specific equivalences or changes. For example, `EqualClass`, `EqualStructuralFeature`, `EqualReference`, `EqualAttribute` indicate the (KM3) types of `leftElement` and `rightElement`. The concept `AssociatedClassExtracted`, in order, links properties undergoing the change *Extract class*.

Matching step. Matching models are computed by matching strategies, i.e., processes that incrementally execute a set of heuristics. The heuristics are needed because the comparison of metamodels (graphs) is a NP complete problem. Fig. 5 is a simple overview of what a matching strategy is. There may be more elaborated matching strategies, e.g., including loops. Fig. 5 presents the kinds of heuristics (using a particular symbol) and their execution order. Every heuristic produces a *matching model*. For the sake of simplicity, we omit intermediate matching models in Fig. 5.

Fig. 5 shows a *Creation* heuristic which prepares a collection of equivalences by matching the elements of *MM1* and *MM2*. Afterward, *Similarity* heuristics compute similarity values by comparing the names, internal properties, and structures of the matched elements. Subsequently, *Filtering* heuristics select equivalences taking into account the confidence value computed by the *Similarity* heuristics. A *Differentiation* heuristic recognizes equivalences, additions, and deletions. The matching step finishes when *Rewriting* heuristics reorganize a given matching model to make it closer to adaptation transformations. Note that the user can build (tune) matching strategies by choosing concrete heuristics (for each kind) from the available library. Moreover, s/he can refine the matching models generated along the process.

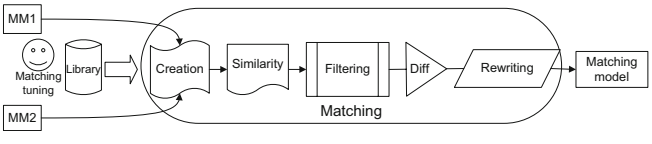


Fig. 5. A simple matching strategy

Let us now describe the kinds of heuristics by means of concrete implementations. We select them because they analyze "safe" indicators (e.g., name) that two elements are the same [6]. In this paper, we describe the heuristics in terms of the KM3 concepts. Our approach remains nonetheless generic (i.e., independent of KM3). The heuristics can be implemented using other formalisms such as MOF or EMF. We now illustrate the family of heuristics using the AtlanMod Transformation Language (ATL) [21]. An ATL transformation takes models (conforming to input metamodels) as input, and yields models (conforming to output metamodels) as output. The transformations are composed of rules. A rule consists of two mandatory parts: the *from*, and the *to* [22]. The *from* part defines an input pattern and an Object Constraint Language (OCL) condition [23] (written in terms of input metamodel concepts). The *to* part specifies an output pattern

and bindings. Each category below has a code listing whose number is enclosed between parenthesis.

Creation (Listing 1.2) The creation heuristics match concepts of *MM1* (line 3) and concepts of *MM2* (line 4), and create equivalences (lines 7-10) when the concepts hold a condition (line 5). The properties `left` and `right` of `Equal` refer to *MM1* and *MM2* (lines 8-9). The condition is written in terms of the KM3 metamodel concepts.

Listing 1.2. Creation transformation excerpt

```
1  rule Creation {
2      from
3          a : KM3!<Concept> in MM1
4          b : KM3!<Concept> in MM2
5          (condition) -- for example, a.type = b.type
6      to
7          e : EqualMM!Equal (
8              left <- a.ref,
9              right <- b.ref
10         )
11 }
```

A simple *Creation* heuristic is *Creation by type*. This creates a mapping when two elements conform to the same KM3 type (i.e., `Class`, `Reference` or `Attribute`) (see comment in line 5). Another *Creation* heuristic is *Creation by Type and FullName* which creates mappings when two elements has the same KM3 type and fullname. The fullname is a string that concatenates the names of elements related to another element. For example, the fullName of `transition` reference is `PetriNet|Net|transition` because the `PetriNet` package contains the `Net` class, and this class contains the `transition` reference.

Similarity (Listing 1.3). The similarity heuristics compute a similarity value for each equivalence prepared by *Creation* heuristics (line 3). A function (*func*) establishes the similarity values (line 6). Note that the *Similarity* heuristics have no longer the KM3 concepts in the input pattern. Instead of that, the Equal concept is used. The function refers to *MM1* and *MM2* concepts by using `left` and `right` of `Equal`.

Listing 1.3. Similarity transformation excerpt

```
1  rule Similarity {
2      from
3          e : EqualMM!Equal
4      to
5          e : EqualMM!Equal (
6              sim <- func
7          )
8  }
```

Next we present four *Similarity* heuristics whose functions calculate similarity values by comparing particular properties of the KM3 metamodels.

- *Name Similarity* compares the names of KM3 elements in different ways, for instance, using string comparison algorithms [24] or dictionaries of synonyms [25].
- *Multiplicity Similarity* compares the multiplicity of references and attributes. This assigns a similarity value to mappings connecting metamodel references that have the same multiplicity (lower and upper bounds).
- *Similarity by Internal Properties* compares several properties of the KM3 elements. Each property contributes a relative similarity value, i.e., the similarity value multiplied by a weight. The net similarity value is the sum of all relative values. For instance, the net similarity value of elements conforming to `Reference` is the sum of the relative values of `names`, `types`, `multiplicities`, and `opposites`.
- *Context Similarity* compares the relationships between metamodel elements. For example, this compares attributes/references contained in a given class, its superclasses, and its associated classes. The implementation of this *Similarity* heuristic is more complex than the previously presented ones. This is inspired from the *Similarity Flooding* (SF) algorithm. Our algorithm is executed in two steps. The first step associates two equivalences (*e1* and *e2*) if there is a relationship between the linked elements. The second step propagates the similarity value from *e1* to *e2* because of the relationship.

Filtering (Listing 1.4). The previous heuristics may have created unwanted equivalences (i.e., equivalences with low similarities). The filtering heuristics select the equivalences (line 3) whose similarity values satisfy a condition (line 4). A basic filtering heuristic is *Threshold*. This selects the mappings with a similarity value higher than a given threshold value (see comment of line 4).

Listing 1.4. Filtering transformation excerpt

```
1 rule Filtering {
2     from
3         e : EqualMM!Equal
4             (condition) --For example, e.similarity > threshold
5     to
6         e : EqualMM!Equal
7 }
```

Differentiation (Listing 1.5). This kind of heuristic distinguishes between equivalent, deleted, and added metamodel elements. A concrete implementation of *Differentiation* heuristics compares KM3 elements to equivalences. The intuition is that not linked metamodel elements correspond to deletions and additions. Listing. 1.5 marks *MM1* elements as deleted elements (line 6) when they are not linked by equivalences (lines 3-4). The implementation contains another rule, ommitted because of space constraints, that marks unlinked *MM2* elements as added elements.

Listing 1.5. Differentiation transformation excerpt

```
1 rule Deleted {
2     from
3         a : KM3!ModelElement in MM1
4             a.notLinked()
5         )
6     to
7         e : EqualMM!Deleted
8 }
```

Rewriting (Listing 1.6). Before this step, most equivalences and differences are contained in a single large collection. This heuristic reorganizes/retypes the equivalences and differences in order to make them semantically richer. We discern three *Rewriting* heuristics: *Nesting*, *Flattening*, and *Complex changes*. The *Nesting* and *Flattening* heuristics reorganize the equivalences considering the relationships (containment and inheritance) between the linked concepts. For instance, the *Nesting* heuristic rewrites `(transition, transition)` as a child of `(Net, Net)` because of the containment relationship between these elements. The *Complex change* heuristic infers complex changes from equivalences, additions, and deletions. For example, Listing 1.6 shows a rule that verifies if change *Extract Class* has happened. The rule assemblies two properties a (added) and d (deleted) using the `AssociatedClassExtracted` type. The conditions are: 1) an introduced class owns the property a (line 5), and 2) this class is associated to other class that contains d (line 7).

Listing 1.6. Complex changes transformation excerpt

```
1 rule AssociatedClassExtracted {
2     from
3         d : EqualMM!DeletedStructuralFeature,
4         a : EqualMM!AddedStructuralFeature (
5                         a.right.target.owner.isNewClass()
6                         and
7                         a.right.target.owner.isAssociatedTo(d.left.
                              ↪target.owner)
8             )
9     to
10        e : EqualMM!AssociatedClassExtracted
11 }
```

4.2 Translation to Adaptation Transformations

In this step, the equivalences and differences are translated into an executable adaptation transformation via a HOT. The HOT takes as input the final matching model, and generates as output a model transformation written in a particular transformation language (e.g., ATL, XSLT, SQL-like). The HOT follows the guidelines below:

– Yield a transformation rule for each `EqualClass` that links no abstract classes. The HOT takes referred left and right classes to yield input and output patterns.

– Create a binding for each `EqualStructuralFeatures` attached to a `EqualClass`. The binding complexity depends on the `Equal` type. While a simple `EqualStructuralFeature` generates a simple binding, `EqualStructuralFeature` extensions (e.g., `AssociatedClassExtracted`) generate more elaborated bindings. In general, sophisticated bindings instruments the code that adapt *M1* models to complex changes.

Listing. 1.7 shows an adaptation transformation, written in ATL, which is generated by a concrete HOT. This creates the transformation rule `Place2Place` (line 1) from the equivalence (`Place`, `Place`). The *from* part matches the elements conforming to `Place` (line 3). The *to* part creates elements conforming to `Place`. The HOT moreover generates a complex binding (see line 6) from the equivalence (`out`, `dst`). The binding calls an additional rule (i.e., `dstPTArc`) to initialize `dst` of `PTArc` (lines 18) using the values `dst` of `Place`.

Listing 1.7. Transformation excerpt (Petri Net example)

```
1  rule Place2Place {
2      from
3          pV1 : MM1!Place
4      to
5          pV2 : MM2!Place (
6              out <- pV1.dst -> collect (tV1 | thisModule.dstPTArc(tV1
                   ↪, pV1)}))
7          )
8  }
9  unique lazy rule dstPTArc {
10     from
11         transition : MM1!Transition,
12         place : MM1!Place
13     to
14         tV2 : MM2!PTArc (
15         dst <- transition
16     )
17 }
```

4.3 Adaptation Transformation Execution

This step simply executes the generated adaptation transformation. The transformation takes any terminal model *M1* and generates a terminal model *M2*.

5 Experimental Validation

Section 5.1 describes the prototype platform. Section 5.2 presents the experimental settings including dataset and procedure. Section 5.3 provides the metrics to evaluate the results. Section 5.4 discusses the experimentation results. Finally, Section 5.5 shows the results of applying the EMF Compare tool to the running examples, and compares them to our results.

5.1 Prototype Implementation

We implement the prototype on the AMMA platform [2]. More specifically, we use the AtlanMod Model Weaver (AMW) [26] to work with matching models, and we specify the heuristics and HOT in ATL. The HOT generates the adaptation transformation in ATL code. In particular, we develop a library that contains the heuristics described in Section 4.1.

5.2 Experimental Settings

Data set. We have results from experimentations which use 8 versions of the Netbeans Java metamodel, and 6 versions of a Petri Net metamodel provided by [14]. For the sake of readability, we just present the results in applying our approach on three versions of each metamodel. These versions are chosen because they contain significant changes. In the Java example, we choose the versions 1.12, 1.13, and 1.15. In the Petri Net example, we use the versions 0, 1, and 2. Table 1 shows the number of elements (classes, attributes and references) contained in the versions. We match the following couples of versions: 0 – 1, 0 – 2, 1.12 – 1.13, and 1.12 – 1.15.

Table 1. Metamodel elements

Example	PetriNet			Java		
Version	0	1	2	1.12	1.13	1.15
Elements	11	11	21	255	256	258

Procedure. We tested different matching configurations until obtaining the strategies more suitable for the examples. Garces et al. [1] presents lessons learned from this selection process. We have picked up the matching strategies (A and B) for matching the Petri Net metamodels and the Java metamodels, respectively. The heuristics that include each strategy are:

- **Matching Strategy A:** Creation by type, Similarity by internal properties, Context similarity, Threshold, Differentiation, Nesting, and Complex changes.
- **Matching Strategy B:** Creation by type and fullname, Similarity by internal properties, Context similarity, Threshold, Differentiation, Nesting, and Flattening.

This selection should not question the applicability of our approach, but show that the matching accuracy and performance highly depends on the metamodels.

5.3 Metrics

We have measured the matching step accuracy by applying three metrics [27]: $Precision(x) = \frac{CorrectFound(x)}{TotalFound(x)}$, $Recall(x) = \frac{CorrectFound(x)}{TotalCorrect(x)}$, and $Fscore(x) = \frac{2*Recall(x)*Precision(x)}{Recall(x)+Precision(x)}$.

The x denotes equivalences, additions/deletions, or complex changes. Besides additions and deletions, we have not evaluated other simple changes because these require no elaborated adaptation transformations. The expected values of these metrics are between 0 and 1. The higher is the precision value, the smaller is the set of wrong mappings. The higher is the recall value, the smaller is the set of the mappings that have not been found. Fscore is a global measure of the matching quality. A high fscore value indicates a matching of high quality.

We have identified the correct equivalences and changes in two ways. In the Petri Net example, we manually discovered the changes. In the Java example, we relied on the changes logged in the Netbeans repository. We also considered other manually discovered changes. We remarked that some repository logs do not report all the performed changes.

Besides matching accuracy, we have measured the matching process performance. This has been executed on a machine with Intel Core 2 Duo (2.4 GHz) and 1GB RAM.

5.4 Results

Matching accuracy. Fig. 6 gives the prototype's accuracy. The histograms display measures (precision, recall, fcore) for each selected couple of version. The three bars (from left to right) show the accuracy of equivalences, additions/deletions, and complex changes. Some bars are missing because certain couples of versions contain no deletions/additions or complex changes.

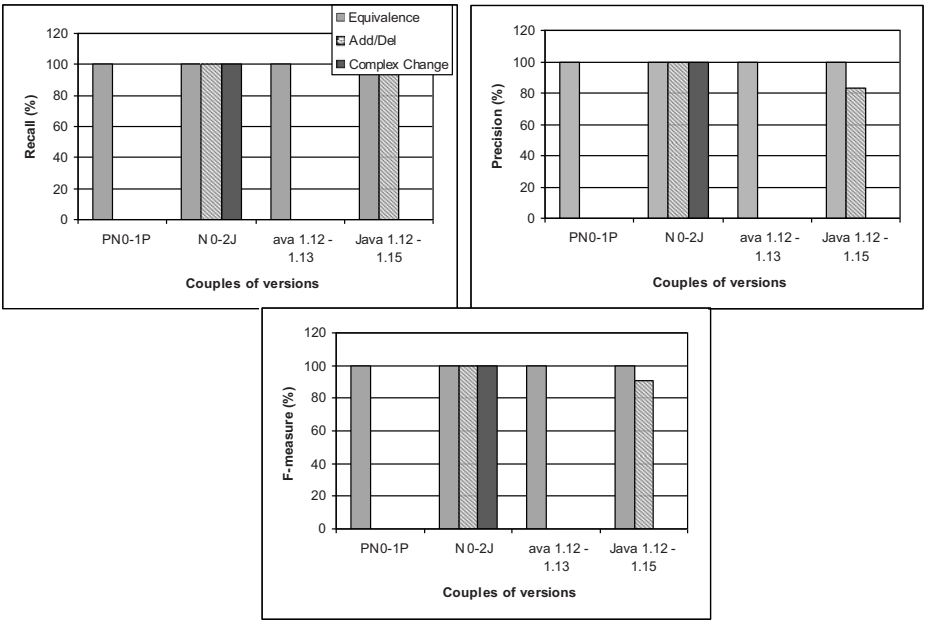


Fig. 6. Matching accuracy results

The results show that the prototype achieves a high accuracy not only in detecting the correct equivalences, additions/deletions, but also in detecting complex changes. Taking fscore as an example, the percentage of correct equivalences, and additions and deletions ranges from 99%-100%, and 90%-100%, respectively. Averanging accross all experiments, the fscore of complex changes is 100%. In particular, our prototype fails in identifying additions/deletions instead of equivalences (1% of cases).

Even though we have applied our approach to only a small number of test cases (mostly because of model availability restrictions), this number is significant in comparison with test cases of other closely related approaches like [14] [15] [16] [17] [18]. We are looking for other benchmarks in open-source projects, and we hope to provide more validation material in the future.

Performance. In the Petri Net example, the matching process consumes less than 1 second. In the Java example, the matching process approximately takes 10 seconds. A table containing the execution times of the heuristics in detail can be find in [1]. Even if the matching step consumes a relevant amount of resources, we should remember that this process generates an adaptation transformation that can be used several times.

5.5 EMF Compare versus Our Approach

We have compared the metamodel changes computed by EMF Compare to our results. We chose EMF Compare because this is a prototype completely available to compare metamodels. Table 2 shows the fscore that EMF Compare and our approach, denoted by i. and ii., deliver on the Petri Net (couple 0-2) and Java (couple 1.12 - 1.15) examples. While EMF Compare is fairly good for identifying additions and deletions, this fails in rendering them as isolated actions. Because model adaptation automation needs to distinguish complex changes (i.e., not only simple changes), our approach is more appropriate for this purpose than EMF Compare.

Table 2. Fscore EMF Compare (i.) - Our approach (ii.)

Example	PetriNet		Java	
Couples of versions	0-2		1.12-1.15	
Approach	i.	ii.	i.	ii.
Additions-Deletions	0.8	1	1	0.9
Complex changes	0	1	0	0

6 Conclusions

In this paper, we presented an MDE approach for adapting models to their evolving metamodel. Matching strategies compute equivalences and changes between two metamodels by executing a set of heuristics. These equivalences and differences are saved in a matching model. A Higher-Order Transformation translates

this matching model into an executable adaptation transformation. We reported the performance and precision of our approach which are pretty good, and may be even further improved by means of tuning. We also compared our solution to related works, and we showed that no other work known to us covers fully the problem as we identified it (e.g., most other works only cover evolution with available trace of changes). Moreover, our validation covers a wider spectrum than existing works: a relatively simple case from the literature (i.e., a Petri Net metamodel), but also a real-life scenario (i.e., a Java metamodel). We have used the family of heuristics to design the constructs of the AtlanMod Matching Language (AML), a Domain-Specific Language (DSL) for expressing matching strategies [28]. AML allows to express not only strategies to match two metamodels, but also any pair of models. By using this language, we hope to implement matching strategies more easily, and extract further guidelines on proper parametrization of them.

Acknowledgements

This work has been partially funded by the ANR project FLFS.

References

1. Garcés, K., Jouault, F., Cointe, P., Bézivin, J.: Adaptation of models to evolving metamodels. Technical report, INRIA (2008)
2. Kurtev, I., Bézivin, J., Jouault, F., Valduriez, P.: Model-based DSL frameworks. In: OOPSLA 2006, Portland, OR, USA, October 22-26, pp. 602–616. ACM, New York (2006)
3. Rahm, E., Bernstein, P.: A survey of approaches to automatic schema matching. The VLDB Journal 10(4), 334–350 (2001)
4. Do, H.H.: Schema Matching and Mapping-based Data Integration. Ph.D thesis, University of Leipzig (2005)
5. Ohst, D., Welle, M., Kelter, U.: Differences between versions of UML diagrams. SIGSOFT Softw. Eng. Notes 28(5), 227–236 (2003)
6. Xing, Z., Stroulia, E.: UMLDiff: an algorithm for object-oriented design differencing. In: ASE 2005, pp. 54–65. ACM, New York (2005)
7. Girschick, M.: Difference detection and visualization in UML class diagrams. Technical report, TU Darmstadt (2006)
8. Treude, C., Berlik, S., Wenzel, S., Kelter, U.: Difference computation of large models. In: Crnkovic, I., Bertolino, A. (eds.) ESEC/SIGSOFT FSE, pp. 295–304. ACM, New York (2007)
9. Sriplakich, P., Blanc, X., Gervais, M.P.: Supporting collaborative development in an open MDA environment. In: ICSM, pp. 244–253. IEEE Computer Society, Los Alamitos (2006)
10. Wenzel, S., Kelter, U.: Analyzing model evolution. In: Robby (ed.) ICSE, pp. 831–834. ACM, New York (2008)
11. Eclipse.org: EMF Compare (2008), http://wiki.eclipse.org/index.php/EMF_Compare

12. Falleri, J.R., Huchard, M., Lafourcade, M., Nebut, C.: Metamodel matching for automatic model transformation generation. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 326–340. Springer, Heidelberg (2008)
13. Melnik, S., Garcia-Molina, H., Rahm, E.: Similarity flooding: A versatile graph matching algorithm and ist application to schema matching. In: Proc. 18th ICDE, San Jose, CA (2002)
14. Wachsmuth, G.: Metamodel adaptation and model co-adaptation. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 600–624. Springer, Heidelberg (2007)
15. Gruschko, B., Kolovos, D., Paige., R.: Towards synchronizing models with evolving metamodels. In: Workshop on Model-Driven Software Evolution, MODSE 2007, Amsterdam, The Netherlands (2007)
16. Cicchetti, A., Ruscio, D.D., Eramo, R., Pierantonio, A.: Automating co-evolution in model-driven engineering. In: EDOC 2008: Proceedings of the 12th IEEE International EDOC Conference, München, Germany (2008)
17. Herrmannsdoerfer, M., Benz, S., Juergens, E.: Automatability of coupled evolution of metamodels and models in practice. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 645–659. Springer, Heidelberg (2008)
18. Vermolen, S.D., Visser, E.: Heterogeneous coupled evolution of software languages. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 630–644. Springer, Heidelberg (2008)
19. Jouault, F., Bézivin, J.: KM3: a DSL for Metamodel Specification. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 171–185. Springer, Heidelberg (2006)
20. Didonet del Fabro, M.: Metadata management using model weaving and model transformation. Ph.D thesis, Université de Nantes (2007)
21. Jouault, F., Kurtev, I.: Transforming models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
22. Eclipse.org: The ATL User Manual (2008), http://www.eclipse.org/m2m/atl/doc/ATL_User_Manualv0.7.pdf
23. OMG: OCL 2.0 Specification, OMG Document formal/2006-05-01 (2006), http://www.omg.org/docs/ptc/05-06-06.pdf
24. Cohen, W., Ravikumar, P., Fienberg, S.: A comparison of string distance metrics for name-matching tasks. In: Kambhampati, S., Knoblock, C.A. (eds.) Proceedings of Workshop on Information Integration on the Web, IIWeb 2003, Acapulco, Mexico, pp. 73–78 (2003)
25. University of Princeton: Wordnet: An Electronic Lexical Database, http://wordnet.princeton.edu/
26. Didonet Del Fabro, M., Bézivin, J., Jouault, F., Breton, E., Gueltas, G.: AMW: A generic model weaver. In: Proceedings of the 1ère Journée sur l'Ingénierie Dirigée par les Modèles, IDM 2005 (2005)
27. Rijsbergen, C.J.V.: Information Retrieval. Butterworths (1979)
28. Garcés, K., Jouault, F., Cointe, P., Bézivin, J.: A Domain Specific Language for Expressing Model Matching. In: Proceedings of the 5ère Journée sur l'Ingénierie Dirigée par les Modèles, IDM 2009 (2009)

A Pattern Mining Approach Using QVT

Jens Kübler[1] and Thomas Goldschmidt[2]

[1] aquintos GmbH
Karlsruhe, Germany
kuebler@aquintos.com
[2] FZI Research Center for Information Technology
Karlsruhe, Germany
goldschmidt@fzi.de

Abstract. Model Driven Software Development (MDSD) has matured over the last few years and is now becoming an established technology. Models are used in various contexts, where the possibility to perform different kinds of analyses based on the modelled applications is one of these potentials. In different use cases during these analyses it is necessary to detect patterns within large models. A general analysis technique that deals with lots of data is pattern mining. Different algorithms for different purposes have been developed over time. However, current approaches were not designed to operate on models. With employing QVT for matching and transforming patterns we present an approach that deals with this problem. Furthermore, we present an idea to use our pattern mining approach to estimate the maintainability of modelled artifacts.

1 Introduction

Model Driven Software Development (MDSD) matured over the last years and is becomming more and more adopted within industry. One of the strengths of MDSD is that it yields the possibility to not only use models as basis for the implementation of a system but rather allow to do different kinds of analysis on an abstract level. Analysis, such as maintainability and complexity measures are based on analysing models. Measures such as frequently occurring constructs are based on finding patterns within these models. If model driven techniques are widely employed within an enterprise models get larger and larger. Finding patterns witin these potentially huge models then becomes a time consuming task. Optimized pattern mining algorithms would be required to cope with this problem.

Pattern mining on databases is used in different contexts to identify frequent patterns within structured data [1]. Algorithms for this kind of knowledge discovery have been highly optimized to be able to deal with huge amounts of data. Furthermore, approaches have been developed to do pattern mining on object structures [2]. However, none of these approaches can deal with generic models based on different metamodels. On the other hand, model transformation engines like M2ToS[3] or specifications like Query/View/Transformations (QVT)[4] provide some means of executing queries on the model as an integral part to do rule matching. They allow for specification of object patterns and retrieval of matching results.

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, pp. 50–65, 2009.

Therefore we propose to use a transformation based approach to do analysis through pattern mining on models with the known pattern mining algorithm “Apriori” and draft a metric that may give a quality indicator that is automatically collectible and comparable between models. Apriori aims at identifying sets of items that frequently occur together focusing on those that are not contained within any other itemset. These itemsets are also known as maximum frequent itemsets which must be more frequent than a parameter given to the algorithm known as support. Identification of maximum frequent itemsets is archived through an iterative generate-and-test approach. To achieve this for models we use pattern matching facilities defined by QVT specification to formulate and execute queries on models. We apply an object oriented version of the Apriori algorithm to detect frequent patterns within models and leverage the GenMax heuristic to speed up their detection.

Through an experimental study on Ecore models we show that patterns from simple structured classes can be mined within acceptable timeframes whereas mining complex structured classes requires more optimization to the algorithm in the future to be efficiently mined. We enlist several hot spots where these optimizations may be introduced.

Furthermore, we present initial work towards an application of our model based pattern mining approach to estimate the maintainability of models. A large part of the complexity of a software product and therefore also it’s models is comes through the process of needing to understand patterns that occur within these models. Thus, we propose to emulate human information processing through pattern mining on models.

The structure of this paper is as follows. Section 2 discusses related work and analysis its shortcomings. The general pattern mining approach is described in Section 3. Section 4 raises the pattern mining approach to the model level. First experimental results of the pattern mining on models are presented in Section 5. Section 6 presents an initial approach to estimate the maintainability of models using pattern mining. Finally, Section 7 concludes and points out future work.

2 Related Work

The Apriori-Algorithm used to mine patterns was extended beyond relational data structures to be able to cope with object structures. While [5] presents an approach to find frequent tree structures within XML data, [6] also includes mining on graph structures. However, both approaches do not account for the structures that appear in object oriented structures, such as inheritance. A more interesting appraoch is presented in [2], it finds frequent item sets within object structures including inheritance, classes attributes and their values. We use this approach as a basis for our metamodel independent, generic pattern mining approach.

In [7] Zhao et al. present a graph-transformation framework for pattern level design validation and evolution. By using key structures it is tried to locate the searched design patterns. Furthermore, it is possible to evolve the defined patterns using graph-transformation. However, in the the focus of [7] is not in identifying possible patterns within a model but rather finding predefined ones.

Mazón et al introduce in [8] QVT transformation rules that allow to automatically obtain a logical representation tailored to a multidimensional database technology.

However, those transformations aim at generating models for multidimensional databases from a UML profile that was created as a platform independent model for the modelling of multidimensional databases. Their approach is not concerned with identifying patterns within models themselves.

3 General Pattern Mining Approach

We understand patterns within the context of model driven software development as frequently occurring object graphs. Therefore pattern mining is the automated process of finding frequently occurring object graphs within model instances.

3.1 Patterns and Pattern Mining

Pattern mining as a technique to knowledge discovery has received much attention in recent decades and several algorithms like apriori and FP-Tree have been proposed to cope with the computationally complex problem. We will focus on the apriori approach for which numerous extensions have been introduced. The following paragraphs informally introduce apriori and discuss how patterns that are rather huge can be addressed. The aforementioned object oriented extension to apriori will be discussed in detail as we will modify it to handle huge patterns.

3.2 Apriori Algorithm

Informally the classic apriori algorithm is often motivated through items of a warehouse that are bought together at the same time. Time points are identified through database transaction ids whereas bought items resemble a set belonging to a transaction id. Apriori aims at identifying sets of items that frequently occur together focusing on those that are not contained within any other itemset. These itemsets are also known as maximum frequent itemsets which must be more frequent than a parameter given to the algorithm known as support. Identification of maximum frequent itemsets is archived through an iterative generate-and-test approach. Candidate itemsets will be generated and tested for frequency above minimum support at each iteration step and discarded, if their frequency is below minimum support. Figure 1 shows an example how four items A, B, C, D may be iteratively combined to candidate itemsets and tested for frequency. Assuming that the whole itemset of $ABCD$ is frequent, apriori generates and tests 14 candidates for frequency which may be acceptable for four items in terms of runtime performance. However research has identified that $2^k - 2$ itemsets may have to be generated and tested, given k as the count of possible items. This means that apriori scales exponentially with the number of items especially if frequent itemsets contain numerous items. These itemsets are also known as long patterns.

Several works such as [9] and [10] introduced heuristics to cope with this potentially time consuming task. Instead of iteratively generate and test all candidate sets the heuristics proposed explicitly calculate all maximum frequent itemsets. Their key contribution is to enumerate all candidate itemsets as shown in figure 2. The enumeration spans a tree which resembles the search space of the algorithm. This representation of search space allows for the application of known tree traversals such as depth-first or

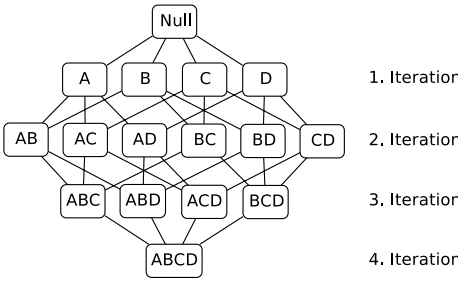


Fig. 1. Classical apriori candidate set generation: Within each iteration frequency is determined for all combinations of candidate sets

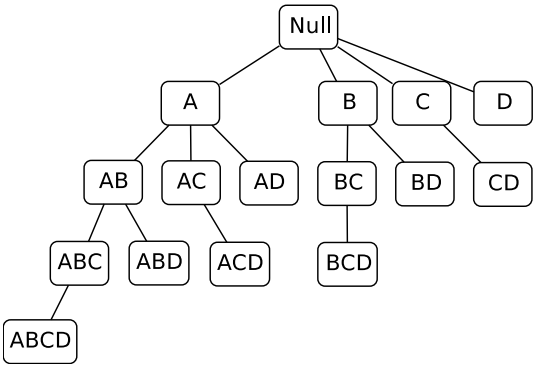


Fig. 2. The complete enumeration of all possible solutions for frequent itemsets containing four elements A, B, C, D. The enumeration spans a tree which can be searched by known tree traversal algorithms.

breadth-first. Leveraging the fact that any frequent itemset is contained at least within one maximal frequent itemset evaluations show that many candidates can effectively be pruned during traversal thus reducing search space. As mining on graphs is computationally more complex than mining on itemsets [11], heuristics are even more required to efficiently mine patterns.

Mining on object graphs has been proposed in [2]. The authors incorporate object-oriented concepts like inheritance, classes and attribute values into their algorithm. The algorithm is outlined in Algorithm 1 and follows the classic generate-and-test paradigm of apriori. Given database D, a query class C_q and minimum support of $minsupp$ the algorithm finds all patterns having a support greater than $minsupp$. Generation of candidate patterns is being split into a two step process of a linear generation step and a nonlinear combination step. Testing is then applied to each of the linear and nonlinear candidate patterns until no further candidate pattern is frequent.

Object oriented extensions reside within the linear extension phase of the algorithm as can be seen in Algorithm 2. Based upon the type of the patterns last extension the subsequent extension will be determined. If the type is a primitive type, i.e. an integer,

Input: database D, query class C_q, minimum support $minsupp$
Output: frequent patterns having support greater than $minsupp$

$LC_1 = \{X_0 : C_q\}$
count support s for pattern $X_0 : C_q$ of LC_1
if $s < minsupp$ **then**
  return $\emptyset$
else
  $L_1 = LC_1$
  $N_1 = \emptyset$
  for $k = 1; L_k \cup N_k \neq \emptyset; k = k + 1$ **do**
    LC_{k+1} = linear extension of patterns L_k
    NC_{k+1} = combination of patterns from L_k and N_k
    count support s of patterns in LC_{k+1} and NC_{k+1}
    L_{k+1} = patterns LC_{k+1} having $s > minsupp$
    N_{k+1} = patterns NC_{k+1} having $s > minsupp$
  end
  return $\bigcup_k (L_k \cup N_k)$
end

Algorithm 1. Object frequent pattern mining taken from [2]: X denotes variables and T their type

for each instance value a new patterns will be generated extending the current pattern with the current instance value. If the type is a class two extensions will be considered. First for each attribute a new pattern is generated by extending the current pattern with the attribute. Second for each subclass a new pattern is generated by extending the current pattern with the current subclass. If the attribute type is a collection the pattern will be extended by the contained type of the collection.

4 Pattern Mining On Models

To mine patterns from models we propose to combine the previously presented object mining approach with the heuristic GenMax algorithm presented in [10]. This heuristic has been evaluated in the same work to outperform other known heuristics like Mafia or MaxMiner, if pattern length distributes between 10 and 25 items given a low support.

Because the GenMax approach is based on the complete set enumeration of all items and oo-apriori alternately executes a linear extension and non-linear combination the search-space is extended continuously. To prevent this extension and be able to apply the GenMax heuristic the linear extension and the non-linear combination is separated as it is shown in Algorithm 3.

To execute the pattern mining on models we used an implementation of the QVT Relational standard called medini QVT [12]. The template expressions that are used to do the matching in QVT-R represent an object pattern which can be used for pattern mining.

Input: linear frequent patterns L_k
Output: linear candidate patterns LC_{k+1}

forall $pattern \in L_k$ **do**
  $LC_{k+1} = \emptyset$
  $X_i : T_i$ = last extension of pattern p
  if T_i *is primitive type* **then**
    foreach *instance value* I *of* T_i **do**
      extend pattern p with $X_i : T_i = I$
      add newly extended pattern to candidate set LC_{k+1}
    end
  else if T_i *is a class* **then**
    foreach *subclass* T_j *of* T_i **do**
      extend pattern p with subclass $X_i : T_j$
      add newly extended pattern to candidate set LC_{k+1}
    end
    foreach *attribute* a *of class* T_i **do**
      extend pattern p with attribute $a : T_j : X_i : T_i.a = X_j : T_j$
      add newly extended pattern to candidate set LC_{k+1}
    end
  else if T_i *is collection* **then**
    extend pattern p with type that is contained within collection T_j: $X_i : T_i = \{X_j : T_j\}$
    add newly extended pattern to candidate set LC_{k+1}
  end
end
return LC_{k+1}

Algorithm 2. Linear pattern extension taken from [2] : X denotes variables whereas T denotes its type

Listing 1.1. Example pattern

```
enforce domain source pattern : ecore::EClass {
    interface = false , eAnnotations = undefined ,
    abstract = true, eIDAttribute = undefined ,
    ePackage = _package : ecore::EPackage { nsPrefix = 'wsdl' ,
        name = 'wsdl' ,
        eFactoryInstance = factory : ecore::EFactory {
            ePackage = _f_package : ecore::EPackage { }
        } ,
        eSuperPackage = undefined,
        eAnnotations = _p_annotations : ecore::EAnnotation { }
    } ,
    nsURI = 'http://www.eclipse.org/wsdl/2003/WSDL' ,
    eClassifiers = _classifier : ecore::EClassifier{ } ,
    name = _name : String { }
};
```

Input: database D, query class C_q, minimum support $minsupp$
Output: frequent patterns having support greater than $minsupp$

$LC_1 = \{X_0 : C_q\}$
count support s for pattern $X_0 : C_q$ in LC_1
if $s < minsupp$ **then**
  return $\emptyset$
else
  $L_1 = LC_1$
  $N_1 = \emptyset$
  for $k = 1; L_k \neq \emptyset; k = k + 1$ **do**
    LC_{k+1} = linear extension of patterns from L_k
    count support s for patterns in LC_{k+1}
    L_{k+1} = patterns of LC_{k+1} having $s > minsupp$
  end
  FP = GenMax($\bigcup_k L_k, minsupp$)
  return FP
end

Algorithm 3. Object frequent pattern mining extended by GenMax heuristic

QVT transformations themselves are also available as a model representation and can be executed based on this representation. All patterns can therefore be represented as relational rules that match exactly its pattern if executed as transformation. Using higher order transformations it becomes also possible to create these model patterns using model transformations. Therefore the representation of patterns using QVT-R relations serves two purposes. One being the representation of the patterns and the other being directly executable to recognize this pattern in models. An example of such a pattern can be seen in Listing 1.1. The same pattern modelled for M^2ToS but in a more human readble form can be seen in figure 3.

Using the `ObjectTemplateExp` class from the QVT-R metamodel it is possible to create patterns that match objects using their types. `PropertyTemplateItems` are used to define properties of objects. They reference either other `ObjectTemplateExp` elements to represent object relations or they contain primitive values, such as integers, that can be expressed using literals. The `CollectionTemplateExp` class can be used to match elements within collections using a combination of quantifiers and conditions. However, the precise semantics of the `CollectionTemplateExp` is subject to discussion [13] which is why medini QVT does not implement it. Therefore, the `CollectionTemplateExp` is not used within the iterative generation of patterns.

An overview on the pattern mining approach is depicted in Figure 4. The process is defined across two different metalevels. First the transformations representing patterns are run against the model that should be mined. This takes place on the model level. Based on the outcome of this matching step the pattern transformations are extended and combined using transformations that work on the models of the pattern transformations. This can be considered as an operation one metalevel above, as these patterns are treated not as runnable transformations but rather as models that are transformed by the extension/combination transformations. For these higher-order (as they produce again

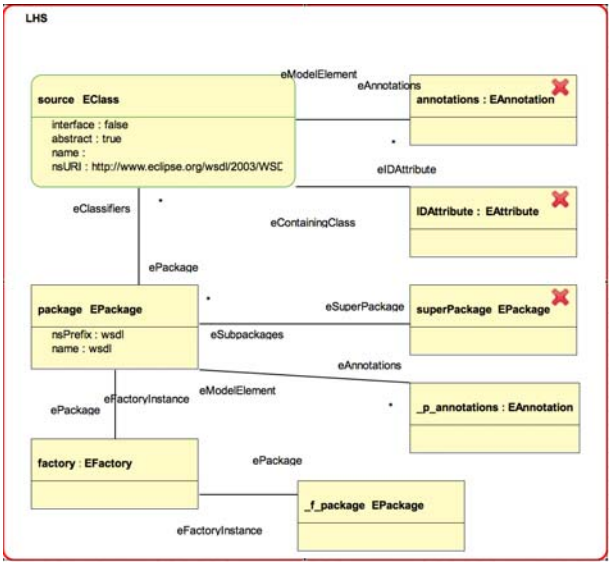


Fig. 3. Pattern from listing 1.1 modelled as M^2ToS Pattern

transformations) transformations the information from the metamodel of the currently mined model are taken into account. For example, during the extension phase it is necessary to know where the patterns can be extended. This information can be found in the metamodel. The modified set of patterns is then again queried for frequency within the model.

4.1 Linear Extension

The linear extension of patterns may lead to cycles resulting in a non-termination of the extension algorithm. Therefore, it is necessary to detect and prevent this situation. For example, a package having a reference to its contained classes where the classes also reference the package may lead to such a cycle. To prevent such extensions an additional check is needed during the extension of attributes. It has to be checked if instances of the extended attribute value already appeared in previous extensions. In this case the extension has to be stopped. Primitive attribute types can be excluded from this check, only class typed attributes are relevant here.

During linear extension new patterns are created for each attribute resulting in patterns that resemble a linked list. If a new attribute is added to the list only those attributes need to be checked for cycles where types of both attributes conform to each other. However cycle detection itself has to be performed on the model level as potentially (allowed due to the structure of the metamodel) possible cycles may not necessarily occur within the model. Model elements are, due to the defined pattern within this phase a directed acyclic graph and therefore a tree. Cycle detection can then be implemented using a depth first search algorithm.

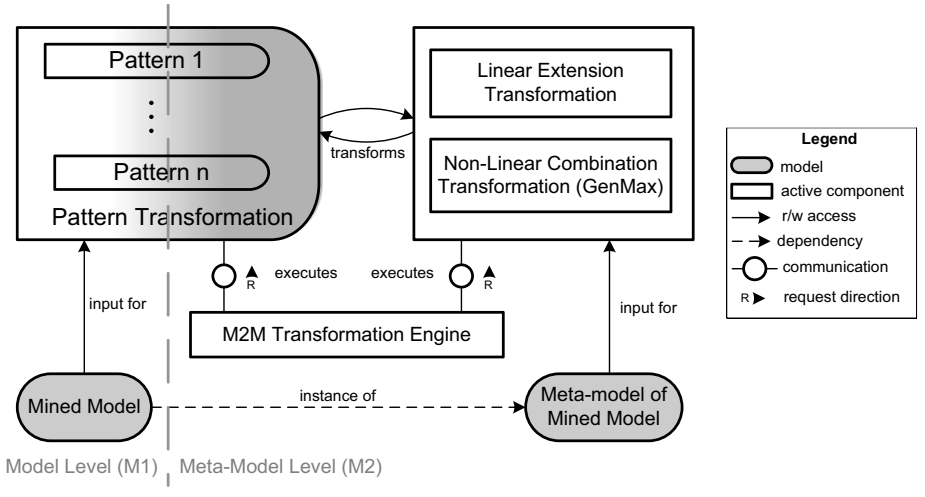


Fig. 4. Pattern mining with a model transformation engine

As OCL is an integral part of the QVT specification and capable of querying equality of model elements it is favourable to use it for cycle detection. Unfortunately OCL has no access to the metalevel of the used metamodel which is why a programmatic creation of OCL queries is required based upon the metamodel representation of the pattern. An example of such a cycle detection query is shown in 1.2 which detects the previously mentioned class and package cycle within ecore models.

Listing 1.2. Cycle detection using OCL: For all EClasses each ePackage is checked if current class is contained within the packages' eClassifiers

```
ecore::EClass.allInstances()->
    iterate(
        var0 : ecore::EClass;
        acc0 : Set(Boolean) = Set{} | acc0->including(
            (var0.ePackage.oclAsType(ecore::EPackage).eClassifiers->
            iterate(
                var1 : ecore::EClassifier ;
                acc1 : Set(Boolean) = Set{} | acc1->including (
                ( var1 = var0 )
                ) )->
            exists( b1 : Boolean | b1 = true )
            )
        ) )->
    exists( b0 : Boolean | b0 = true )
```

4.2 Non-linear - GenMax

Union of Graphs Like other heuristics the GenMax algorithm requires the union of elements of a set to create frequent itemsets. To be able to exploit the performance advantages of the GenMax algorithm on object-oriented pattern mining the union needs to be defined upon object graphs. As mentioned before cycle detection during the linear phase prevents generation of cyclic graphs as patterns. Therefore it is sufficient to define union on trees. As patterns that are created by the algorithm are composed of classes, attributes and instance values union has to be considered for these elements.

In contrast to unstructured sets the union of typed trees is not defined in all cases. For example, consider two classes with a different type which are not conform to each other but have a common base class. The union of these classes is undefined iff there is no type that contains the attributes of both classes. Therefore, a union of typed trees first of all has to account for their types. Formally, the union of two classes C_1 and C_2 having the respective types T_1 and T_2 is defined exactly when there is a class C being of type T where T conforms both to T_1 and T_2.

$$C_1 \cup C_2 = \begin{cases} C : T_\cup, \exists T_\cup : T_\cup \text{ conforms to } T_1 \wedge T_\cup \text{ conforms to } T_2 \\ \bot \; else \end{cases}$$

Note that if multiple inheritance is allowed a type $T_\cup$ may exist so that $T_\cup \neq T_1 \wedge T_\cup \neq T_2$. For single inheritance, the union corresponds to the most specific type.

As patterns may contain a partial set of the attributes of a class the union of attributes has to be considered, too. Two different cases of attribute union may occur. In the first case attribute names of the patterns to be unified differ so that the union contains both attributes and is therefore defined. In the second case the attribute names are equal. As attributes may either single or multi-valued these cases need also to be treated separately. If the attribute is single-valued the union can be done recursively from this point on and is defined iff the result of the recursive union is defined. If the attribute is multi-valued, union is defined, iff collections are interpreted as sets. This is a limitation in the sense that semantics of certain collections like sequences are discarded because the semantics of their union may be ambiguous. For example consider a sequence of elements A, B and a sequence of elements A, C. A union of these sequences may result in several new sequences like A, B, A, C or A, A, B, C effectively requiring a cross product between all elements to cover a full union. For the time being we defer a thorough investigation of this situation to future work.

Another case to consider is the union of patterns containing primitive instance values which may also be undefined. Suppose an attribute A of type *Integer*. Furthermore, suppose a pattern that contains attribute A having value $I_1 = 0$ and within another pattern having value $I_2 = 1$. A union of this attribute is not possible because no unique value can be used.

Ordering of Patterns. A complete enumeration of all patterns is the foundation for the GenMax algorithm. A requirement to the algorithm is that the enumeration must be strictly totally ordered. For simple sets an order is mostly either defined lexically or by the order of primitive values. As in our case patterns are typed trees ordering has to be defined upon these.

It is worth to mention that the GenMax algorithm strictly relies upon the ordering and not upon the contents of items to be ordered. Therefore we can choose one order out of several possible orders that may be defined for object graphs. In our preliminary version we order patterns starting with their root nodes based upon their class types. We order incompatible class types according to their class name whereas compatible class types are ordered according to the most general class type. If class types match, we order patterns according to their attributes. First we sort attributes of both classes by name. If names mismatch, ordering is determined by their lexicographical order. If both names match, we apply the described ordering recursively to both attributes. If attribute types are primitive we resort to their natural ordering.

We consider our chosen order preliminary as a order may have significant impact on the performance of the heuristic. Initial investigation suggests that query times differ significantly depending on the structure of mined patterns therefore affecting overall performance.

5 Experimental Results

5.1 Setup

All our experiments were performed on a 2.33GHz Intel Xeon E5545 dual quad-core processor with $4GB$ physical memory, running Debian Etch 64bit and Sun Java 1.5.0.11. As the current implementation is not capable of using multiple cores only one out of eight cores were used for mining. The application was given a fair amount of $3GB$ RAM as the mining algorithm does not use tertiary memory, yet. The times were reported by the logging framework Log4j as it is known to cause low impact upon total runtime performance.

Although our implementation is not restricted to any kind of metamodel, we chose ecore model instances for evaluation as there are many models with different characteristics available within the eclipse project. We focused our evaluations on two distinctive classes of the ecore model namely "EAnnotation" and "EClass". We consider class "EAnnotation" as structurally weak as it contains only seven attributes and references whereas class "EClass" is considered structurally strong having 24 attributes and references. Mining "EAnnoation" is expected to be a basic task whereas mining "EClass" is expected to be a complex task.

To estimate the impact of the amount of class instances mining was performed upon three different models for each of the considered classes. Models were retrieved from several CVS repositories from within the eclipse project. For class "EAnnotation" models BPEL, GMFGEN and UML ($26, 229, 3017$ EAnnotations) were taken into account. Models used for mining on class "EClass" were EMOF, XSD and UML ($21, 56, 243$ EClasses).

5.2 Results

An example of a mined pattern from an "EAnnotation" can be seen in Listing 1.3. This pattern is one out of three frequent patterns which was mined from the Eclipse UML

model given a support of 10%. Annotations are dominantly used for documentation and as this pattern shows many elements are documented with the same documentation string of “The cache of context-specific information.”.

Listing 1.3. One out of three patterns mined with 10 % support upon the UML model

```
1  enforce domain source var8609 : ecore::EAnnotation {
2    details = var8610 : ecore::EStringToStringMapEntry {
3      value = 'The cache of context-specific information.',
4      key = 'documentation'
5      },
6    source = 'http://www.eclipse.org/emf/2002/GenModel',
7    eModelElement = var8611 : ecore::EModelElement {
8    eAnnotations = var8612 : ecore::EAnnotation { }
9    }
10 };
```

The overall performance of our implementation is shown in table 1. As expected structure weak “EAnnotation” can be mined much faster than structure strong “EClass” for all models. Mining patterns of class “EAnnotation” completed within acceptable time delivering frequent patterns. The time to mine frequent patterns increased if support was lowered. On the other hand mining class “EClass” is such a time consuming task that mining could not be completed due to memory shortage. Even by applying a rigorous filter that discards all class attributes which names begin with *eAll* and limiting the generation of linear patterns to 20 mining could not be completed for some models in case of low support.

Table 1. Overall performance of the implementation mining EAnnotations (EAn.) and EClasses

Ecore Model	Mining Class	Support %	Total time (MM:SS)	Model Queries #	Frequent Patterns #	Limit #	Filter	Ecore Model	Mining Class	Support %	Total time (MM:SS)	Model Queries #	Frequent Patterns #	Limit #	Filter
BPEL	EAn.	10%	0:01	70	1	∞	–	EMOF	EClass	95%	0:02	222	1	20	w/o eAll
GMFGEN	EAn.	95%	0:04	271	1	∞	–	EMOF	EClass	75%	0:05	220	1	20	w/o eAll
GMFGEN	EAn.	75%	0:04	277	1	∞	–	EMOF	EClass	50%	0:04	376	2	20	w/o eAll
GMFGEN	EAn.	50%	0:04	284	1	∞	–	EMOF	EClass	25%	N/A	N/A	N/A	20	w/o eAll
GMFGEN	EAn.	25%	0:04	284	1	∞	–	XSD	EClass	95%	0:03	228	1	20	w/o eAll
GMFGEN	EAn.	10%	0:21	1663	4	∞	–	XSD	EClass	75%	0:07	220	1	20	w/o eAll
UML	EAn.	95%	0:12	18	1	∞	–	XSD	EClass	50%	N/A	N/A	N/A	20	w/o eAll
UML	EAn.	75%	2:06	1681	1	∞	–	UML	EClass	95%	0:14	223	1	20	w/o eAll
UML	EAn.	25%	2:09	1681	1	∞	–	UML	EClass	75%	0:27	620	2	20	w/o eAll
UML	EAn.	10%	5:41	4126	3	∞	–	UML	EClass	50%	13:16	214	1	20	w/o eAll
UML	EAn.	5%	10:21	5388	3	∞	–	UML	EClass	25%	N/A	N/A	N/A	20	w/o eAll

6 Applying Pattern Mining to Estimate Maintainability of Models

Maintainability has been identified as a key factor for software development costs and may account for up to 80% of total costs [14]. Therefore measuring maintainability and in a broader sense software quality has been subject to research both within functional as well as object oriented software development to improve software quality and reduce maintenance costs. While metrics for measuring object oriented maintainability have been defined, model driven software development (MDSD) is still lacking this key component. Only selected metamodels such as the Unified Markup Language (UML) have been investigated with regards to quality. Comprehensive, comparable and automatically collectible metrics that show strong correlations between measured quality factors are required to ensure low maintainability costs.

Focusing on object oriented methodologies attempting to measure software quality led to the definition of metric suites such as those of Chidamber and Kemerer [15] and metrics for object-oriented design (MOOD) [16]. The suites have been adopted to UML with restrictions in [17] and [18] resulting in a weak external validity of measures. Although this drawback could be accepted for the sake of having similar measures at least these metrics can not be adopted to arbitrary metamodels and model instances thereof. Suppose model instances of a metamodel which models finite state machines. The notion of inheritance does not exist within such a metamodel thus the metric depth of inheritance is useless in context of finite state machines.

As a software project may employ several different domain specific languages quality issues should be addressed for all models used. Manually defining quality metrics for each model may be a time consuming task so a generic automated approach to measure maintainability within models seems more favorable.

Several definitions of maintainability have been proposed i.e. in IEEE 610.12 or ISO/IEC 12207 and 14764. IEEE 610.12 defines maintainability as “the ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment.” This definition is rather imprecise in a sense that it leaves concerned quality factors up for interpretation. As a constructive approach to maintainability we use the ISO 9126 quality model and follow this interpretation as maintainability being composed of stability, analysability, changeability and testability. We focus our efforts upon analysability and use this quality factor to direct our work.

6.1 Related Work within Maintainability Metrics

Some commonly used models have been surveyed with respect to quality. Quality for UML use case diagrams have been investigated in [19] and addressed through a checklist for these kinds of diagrams. The Chidamber and Kemerer metric suite has been defined for UML models in [17] by means of the Object Constraint Language (OCL). The MOOD metric suite has been transfered to the UML in [18]. Several other metrics such as number of aggregations or number of classes have been proposed in [20]. The UML metamodel itself has been investigated with respect to quality attributes in [21] with metrics such as “Average Number of Stereotypes” and “Number of Hierarchies”. [22] defined metrics for assessing OCL expressions in a structured way through a Goal-Question-Metric approach. Their definitions align to human cognitive concepts such as

tracing and chunking. Ideas on how to measure model transformations can be found in [23] and [24] but up to now metrics have not been defined for model transformations, yet. However all these approaches are metamodel-specific and not generalisable to different kinds of metamodels therefore eliminating comparability.

6.2 Pattern Mining as Maintainability Measure

Human analysis of software products are conducted either top-down or bottom-up according to [14]. Using a top-down approach the analyst tries to apply his knowledge about design and domain to classify the software product under analysis. In order to do this he tries to gain an overview of the whole application. He will then successively pick selected software segments and determine their relevance for his current mental model of the software. Using a bottom-up approach the analyst will start reading comments of source code or other software artifacts. The control flow of certain sections will then be inspected sequentially and selected variables will be traced throughout the flow. The information gained will be integrated to a mental software model which opposed to the top-down approach does not contain any information as to why the model is characterized the way it presents itself. [14] notes that top-down analysis is being conducted more often by experts whereas bottom-up analysis is being used more often by novice analysts.

These findings give strong indication that experts may have abstract mental patterns at hand which are being used for analyzing the software product whereas novices must resort to documentation. If analysability is measured in terms of time to analyze parts of a software product the required time will be low if the analyzed parts dominantly adhere to the expert's patterns. On the other hand the time will be very high, if the expert can apply only a few of his patterns or the software heavily differs from patterns known to him. These general observations were also stated for visual patterns in [25] which is why we propose to incorporate them into a analysability metric.

As a first step towards such a metric we propose to identify patterns automatically through the previously defined pattern mining approach upon the abstract syntax of models. A second step that involves determining the distance from several patterns to a model instance will be subject to future research.

7 Conclusion and Future Work

In this paper, we presented an approach that makes use of QVT Relations transformations to execute pattern mining in order to find frequent patterns within models. Thus, we leverage the well known mining algorithm apriori and some of its variations to the model level. Initial evaluation showed that structurally weak classes may be mined within a acceptable timeframe whereas mining structurally strong classes currently can not be mined efficiently.

In short terms our future work will focus upon improving overall performance of our approach. We identified several spots such as ordering of graph structures, usage of graph optimized heuristics, using a different query engine like M^2ToS or partially storing the search space to tertiary memory. A more formal definition of a quality metric based upon mined patterns will then be subject to our research as well as its validation.

Future work will investigate finding of appropriate approaches and metrics for measuring the maintainability and evolution capabilities of artifacts within model-driven environments. More precisely, a metamodel independent measurement method motivated by the concept of decomposition is presented that finds frequent patterns within models by means of automated frequent pattern mining. In our work we emphasized that current object oriented metrics are not suitable to address maintainability within a model driven context. We proposed a more general approach to define quality metrics that is independent of the metamodel used.

There are also other areas where the presented pattern mining approach can be used to detect frequent patterns in models. One idea which may give interesting results is the analysis of user behavior during modeling. In many evolution scenarios it is crucial to know what impact a certain change has. So, for example, in the evolution of metamodels there are changes that have a huge impact on the instances of the metamodel. Certain metamodel changes result in invalid instances of the metamodel. One solution to this problem is to allow only the user to make certain refactorings to the metamodel which have a known impact on the instances that can then be adopted accordingly. Some refactorings, such as renaming are already known from experiences programming languages. However to find out which changes are also frequently done in some form of a pattern, it would be interesting to use the presented pattern mining approach on the logs of editing operations (which can also be stored as models, as e.g., in EMF with it's EMF-Change recording). Results of this mining might then be analyzed if possible additional refactoring operations can be provided.

References

1. Han, J., Kamber, M.: Data Mining: Concepts and Techniques. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, San Francisco (2000)
2. Kuba, P., Popelinsky, L.: Mining frequent patterns in object-oriented data. In: Proceedings of the 2nd International Workship on Mining Graphs, Trees and Sequences, pp. 15–25 (2004)
3. Reichmann, C.: Graphisch notierte Modell-zu-Modell-Transformationen für den Entwurf eingebetteter elektronischer Systeme. Ph.D thesis, University Karlsruhe (2005)
4. Object Management Group: Meta Object Facility (MOF) 2.0 Query/View/Transformation (QVT), http://www.omg.org/docs/formal/08-04-03.pdf
5. Garboni, C., Masseglia, F., Trousse, B.: Sequential Pattern Mining for Structure-Based XML Document Classification. In: Fuhr, N., Lalmas, M., Malik, S., Kazai, G. (eds.) INEX 2005. LNCS, vol. 3977, pp. 458–468. Springer, Heidelberg (2006)
6. Klukas, C., Koschutzki, D., Schreiber, F.: Graph pattern analysis with patterngravisto. Journal of Graph Algorithms and Applications 9, 19–29 (2005)
7. Zhao, C., Kong, J., Dong, J., Zhang, K.: Pattern based design evolution using graph transformation. Journal of Visual Languages and Computing (JVLC) 18(4), 378–398 (2007)
8. Mazón, J.N., Pardillo, J., Trujillo, J.: Applying transformations to model driven data warehouses. In: Tjoa, A.M., Trujillo, J. (eds.) DaWaK 2006. LNCS, vol. 4081, pp. 13–22. Springer, Heidelberg (2006)
9. Burdick, D., Calimlim, M., Gehrke, J.: Mafia: A maximal frequent itemset algorithm for transactional databases. In: Intl. Conf. on Data Engineering, pp. 443–452 (2001)
10. Gouda, K., Zaki, M.: Genmax: An efficient algorithm for mining maximal frequent itemsets. Data Mining and Knowledge Discovery 11(3), 223–242 (2005)

11. Yang, G.: The complexity of mining maximal frequent itemsets and maximal frequent patterns. In: KDD 2004: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 344–353. ACM, New York (2004)
12. ikv++: ikv++ mediniQVT, http://projects.ikv.de/qvt (last access, 20-12-2007)
13. ikv++: Discussion CollectionTemplateExp., http://projects.ikv.de/qvt/discussion/1/12 (last access, 06-04-2009)
14. Masak, D.: Legacysoftware. Springer, Heidelberg (2005)
15. Chidamber, S., Kemerer, C.: A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20(6), 476–493 (1994)
16. Abreu, F., Brito, R.: Objectoriented software engineering: Measuring and controlling the development process (1994)
17. McQuillan, J., Power, J.: A definition of the chidamber and kemerer metrics suite for uml. Technical report, National University of Ireland (2006)
18. e Abreu, F.B.: Using ocl to formalize object oriented metrics definitions. Technical report, FCT/UNL and INSC (2001)
19. Phalp, K.T., Vincent, J., Cox, K.: Assessing the quality of use case descriptions. Software Quality Control 15(1), 69–97 (2007)
20. Ma, H., Shao, W., Zhang, L., Ma, Z., Jiang, Y.: Applying oo metrics to assess uml metamodels. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) UML 2004. LNCS, vol. 3273, pp. 12–26. Springer, Heidelberg (2004)
21. Ma, H., Shao, W., Zhang, L., Ma, Z., Jiang, Y.: Applying oo metrics to assess uml metamodels. In: Baar, T., Strohmeier, A., Moreira, A., Mellor, S.J. (eds.) UML 2004. LNCS, vol. 3273, pp. 12–26. Springer, Heidelberg (2004)
22. Reynoso, L., Genero, M., Piattini, M., Manso, E.: Assessing the impact of coupling on the understandability and modifiability of ocl expressions within uml/ocl combined models. In: 11th IEEE International Symposium on Software Metrics, September 19-22, 10 pp. (2005)
23. Goldschmidt, T., Kuebler, J.: Towards Evaluating Maintainability Within Model-Driven Environments. In: Software Engineering 2008, Workshop Modellgetriebene Softwarearchitektur - Evolution, Integration und Migration (2008)
24. Saeki, M., Kaiya, H.: Measuring model transformation in model driven development. In: Proceedings of the CAiSE 2007 Forum at the 19th International Conference on Advanced Information Systems Engineering. CEUR Workshop Proceedings, CEUR-WS.org, vol. 247 (2007)
25. Solso, R.L.: Cognitive Psychology. Allyn and Bacon (2001)

A Language-Theoretic View on Guidelines and Consistency Rules of UML

Zhe Chen[1] and Gilles Motet[1,2]

[1] Laboratory LATTIS, INSA, University of Toulouse,
135 Avenue de Rangueil, 31077 Toulouse, France
zchen@insa-toulouse.fr
[2] Foundation for an Industrial Safety Culture,
6 Allée Emile Monso, 31029 Toulouse, France
gilles.motet@insa-toulouse.fr

Abstract. Guidelines and consistency rules of UML are used to control the degrees of freedom provided by the language to prevent faults. Guidelines are used in specific domains (e.g., avionics) to recommend the proper use of technologies. Consistency rules are used to deal with inconsistencies in models. However, guidelines and consistency rules use informal restrictions on the uses of languages, which makes checking difficult. In this paper, we consider these problems from a language-theoretic view. We propose the formalism of C-Systems, short for "formal language control systems". A C-System consists of a controlled grammar and a controlling grammar. Guidelines and consistency rules are formalized as controlling grammars that control the uses of UML, i.e. the derivations using the grammar of UML. This approach can be implemented as a parser, which can automatically verify the rules on a UML user model in XMI format. A comparison to related work shows our contribution: a generic top-down and syntax-based approach that checks language level constraints at compile-time.

1 Introduction

The UML (Unified Modeling Language) is a graphic modeling language developed by OMG (Object Management Group), and defined by the specifications [1] and [2]. UML has emerged as the software industry's dominant modeling language for specifying, designing and documenting the artifacts of systems [3] [4].

Evolving descriptions of software artifacts are frequently inconsistent, and tolerating this inconsistency is important [5] [6]. Different developers construct and update these descriptions at different times during development [7], thus resulting in inconsistencies. They develop multiple views on a system providing pieces of information which are redundant or complementary. Constraints exist on these pieces of information whose violation leads to inconsistent models. Inconsistency problems of UML models have attracted great attention from both academic and industrial communities [8] [9] [10]. A list of 635 consistency rules are identified by [11] [12].

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, pp. 66–81, 2009.

Guidelines, which also contain a set of rules, are often required on models which are specific to a given context. For instance, OOTiA (Object-Oriented Technology in Aviation) demands that "the length of an inheritance should be less than 6" [13]. This context is domain specific. If these constraints are not respected, the presence of faults is not sure but its risk is high. The context can also be technology specific. For instance, "multiple inheritance should be avoided in safety critical, certified systems" (IL #38 of [13]), if the UML models are implemented by Java code, as this language does not provide the multiple inheritance mechanism.

It seems that consistency rules and guidelines are irrelevant at first glance. However, in fact, they have the same origin from a language-theoretical view. We noticed that both of the two types of potential faults in models come from *the degrees of freedom* offered by languages. These degrees of freedom cannot be eliminated without reducing the language capabilities [14]. For instance, the multiple diagrams in UML are useful, as they describe various viewpoints on one system, even if they are at the origin of numerous inconsistencies. In the same way, multiple inheritance can be implemented in the C++ language.

To prevent these risks of faults, *the use of languages* must be controlled. To do it, guidelines are old and popular means in industry. However, their expression is informal and their checking is difficult. For instance, 6 months were needed to check 350 consistency rules on an avionics UML model including 116 class diagrams.

This paper aims at formalizing *the acceptable use of languages* and proposing a way to check *the use correctness*, by considering guidelines and consistency rules from a language-theoretical view. To achieve this goal, acceptable uses of a language are defined as a grammar handling the productions of the grammar of the language. To support this idea, UML must be specified by a formal language, or at least a language with precisely defined syntax, e.g., XMI in this paper. Thus, a graphic model can be serialized. This formalism also provides a deeper view on the origin of inconsistencies in models.

This paper is organized as follows. First, we introduce the grammar of UML in XMI in Section 2. Then in Section 3, we define the *C-System*, i.e. a formalism containing controlling grammars that restrict the use of the grammar of UML. We illustrate the formalism using examples in Section 4. Related work and implementation of this approach are discussed in Sections 5 and 6. Section 7 concludes the paper.

2 The Grammar of UML in XMI

XMI (XML Metadata Interchange) [15] is used to facilitate interchanging UML models between different modeling tools in XML format. Many tools implement the conversion, e.g., Altova UModel® can export UML models as XMI files.

A UML model in XMI is an XMI-compliant XML document that conforms to its XML schema, and is a derivative of the *XMI document productions* which is defined as a grammar. The XML schema is a derivative of the *XMI schema*

productions. The XMI specification defines both the XMI schema productions and the XMI document productions in [15].

XMI provides a mapping between a UML user model and an XML document, and a mapping between UML (also MOF) and an XML Schema. XMI generates an XML file using the XMI document productions, and generates an XML schema using the XMI schema productions. Each of the two sets of productions composes a context-free grammar in EBNF [16]. A UML user model can be expressed using an XMI-compliant XML document that conforms to the corresponding XML Schema, and is a derivative of the XMI document grammar.

The grammar and its productions for deriving XMI-compliant XML documents of UML models are defined in [15]. The main part of the grammar is given here after. To make our presentation more concise, we omit declaration and version information of XML files (and the related productions whose names start with "1").

To make later reasoning easier, we modified some representations of the productions, but without changing the generative power of the grammar.

1. The choice operator "|" is used to compose several productions with the same left-hand side into a single line in [15]. We decomposed some of these productions into several productions without the choice operator. An original production n having k choices might be divided into a set of productions $\{n\_i\}_{1 \leq i \leq k}$. For example, the original production 2 with three choices was divided into the productions 2_1, 2_2 and 2_3.

2. The closure operator "*" is used to simplify the representation of the grammar in [15], but it also would make the representation of reasoning confusing. Thus, the productions whose names start with "3" were added to replace the productions with closure operators.

The **grammar** G of UML in XMI includes the following productions (each production is labeled by a name starting with a digit):

```
3_1:  XMIElements ::= 2:XMIElement
3_2:  XMIElements ::= 2:XMIElement 3:XMIElements

2_1:  XMIElement ::= 2a:XMIObjectElement
2_2:  XMIElement ::= 2b:XMIValueElement
2_3:  XMIElement ::= 2c:XMIReferenceElement

2a_1: XMIObjectElement ::= "<" 2k:QName 2d:XMIAttributes "/>"
2a_2: XMIObjectElement ::= "<" 2k:QName 2d:XMIAttributes ">"
                           3:XMIElements "</" 2k:QName ">"

2b_1: XMIValueElement ::= "<" xmiName ">" value "</" xmiName ">"
2b_2: XMIValueElement ::= "<" xmiName "nil='true'/>"

2c_1: XMIReferenceElement::= "<" xmiName 2l:LinkAttribs "/>"
2c_2: XMIReferenceElement::= "<" xmiName 2g:TypeAttrib
                             2l:LinkAttribs "/>"
```

```
2d_1: XMIAttributes ::= 2g:TypeAttrib  2e:IdentityAttribs
                        3h:FeatureAttribs
2d_2: XMIAttributes ::= 2e:IdentityAttribs  3h:FeatureAttribs

2e:   IdentityAttribs ::= 2f:IdAttribName "=‘" id "’"

2f_1: IdAttribName ::= "xmi:id"
2f_2: IdAttribName ::= xmiIdAttribName

2g:   TypeAttrib ::= "xmi:type=‘" 2k:QName "’"

3h_1: FeatureAttribs ::= 2h:FeatureAttrib
3h_2: FeatureAttribs ::= 2h:FeatureAttrib  3h:FeatureAttribs

2h_1: FeatureAttrib ::= 2i:XMIValueAttribute
2h_2: FeatureAttrib ::= 2j:XMIReferenceAttribute

2i:   XMIValueAttribute ::= xmiName "=‘" value "’"

2j:   XMIReferenceAttribute ::= xmiName "=‘" (refId | 2n:URIref)+"’"

2k:   QName ::= "uml:" xmiName  |  xmiName

2l:   LinkAttribs ::= "xmi:idref=‘" refId "’" | 2m:Link

2m:   Link ::= "href=‘" 2n:URIref "’"

2n:   URIref ::= (2k:QName)? uriReference
```

In the grammar, the symbol "::=" stands for the conventional rewriting symbol "→" in formal languages theory [17]. Each nonterminal starts with a capital letter, prefixing a label of the related production, e.g., "2:XMIElement" is a nonterminal with possible productions "2_1, 2_2, 2_3". Each terminal starts with a lowercase letter or is quoted.

As an example to illustrate the use of the grammar, Figure 1 represents a package *Root* which includes three classes, where the class *FaxMachine* is derived from *Scanner* and *Printer*. The core part of the exported XMI 2.1 compliant file (using Altova UModel®) is as follows:

```
<uml:Package xmi:id="U00000001-7510-11d9-86f2-000476a22f44"
             name="Root">
     <packagedElement xmi:type="uml:Class"
             xmi:id="U572b4953-ad35-496f-af6f-f2f048c163b1"
             name="Scanner" visibility="public">
          <ownedAttribute xmi:type="uml:Property"
             xmi:id="U46ec6e01-5510-43a2-80e9-89d9b780a60b"
          name="sid" visibility="protected"/>
```

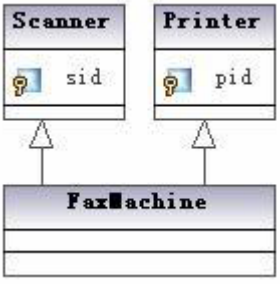


Fig. 1. A Class Diagram

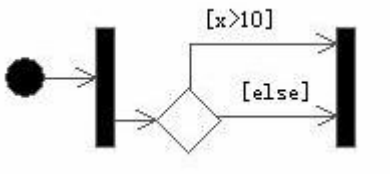


Fig. 2. An Activity Diagram

```
    </packagedElement>
    <packagedElement xmi:type="uml:Class"
            xmi:id="Ua9bd8252-0742-4b3e-9b4b-07a95f7d242e"
            name="Printer" visibility="public">
        <ownedAttribute xmi:type="uml:Property"
            xmi:id="U2ce0e4c8-88ee-445b-8169-f4c483ab9160"
        name="pid" visibility="protected"/>
    </packagedElement>
    <packagedElement xmi:type="uml:Class"
            xmi:id="U6dea1ea0-81d2-4b9c-aab7-a830765169f0"
            name="FaxMachine" visibility="public">
        <generalization xmi:type="uml:Generalization"
            xmi:id="U3b334927-5573-40cd-a82b-1ee065ada72c"
            general="U572b4953-ad35-496f-af6f-f2f048c163b1"/>
        <generalization xmi:type="uml:Generalization"
            xmi:id="U86a6818b-f7e7-42d9-a21b-c0e639a4f716"
            general="Ua9bd8252-0742-4b3e-9b4b-07a95f7d242e"/>
    </packagedElement>
</uml:Package>
```

This text is a derivative of the XMI document productions, c.f. the previous grammar G. We may use the sequence of productions "2a_2, 2k(Package), 2d_2, 2e, 2f_1, 3h_1, 2h_1, 2i" to derive the following sentential form:

```
<uml:Package xmi:id="U00000001-7510-11d9-86f2-000476a22f44"
            name="Root">
    3:XMIElements "</" 2k:QName ">"
```

Note that the production 2k has a parameter *xmiName*, i.e. the value of the terminal when apply the production. In a derivation, we specify a value of the parameter as "2k(value)". For example, "2k(Package)" is a derivation using 2k with *xmiName* = "Package". For simplicity, we consider "2k(value)" as a terminal as a whole. We continue to apply productions, and finally derive the XMI file previously presented.

Notice that the model of Fig. 1 (both in UML and XML) does not conform to the guidelines in OOTiA about multiple inheritance, since it uses multi-inheritance. The model of Fig. 2 has an inconsistency: "the number of outgoing edges of $ForkNode$ is not the same as the number of incoming edges of $JoinNode$". In particular, $JoinNode$ joins two *outgoing* edges from the same $DecicionNode$, This join transition will never be activated, since only one of the two *outgoing* edges will be fired.

We will define a formal model to check the conformance to these rules by controlling the use of the grammar of UML.

3 The C-System: A Formal Language Control System

In this section, we propose the formal model for controlling the use of grammars based on classical language theory [17].

Let $G = (N, T, P, S)$ be a grammar, where N is the set of nonterminals, T is the set of terminals, P is the set of productions of the form $l : A \to \alpha$ where l is the name of the production, $A \in N$, $\alpha \in (N \bigcup T)^*$, and S is the start symbol. A derivation using a specified production p is denoted by $\alpha \overset{p}{\Rightarrow} \beta$, and multiple derivations are denoted by $\alpha \Rightarrow^* \gamma$.

Definition 1. *A **controlling grammar** $\hat{G}$ over a **controlled grammar** (or simply grammar) $G = (N, T, P, S)$ is a quadruple $\hat{G} = (\hat{N}, \hat{T}, \hat{P}, \hat{S})$, where $\hat{T} = P$. The language $L(\hat{G})$ is called a **controlling language**.* □

The symbol $\hat{G}$ is read "control G" or "G hat". For making reading easier, we assume that $N \cap \hat{N} = \emptyset$. $\hat{T} = P$ means that the terminals of $\hat{G}$ are exactly the productions of G.

If we use an automaton A to process the input string, such that $L(A) = L(G)$, then we can also use a **controlling automaton** $\hat{A}$ to represent the controlling language.

As we know, each string $w \in L(G)$ has at least one *leftmost* derivation (denoted by "lm") using a sequence of productions from P, e.g. $p_1 p_2 ... p_k$. The controlling grammar restricts the derivation in the sense that the sequences of applied productions should be in the language it specifies, i.e., $p_1 p_2 ... p_k \in L(\hat{G})$. Formally, we have the following definition.

Definition 2. *Given a grammar $G = (N, T, P, S)$, the language of the grammar with a controlling grammar $\hat{G}$ is:*

$$L(G^{\overrightarrow{\cdot}}\hat{G}) = \{w | S \underset{lm}{\overset{p_1}{\Rightarrow}} w_1 \cdots \underset{lm}{\overset{p_k}{\Rightarrow}} w_k = w,\ p_1, p_2, ..., p_k \in P \text{ and } p_1 p_2 ... p_k \in L(\hat{G})\}$$

*We say that G and $\hat{G}$ constitute a **C-System** $C = G^{\overrightarrow{\cdot}}\hat{G}$, short for **formal language control system**. The language $L(C) = L(G^{\overrightarrow{\cdot}}\hat{G})$ is called a **global system language**.* □

The symbol $\overrightarrow{\cdot}$ is called "meta composition". Its left operand is controlled by the right operand, which is a meta level grammar. If we use automata-based

notations, a string $w \in L(A \overrightarrow{\cdot} \hat{A})$ if and only if A accepts w, and $\hat{A}$ accepts the sequence of the labels of the transitions used.

A ***regular C-System*** is a C-System of which the controlled grammar G is a regular grammar (or A is a finite automaton). Some variants of regular C-Systems are proposed for ensuring system safety requirements, e.g. Input/Output C-Systems [18], Interface C-Systems [19] [20]. We denote by $\mathcal{C}_R$ the family of regular C-Systems.

A ***context-free C-System*** is a C-System of which the controlled grammar G is a context-free grammar (or A is a pushdown automaton). We denote by $\mathcal{C}_{CF}$ the family of context-free C-Systems.

Generally, we denote by $\mathcal{C}_X^Y$ the family of C-Systems that consist of X-type controlled grammar and Y-type controlling grammar, where $X, Y \in \{R, CF\}$. Although X, Y could be also other types in Chomsky hierarchy, e.g. context-sensitive, this is beyond the scope of this paper.

Obviously, the set of accepted inputs is a subset of the controlled language, such that the sequence of the applied productions belongs to the controlling language. Consider a simple example as follows.

Example 1. Given a regular grammar G and a regular controlling grammar $\hat{G}$:

$$G \begin{cases} p_1 : S \to aS \\ p_2 : S \to bS \\ p_3 : S \to \epsilon \end{cases} \qquad \hat{G} \begin{cases} \hat{S} \to p_1\hat{S} | p_3\hat{S} | p_2 A \\ A \to p_2 A | p_3 A | \epsilon \end{cases}$$

$L(G)$ accepts the language $(a|b)^*$, e.g., *aab*, *abab*. $L(\hat{G})$ accepts the language $(p_1|p_3)^* p_2 (p_2|p_3)^*$. The trivial grammar G is considered to provide a simple illustration of the introduced principles.

The grammars G and $\hat{G}$ constitute a regular C-System $C = G \overrightarrow{\cdot} \hat{G} \in \mathcal{C}_R^R$.

Given the string $aab \in L(G)$, we conclude that $aab \in L(G \overrightarrow{\cdot} \hat{G})$, because we have the leftmost derivations $S \overset{p_1}{\Rightarrow} aS \overset{p_1}{\Rightarrow} aaS \overset{p_2}{\Rightarrow} aabS \overset{p_3}{\Rightarrow} aab$, where $p_1p_1p_2p_3 \in L(\hat{G})$ as $\hat{S} \Rightarrow p_1\hat{S} \Rightarrow p_1p_1\hat{S} \Rightarrow p_1p_1p_2A \Rightarrow p_1p_1p_2p_3A \Rightarrow p_1p_1p_2p_3$. On the contrary, we have $abab \notin L(G \overrightarrow{\cdot} \hat{G})$. Although we have the leftmost derivation $S \overset{p_1}{\Rightarrow} aS \overset{p_2}{\Rightarrow} abS \overset{p_1}{\Rightarrow} abaS \overset{p_2}{\Rightarrow} ababS \overset{p_3}{\Rightarrow} abab$, $p_1p_2p_1p_2p_3 \notin L(\hat{G})$.

In fact, the language $L(C) = L(G \overrightarrow{\cdot} \hat{G})$ is equivalent to the language a^*b^+, which is the subset of $(a|b)^*$ satisfying the constraints: "every a should appear before b" and "at least one b". □

We remark here that our model is different from regularly controlled grammars [21] [22], in the sense that we restrict derivations to be *leftmost* and allow *context-free controlling grammars*. These differences result in different theoretical results, which are beyond the scope of this paper.

4 Examples

In this section we use some practical examples to illustrate the idea of the previous section. We denote the grammar of UML by $G = (N, T, P, S)$, where P is

the set of productions listed in Section 2, and each production $p \in P$ is labeled by a name starting with a digit.

Example 2. Consider two rules on class diagrams:

Rule 1: Each class can have at most one generalization. This rule is a guideline, as we mentioned in Section 1 and at the end of Section 2. This rule is also a consistency rule in the context of Java, since Java does not allow multiple inheritance. However we may derive a class from multiple classes in the context of C++.

Rule 2: Each class can have at most 30 attributes. This rule may be adopted by software authorities as a guideline in avionics, in order to increase the safety of software systems by minimizing the complexity of classes.

Note that these rules cannot be explicitly integrated into the grammar of UML, but only recommended as guidelines or consistency rules. We cannot put rule 1 into the standard of UML, since UML models can be implemented with both C++ and Java programming languages. Rule 2 is a restriction for a specific domain, and we should not require all programmers to use limited attributes by specifying the UML language.

We aim to specify the rules from the meta-language level, thus control the use of the language. Consider the example of Fig. 1, to obtain the associated XMI text, the sequence of applied productions of G in the leftmost derivation is as follow ("..." stands for some omitted productions, to save space):

```
2a_2, 2k(Package), 2d_2, 2e, 2f_1, 3h_1, 2h_1, 2i,
..., 2k(packagedElement), ..., 2k(Class),
      ..., 2k(ownedAttribute), ..., 2k(Property),
..., 2k(packagedElement),
..., 2k(packagedElement), ..., 2k(Class),
      ..., 2k(ownedAttribute), ..., 2k(Property),
..., 2k(packagedElement),
..., 2k(packagedElement), ..., 2k(Class),
      ..., 2k(generalization), ..., 2k(Generalization),
      ..., 2k(generalization), ..., 2k(Generalization),
..., 2k(packagedElement),
..., 2k(Package)
```

Let c, g stand for $2k(Class), 2k(Generalization)$, respectively. Note that the occurrence of two g after the third c violates Rule 1. In fact, all the sequences of productions in the pattern "*...c...g...g...*" are not allowed by the rule (there is no c between the two g), indicating that the class has two generalizations.

Thus, we propose the following controlling grammar $\hat{G}_c$ to restrict the use of the language to satisfy Rule 1:

$$\hat{G}_c \begin{cases} S \rightarrow c\ \ Q_c \mid D\ S \mid D \\ Q_c \rightarrow c\ \ Q_c \mid g\ \ Q_g \mid D\ Q_c \mid D \\ Q_g \rightarrow c\ \ Q_c \mid D\ Q_g \mid D \\ D \rightarrow \{p \mid p \in P\ \wedge\ p \notin \{c, g\}\} \end{cases} \tag{1}$$

where S, Q_c, Q_g, D are nonterminals, D includes all productions except c, g. $L(\hat{G}_c)$ accepts the sequences of productions satisfying Rule 1.

Implicitly, the controlling grammar specifies an automaton $\hat{A}_c$ in Fig. 3, where $\lightning$ is an implicit error state (the dashed circle). Strings of the pattern $D^*cD^*gD^*gD^*$ will lead $\hat{A}_c$ to the error state.

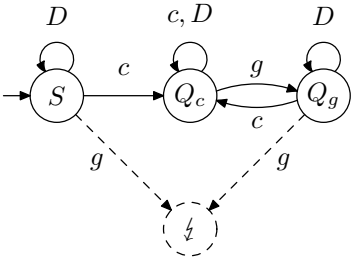


Fig. 3. The Automaton $\hat{A}_c$

If the sequence of productions applied to derive a model is accepted by the language $L(\hat{G}_c)$, then the model conforms to Rule 1. In Fig. 1, the derivation of the class $FaxMachine$ uses the pattern $D^*cD^*gD^*gD^* \notin L(\hat{G}_c)$, which leads to $\lightning$ of the automaton, thus it violates Rule 1. On the contrary, the derivations of $Scanner$ and $Printer$ are accepted by $L(\hat{G}_c)$, thus satisfy Rule 1.

Now consider Rule 2. Let c, pr, pe stand for $2k(Class)$, $2k(Property)$, $2k(PackagedElement)$, respectively. Note that the occurrence of more than 30 pr after a c violates Rule 2. In fact, all the sequences of productions in the pattern "$...c...(pr...)^n, n > 30$" are not allowed by the rule (there is no c between any two pr), indicating that the class has more than 30 attributes.

To satisfy Rule 2, we propose the following controlling grammar $\hat{G}_p$ to restrict the use of the language:

$$\hat{G}_p \begin{cases} S \rightarrow pe\ \ S \mid c\ \ Q_c \mid D\ S \mid D \\ Q_c \rightarrow pe\ S \mid c\ \ Q_c \mid pr\ \ Q_1 \mid D\ Q_c \mid D \\ Q_i \rightarrow pe\ S \mid c\ \ Q_c \mid pr\ \ Q_{i+1} \mid D\ Q_i \mid D\ \ (1 \leq i < 30) \\ Q_{30} \rightarrow pe\ \ S \mid c\ \ Q_c \mid D\ Q_{30} \mid D \\ D \rightarrow \{p \mid p \in P\ \wedge\ p \notin \{c, pr, pe\}\} \end{cases} \tag{2}$$

where S, Q_c, Q_i are nonterminals, D includes all productions except c, pr, pe. $L(\hat{G}_p)$ accepts the sequences of productions satisfying Rule 2.

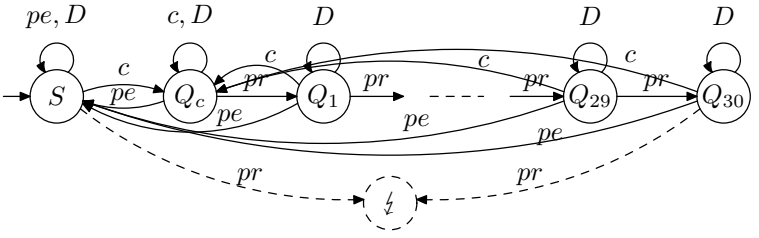


Fig. 4. The Automaton $\hat{A}_p$

Implicitly, the controlling grammar specifies an automaton $\hat{A}_p$ in Fig. 4. Strings of the pattern "$D^*cD^*(pr\ D^*)^n, n > 30$" will lead $\hat{A}_p$ to the error state.

If the sequence of productions applied to derive a model is accepted by the language $L(\hat{G}_p)$, then the model conforms to Rule 2. In Fig. 1, the derivations of the classes *Scanner* and *Printer* use the pattern $D^*cD^*prD^* \in L(\hat{G}_p)$, thus satisfy Rule 2.

Thanks to the controlling grammars, when a model violates required rules, the controlling language will reject the model (an implicit error state ↯ will be activated). Some error handling method may be called to process the error, e.g., printing an error message indicating the position and the cause.

We can also use controlling grammar to handle a consistency rule concerning activity diagrams.

Example 3. In an activity diagram, the number of outgoing edges of $ForkNode$ should be the same as the number of incoming edges of its pairwise $JoinNode$.

Let n, f, j, i, o stand for $2k(node)$, $2k(ForkNode)$, $2k(JoinNode)$, $2k(incoming)$, $2k(outgoing)$, respectively. We propose the following controlling grammar $\hat{G}_a$ to restrict the use of the language to satisfy the rule:

$$\hat{G}_a \begin{cases} S \to N\ \ F\ \ I^*\ \ Q\ \ O^*\ \ N \mid N\ \ I^*\ \ O^*\ \ N \mid D^* \\ Q \to O\ \ Q\ \ I \mid N\ \ S\ \ N\ \ J \\ N \to n\ \ D^* \\ F \to f\ \ D^* \\ J \to j\ \ D^* \\ I \to i\ \ D^* \\ O \to o\ \ D^* \\ D \to \{p \mid p \in P\ \wedge\ p \notin \{n, f, j, i, o\}\} \end{cases} \tag{3}$$

$L(\hat{G}_a)$ accepts all the sequences of productions of the pattern $NFI^*O^nNSNJI^nO^*N$, which leads to models respecting the rule. This context-free grammar implicitly specifies a PDA (Pushdown Automaton [17]), which is more complex than the automata in Figures 3 and 4.

Globally, any UML user model M derived from the C-System $C = G \overrightarrow{\cdot} \hat{G}_a \in \mathcal{C}^{CF}_{CF}$, i.e. $M \in L(C)$, conforms to the rule in Example 3.

As a more concrete instance, we consider the model in Fig. 2. The XMI-compliant document of the model in Fig. 2 is the follows:

```
<packagedElement xmi:type="uml:Activity"
             xmi:id="U937506ed-af64-44c6-9b4c-e735bb6d8cc6"
             name="Activity1" visibility="public">
<node xmi:type="uml:InitialNode" xmi:id="U16aa15e8-0e5d-
       4fd1-930a-725073ece9f0">
   <outgoing xmi:idref="Ue9366b93-a45b-43f1-a201-2038b0bd0b30"/>
</node>
<node xmi:type="uml:ForkNode" xmi:id="U26768518-a40c-
                  4713-b35e-c267cc660508" name="ForkNode">
   <incoming xmi:idref="Ue9366b93-a45b-43f1-a201-2038b0bd0b30"/>
   <outgoing xmi:idref="Ua800ba9b-e167-4a7c-a9a9-80e6a77edeb7"/>
</node>
<node xmi:type="uml:DecisionNode" xmi:id="Uc9e4f0de-8da6-
       4c98-9b95-b4cde30ccfc0" name="DecisionNode">
   <incoming xmi:idref="Ua800ba9b-e167-4a7c-a9a9-80e6a77edeb7"/>
   <outgoing xmi:idref="Ua4a2b313-13d6-4d69-9617-4803560731ef"/>
   <outgoing xmi:idref="U6eede33f-98ac-4654-bb17-dbe6aa7e46be"/>
</node>
<node xmi:type="uml:JoinNode" xmi:id="Ud304ce3c-ebe4-
                  4b06-b75a-fa2321f8a151" name="JoinNode">
   <incoming xmi:idref="Ua4a2b313-13d6-4d69-9617-4803560731ef"/>
   <incoming xmi:idref="U6eede33f-98ac-4654-bb17-dbe6aa7e46be"/>
</node>
<edge xmi:type="uml:ControlFlow"
       xmi:id="Ua4a2b313-13d6-4d69-9617-4803560731ef"
       source="Uc9e4f0de-8da6-4c98-9b95-b4cde30ccfc0"
       target="Ud304ce3c-ebe4-4b06-b75a-fa2321f8a151">
   <guard xmi:type="uml:LiteralString"
       xmi:id="U6872f3b3-680c-430e-bdb3-21c0a317d290"
       visibility="public" value="x>10"/>
</edge>
<edge xmi:type="uml:ControlFlow"
       xmi:id="U6eede33f-98ac-4654-bb17-dbe6aa7e46be"
       source="Uc9e4f0de-8da6-4c98-9b95-b4cde30ccfc0"
       target="Ud304ce3c-ebe4-4b06-b75a-fa2321f8a151">
   <guard xmi:type="uml:LiteralString"
       xmi:id="Ub853080d-481c-46ff-9f7c-92a31ac24349"
       visibility="public" value="else"/>
</edge>
<edge xmi:type="uml:ControlFlow"
       xmi:id="Ua800ba9b-e167-4a7c-a9a9-80e6a77edeb7"
       source="U26768518-a40c-4713-b35e-c267cc660508"
       target="Uc9e4f0de-8da6-4c98-9b95-b4cde30ccfc0"/>
```

```
<edge
       xmi:type="uml:ControlFlow"
       xmi:id="Ue9366b93-a45b-43f1-a201-2038b0bd0b30"
       source="U16aa15e8-0e5d-4fd1-930a-725073ece9f0"
       target="U26768518-a40c-4713-b35e-c267cc660508"/>
</packagedElement>
```

It is easy to detect that the sequence of applied productions "$...nD^*fD^*iD^*oD^*nD^*$ ... $nD^*jD^*iD^*i...$" is not accepted by $L(\hat{G}_a)$ (one o follows f, while two i follow j), thus there is an inconsistency.

We remark here that there are two preconditions of using the controlling grammar concerning the sequences of the model elements in the XML document: 1. *ForkNode* must appear before its pairwise *JoinNode*; 2. *incoming* edges must appear before *outcoming* edges in a node. The two conditions are trivial, since it is easy to control their positions in the exporting XMI documents in implementing such a transformation.

5 Related Work

The most popular technique for verifying software correctness is *model checking* [23]. In this framework, we have three steps in verifying a system. First, we formalize system behavior as a model (e.g., a transition system, a Kripke model [24]). Second, we specify the properties that we aim at validating using temporal logics. Third, we use a certain checking algorithm to search for a counterexample which is an execution trace violating the specified properties. If the algorithm finds such a counterexample, we have to correct the original design.

Most checking tools use specific semantics of UML diagrams. They have the flavor of model checking, e.g., Egyed's UML/Analyzer [25] [26] and OCL (Object Constraint Language) [27]. At first, developers design UML diagrams as a model. Then, we specify the consistency rules as OCL or similar expressions. Certain algorithms are executed to detect counterexamples that violate the rules [28]. Note that these techniques do not discriminate the rules on the model level and those concerning the language level features.

Unlike these techniques, our framework takes another way of ensuring correctness. It consists of the following steps:

1. Specifying the grammar G of a language. It specifies an *operational semantics*, which defines what a language is able to model. Developing the grammar is mainly performed by *language designers*.
2. Modeling correctness rules of the use of languages as a controlling grammar $\hat{G}$. It specifies a *correctness semantics*, which defines what a language is authorized to derive. This process is the duty of *safety engineers* whose responsibility is to assure the correct use of the language.

3. The two grammars constitute a consistent language as a whole, that is, any derivations of the global system language is a correct and consistent use of the language.

In particular, our work differs from model checking in the following aspects:

1. Our work has different objectives, and uses different approaches to those of model checking. As we show in Fig. 5, model checking techniques use a **bottom-up approach** — they verify execution traces T^* at the lower level L_1 to prove the correct use of the grammar G at the middle level L_2. Whereas our proposal uses a **top-down approach** — we model correctness rules as acceptable sequences of productions (P^*) at the higher level L_3 to ensure the correct use of G. Then any derivatives (at L_1) that conform to the C-System $C = G \overrightarrow{\cdot} \hat{G}$ are definitely a correct use. So the two techniques are complementary.

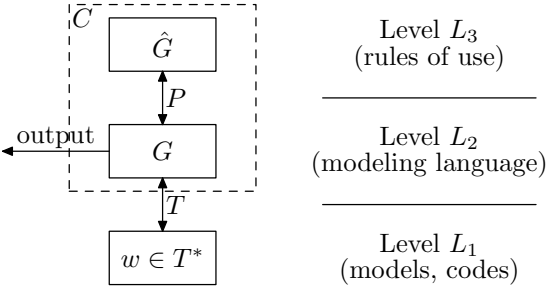


Fig. 5. Three Levels of the Framework

2. Our work and model checking express language-level and model-level constraints, respectively. Language-level constraints are more effective, because they implicitly have reusability. That is, we only need to develop one language-level constraint and apply it to all the models in the language. However, using model checking, we need to replicate model-level constraints for each model. Additionally, model checking can process model-specific constraints.

3. Our work and model checking use syntax-based and semantics-based approaches (or static and dynamic analysis), respectively. As a result, our approach is generic and metamodel-independent, and concerns little about semantics. So it can be applied to all MOF-compliant languages, not only to UML. However, model checking techniques depends on the semantics of a language, thus specific algorithms should be developed for different models.

4. Our work and model checking catch errors at compile-time and runtime, respectively. As a result, our approach implements membership checking of context-free languages, which is decidable. That is, it searches in a limited space, which is defined by grammars. Model checking may search in a larger, even infinite space, so we have to limit the space of computing, and introduce the risk of missing solutions.

6 Discussion

In this section, we would like to shortly discuss some issues which are beyond the scope of this paper.

The first issue concerns the implementation of controlling grammars. The controlled and controlling grammars can be implemented using two parsers separately. The technique for constructing a parser from a context-free grammar is rather mature [29] [30]. Some tools provide automated generation of parsers from a grammar specification, such as Yacc, Bison.

Notice that the inputs of controlling parsers are the sequences of productions applied in the parsing of $L(G)$. So there are communications between the two parsers. Once module G uses a production p_i, then the name of the production is sent to $\hat{G}$ as an input. If $L(\hat{G})$ accepts the sequence of productions and $L(G)$ accepts the model, then $L(G \overrightarrow{\cdot} \hat{G})$ accepts the model.

The second issue deals with multiple rules. If we have multiple guidelines or consistency rules, each rule is formalized using a grammar. We can develop an automated tool that converts the grammars into automata, and then combine these automata to compute an intersection, i.e., an automaton A' [17]. The intersection A' can be used as a controlling automaton, which specifies a controlling language $L(A')$ that includes all the semantics of the rules.

The third issue is about the tradeoff between cost and benefits of applying the proposed approach. It seems that writing a controlling grammar is expensive, because it involves formal methods. However, it is probably not the case. As we mentioned, a controlling grammar specify language-level constraints, and can be reused by all the models derived from the controlled grammar. Thus the controlling grammar can be identified and formalized by the organizations who define the language or its authorized usage, e.g., OMG and FAA (Federal Aviation Administration), respectively. Developers and software companies can use the published standard controlling grammar for checking inconsistencies in their models. By contraries, if every user writes their own checking algorithms and codes, e.g., in OCL or other programming languages, the codes will be hard to be reused by other users who have different models to check. Thus the total cost of all the users may be higher. Of course, more empirical results on the tradeoff is a good direction for future work.

7 Conclusion

We provided a language-theoretic view on guidelines and consistency rules of UML. We proposed the formalism of C-Systems, short for "formal language control systems". To the best of our knowledge, none related work proposed similar methodologies. Rules are considered as controlling grammars which control the use of modeling languages. This methodology is generic, syntax-based and metamodel-independent. It provides a top-down approach that checks and reports violations of language level constraints at compile-time. It can be also applied to all MOF-compliant languages, not only to UML, since it does not depend on the specific semantics of languages.

Since we focused on the methodological foundation, one of the future work is to develop an automated checking tool implementing the presented principles. We will also examine instant checking techniques of our method. One feature of UML/Analyzer is instant checking, which only verifies the small portion where the model changes, in order to save the cost of checking [31]. Intuitively, our approach is also easy to be extended to instant checking. We only need to generate the XMI document of the changed part of diagrams (e.g. a class in a class diagram), and verify it. However, this calls for more works in detail.

References

1. OMG: Unified Modeling Language: Infrastructure, version 2.1.1 (07-02-06). Object Management Group (2007)
2. OMG: Unified Modeling Language: Superstructure, version 2.1.1 (07-02-05). Object Management Group (2007)
3. Medvidovic, N., Rosenblum, D.S., Redmiles, D.F., Robbins, J.E.: Modeling software architectures in the unified modeling language. ACM Trans. Softw. Eng. Methodol. 11(1), 2–57 (2002)
4. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide, 2nd edn. Addison-Wesley, Reading (2005)
5. Balzer, R.: Tolerating inconsistency. In: Proceedings of the 13th International Conference on Software Engineering (ICSE 1991), pp. 158–165. IEEE Computer Society, Los Alamitos (1991)
6. Easterbrook, S.M., Chechik, M.: 2nd international workshop on living with inconsistency. In: Proceedings of the 23rd International Conference on Software Engineering (ICSE 2001), pp. 749–750. IEEE Computer Society, Los Alamitos (2001)
7. Nuseibeh, B., Easterbrook, S.M., Russo, A.: Leveraging inconsistency in software development. IEEE Computer 33(4), 24–29 (2000)
8. Kuzniarz, L., Reggio, G., Sourrouille, J.L., Huzar, Z. (eds.): Workshop on consistency problems in UML-based software development I, co-located with UML 2002 (2002), http://www.ipd.bth.se/consistencyUML/
9. Kuzniarz, L., Huzar, Z., Reggio, G., Sourrouille, J.L. (eds.): Workshop on consistency problems in UML-based software development II, co-located with UML 2003 (2003), http://www.ipd.bth.se/consistencyUML/
10. Huzar, Z., Kuzniarz, L., Reggio, G., Sourrouille, J.L. (eds.): Workshop on consistency problems in UML-based software development III, co-located with UML 2004 (2004), http://www.ipd.bth.se/consistencyUML/
11. Vidal, J.-P.S., Malgouyres, H., Motet, G.: UML 2.0 Consistency Rules (2005), http://www.lattis.univ-toulouse.fr/UML/
12. Vidal, J.P.S., Malgouyres, H., Motet, G.: UML 2.0 consistency rules identification. In: Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP 2005). CSREA Press (2005)
13. Federal Aviation Administration: Handbook for Object-Oriented Technology in Aviation (OOTiA), vol. 2.1, Considerations and issues. U.S. Department of Transportation (October 2004)
14. Motet, G.: Risks of faults intrinsic to software languages: Trade-off between design performance and application safety. Safety Science (2009)
15. OMG: MOF 2.0 / XMI Mapping, version 2.1.1 (07-12-01). Object Management Group (2007)

16. ISO/IEC: ISO/IEC 14977:1996(E): Extended BNF (1996)
17. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading (1979)
18. Chen, Z., Motet, G.: Modeling system safety requirements using input/output constraint meta-automata. In: Proceedings of the 4th International Conference on Systems (ICONS 2009), pp. 228–233. IEEE Computer Society, Los Alamitos (2009)
19. Chen, Z., Motet, G.: System safety requirements as control structures. In: Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC 2009). IEEE Computer Society, Los Alamitos (2009)
20. Chen, Z., Motet, G.: Formalizing safety requirements using controlling automata. In: Proceedings of the Second International Conference on Dependability (DEPEND 2009). IEEE Computer Society, Los Alamitos (2009)
21. Ginsburg, S., Spanier, E.H.: Control sets on grammars. Mathematical Systems Theory 2(2), 159–177 (1968)
22. Dassow, J., Paun, G., Salomaa, A.: Grammars with controlled derivations. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, pp. 101–154. Springer, Heidelberg (1997)
23. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge (2000)
24. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and Reasoning about Systems, 2nd edn. Cambridge University Press, Cambridge (2004)
25. Egyed, A.: Fixing inconsistencies in UML design models. In: Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), pp. 292–301. IEEE Computer Society, Los Alamitos (2007)
26. Egyed, A.: UML/Analyzer: A tool for the instant consistency checking of UML models. In: Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), pp. 793–796. IEEE Computer Society, Los Alamitos (2007)
27. OMG: Object Constraint Language, version 2.0 (06-05-01). Object Management Group (2006)
28. Chiorean, D., Pasca, M., Cârcu, A., Botiza, C., Moldovan, S.: Ensuring UML models consistency using the OCL environment. Electr. Notes Theor. Comput. Sci. 102, 99–110 (2004)
29. Earley, J.: An efficient context-free parsing algorithm. Commun. ACM 13(2), 94–102 (1970)
30. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading (1986)
31. Egyed, A.: Instant consistency checking for the UML. In: Osterweil, L.J., Rombach, H.D., Soffa, M.L. (eds.) Proceedings of the 28th International Conference on Software Engineering (ICSE 2006), pp. 381–390. ACM, New York (2006)

A Domain Specific Language for Extracting Models in Software Modernization

Javier Luis Cánovas Izquierdo and Jesús García Molina

University of Murcia
{jlcanovas,jmolina}@um.es

Abstract. Model-Driven Engineering techniques can be used both to create new software and to modernize existing software systems. Model-Driven Software Modernization requires a first step for the extraction of models. Most modernization scenarios involve dealing with the GPL source code of the existing system. Techniques and tools providing efficient means to extract models from source code are therefore needed.

In this paper, we analyze the difficulties encountered when using the existing approaches and we propose a language, called Gra2MoL, which is especially tailored to address the problem of model extraction. This provides a powerful query language for concrete syntax trees, and mappings between source grammar elements and target metamodel elements are expressed by rules similar to those found in model transformation languages. Moreover, the approach also allows reusing existing grammars.

1 Introduction

Model-Driven Engineering (MDE) techniques are not only used to create new systems, but also to evolve or modernize legacy software. The field of model-based software modernization is currently emerging and a great research and development effort will thus be necessary in the years to come. The OMG has recently proposed several modernization standards in its ADM initiative [1], such as KDM [2]; certain tools, such as MoDisco [3], are currently under development, and some research challenges have even been identified for the evolution of systems built by using MDE techniques [4].

Most modernization scenarios [5], such as platform migration or application improvement, involve dealing with source code written in a General Purpose Language (GPL). Techniques and tools providing efficient means to extract models from source code are therefore essential. In this extraction process, models conforming to a target metamodel are generated from source code conforming to the grammar of a GPL. Thus, a bridge between the technical spaces grammarware [6] and MDE must be built. This bridge is normally implemented by dedicated parsers (i.e. tools performing both parsing and model generation tasks), since the use of approaches such as bridging *grammarware* and MDE or transforming programs have certain drawbacks which limit their usefulness, as will be explained later in this paper. Bridging approaches [7,8] aim to create textual

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, pp. 82–97, 2009.

domain specific languages (DSL) which have a simpler structure than GPLs; in the case of program transformations [9,10], the resulting program must still be converted into a model. Since the construction of dedicated parsers is a time-consuming task, we have defined a DSL, denominated as Gra2MoL, which has been specifically designed for extracting models from GPL source code.

Gra2MoL allows mappings to be established between grammar elements and target metamodel elements in a similar way to which model-to-model transformations are expressed in languages as ATL [11] and RubyTL [12]. It provides a powerful query language to ease the navigation and query of syntax trees. We have used Gra2MoL to extract models from Java and PL/SQL code, and this experience has shown a reduction in development time, while maintenance is also improved and existing grammars are reused. In this paper we present Gra2MoL, and compare it with the existing approaches.

This paper is organized as follows. Section 2 analyzes the difficulties encountered when using existing solutions for model extraction and introduces Gra2MoL. In Section 3, we describe the language for querying concrete syntax trees provided by Gra2MoL. Section 4 presents the main features of Gra2MoL and explains how it has been implemented, while Section 5 shows an example of the language. Finally, Section 6 presents our conclusions and some future work.

2 Model Extraction from Source Code

In this section we contrast different approaches which could be used to extract models from GPL code, indicating their main limitations. This discussion will motivate the approach proposed in this paper. We will begin by giving a definition of *model extraction* in the context of model-driven modernization, identifying the main issues to be addressed.

Figure 1 shows the elements involved in the process of extracting models from GPL source code. This process is a grammar-to-model transformation T which has as its input a program P along with the grammar definition G to which it conforms. It generates a target model M_T conforming to a target metamodel MM_T which defines the information to be extracted. The extraction process also requires specifying mappings between the grammar elements and the metamodel elements. As we will see, the form of these mappings is different depending on each considered approach. The input program is represented by either an Abstract Syntax Tree (AST) or a Concrete Syntax Tree (CST). Along this paper, we will use the term "syntax tree" to refer both AST and CST.

Therefore extracting models from source code is a scenario which requires a bridge between *grammarware* and MDE techniques (*modelware*), the same as the definition of a textual concrete syntax for an abstract syntax metamodel [7,13]. With GPL code, creating this bridge requires an efficient mechanism for traversing syntax trees since the model elements to be extracted are usually composed of information that is scattered in such trees. In particular, this scattering is mainly caused by the means used to represent the references between elements. Models are graphs and any model element can directly refer to another, whereas

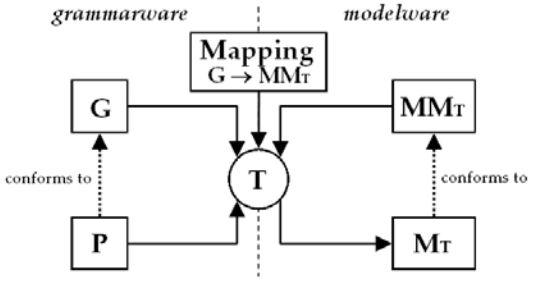


Fig. 1. Process of extracting models from source code

in a syntax tree that represents certain code which conforms to a GPL grammar, the references between grammar elements are implicitly established by means of identifiers. Transforming an identifier-based reference into an explicit reference involves looking for the "identified" node on the syntax tree. For instance, if a model element is extracted from a "function call" statement where one argument is a global variable, certain necessary information, such as the type of the variable or the function signature, is located outside the current scope ([14] calls this kind of transformations global-to-local transformations).

2.1 Approaches for Model Extraction

The chosen strategy is normally that of creating **dedicated parsers**. Given a grammar and a target metamodel, a dedicated parser provides a specific solution which performs both parsing and model generation tasks. The former is in charge of extracting a syntax tree from the source code and the latter traverses such syntax trees in order to generate the target model. For example, in [15] a dedicated parser is built to extract models from PL/SQL code, and another with which to extract models from VB code is presented in [16]. However, dedicated parser development is a time-consuming and very expensive task. The effort required is usually alleviated by automatically extracting an AST from the source code. This step is performed by using an API, which is intended to make the management of such tree easier. An example of such APIs is the JDT Eclipse project [17], which works with Java source code. But APIs do not currently exist for a number of the GPLs widely used in modernization (e.g. PL/SQL language). In addition, although these APIs tackle AST extraction and management, a mechanism for retrieving scattered information must still be hard-coded, so APIs do not considerably shorten the development time.

MoDisco (Model Discovery) [3] is a model extraction framework, which is part of the Eclipse GMT project [18]. This framework is currently under development and provides a model managing infrastructure through which to implement dedicated parsers ("discoverers" in MoDisco terminology). A KDM based metamodel, a metamodel extension mechanism and a methodology for designing such extensions are also planned.

Several approaches for bridging *grammarware* and *modelware* have been defined in the context of creating textual domain specific languages (DSL) for MDE. These approaches can be classified in two groups according to whether they are focused on grammars or metamodels. Grammar-based approaches are oriented towards automatically generating metamodels from grammars, whereas metamodel-based approaches work in the opposite direction. In model-driven modernization, the process starts from existing source code which conforms to the grammar of a GPL. Therefore metamodel-based approaches, such as TCS, are not well suited. As is stated in [13]: "*If the problem at hand is to develop a single, eventually general purpose language then the efforts for developing a dedicated parser are worthwhile*" (rather than using TCS).

Grammar-based approach limitations. XText [7] is an example of a grammar-based approach, which is part of the openArchitectureWare toolkit [19]. An EBNF-based language is provided to specify the input grammar to the xText processor, that is, a specification of the textual concrete syntax including grammar rules intended to guide the generation of the corresponding metamodel. Three artifacts are thus generated: i) a metamodel of the language, ii) a parser to recognize the language syntax and to create models conforming to the generated metamodel, and iii) a language specific editor.

However, several problems arise when xText is used in modernization, since dealing with GPL source code involves problems not addressed by this approach, which is aimed at simpler languages than a GPL. The automatically generated metamodel from the grammar of a GPL is of poor quality because it includes superfluous elements and grammatical aspects, and the semantic gap between this metamodel and the target metamodel is thus very high. A model-to-model transformation is therefore required to convert models generated by xText into models conforming to the desired target metamodel. However, since current model-to-model transformation languages do not offer an efficient mechanism to resolve the problem of gathering scattered information, the definition of such transformation is a complex task.

With regard to grammar reuse, neither existing grammar reuse, the reuse of grammars for well-known parser generators (e.g. ANTLR), nor the reuse of xText grammar specifications is promoted. On the one hand, translating a grammar specification provided by a parser generator into a xText EBNF-based specification is extremely complicated since some parser options which are needed to recognize GPLs cannot be specified (e.g. in Java, the use of backtracking or the inclusion of syntactic predicates). On the other hand, xText grammar specifications are oriented to a specific metamodel so they include specific rules for such a metamodel.

Wimmer et al. [8] and Kunert [20] have proposed improving the quality of the generated metamodel by applying heuristics and including manual annotations to the grammar. However, the quality of the metamodel generated from a GPL grammar is still low and it is necessary to additionally define a model-to-model transformation. Moreover, tools supporting these two approaches are not yet available.

Program transformation approach limitations. Program transformation languages, such as Stratego/XT [9] and TXL [10], could be used to extract models from source code by expressing the abstract syntax as a context-free grammar rather than a metamodel. However, when such languages are used, the following limitations are encountered. Firstly, the result of a program transformation execution is a program conforming to a grammar, and therefore a tool for bridging *grammarware* and *modelware* would be still needed to obtain the model conforming to the target metamodel. Secondly, grammar reuse is not promoted because each toolkit uses its own grammar definition language. Moreover, each toolkit only provides a limited number of GPL grammars (i.e. Java and C in Stratego and TXL).

2.2 Our Approach for Model Extraction

In the context of an Oracle Forms migration project, we faced model extraction from PL/SQL code. Then we considered the definition of a DSL in order to overcome the limitations of the previously discussed approaches. This DSL should decrease the development time, make the maintenance easier and promote the reuse of existing grammars (e.g. ANTLR and JavaCC grammars). To achieve these objectives, it is necessary to raise two key design issues: how can mappings between grammar elements and metamodel elements be expressed in a simple and readable way, and what notation is appropriate when retrieving scattered information from syntax trees.

Model-to-model transformation languages could be used to express the mappings between grammar elements and metamodel elements, but this possibility would require models rather than GPL code as input, and a dedicated parser would therefore have to be implemented to extract a model conforming to an intermediate metamodel (i.e. an abstract syntax tree metamodel) from source code. A model-to-model transformation would then be applicable. However, defining these transformations would lead to an important problem: the inadequacy of the query language.

Many current model transformation languages, such as ATL [11] or QVT [21], provide a variant of the OCL navigation language [22] which allows model graphs to be traversed. Although OCL-like expressions are appropriate for most practical model-to-model transformation definitions, they are not convenient for typical global-to-local transformations involved in a model extraction from GPL code: long navigation chains must be written using dot notation. Integrating a more suitable query language in an existing model transformation language would involve important changes if a language supporting two different query mechanisms were to be obtained. For instance, a plugin mechanism could be implemented.

We have therefore created a DSL which has been specially designed to express grammar-to-model mappings, and which provides a powerful query language for syntax trees. In particular, since the scattered information problem appears in both AST and CST, and obtaining a CST is easier than an AST, we use CSTs for representing the source code. This DSL, denominated as Gra2MoL (Grammar

Table 1. Comparison of Gra2MoL with the analyzed approaches. NA = Not applicable, G = Grammar, MM_T = Target metamodel, MM_I = Intermediate metamodel, T = Transformation definition, P = Dedicated parser, T_{PT} = Program transformation definition, G_{xt} = xText grammar, $m2m$ = model-to-model transformation definition, G_{AS} = Abstract syntax grammar.

Approach	Syntax tree navigation	Artifacts to be created	Post processing	Existing grammar reuse	Built grammar reuse	Purpose
Dedicated parser (+ API)	GPL code (+ primitives)	MM_T P	None	Yes	NA	Specific model extraction
Grammar based bridging (xText)	Poor support	G_{xt} MM_T $m2m$	M2M transformation: generated $MM_I \rightarrow MM_T$	No	No	DSL creation
Program transf.	Stratego incorporates a query language [23]	MM_T T_{PT} G_{AS} $m2m$	Extracting a model from a program conforming to G_{AS}	Limited (a few grammars)	Yes	Program transf.
Gra2MoL	*Structure-shy* query language	MM_T T	None	Yes	NA	General purpose model extraction

To Model Transformation Language) will be described in Section 4 and the query language is introduced in the following section.

Table 1 contrasts Gra2MoL with the analyzed approaches. The columns show the properties which are compared: the ability to navigate the syntax tree; which artifacts must be created; whether post-processing is necessary; whether it is possible to reuse existing and provided grammars; and the main purpose of the approach. The artifacts to be created and the post-processing tasks determine the effort level of each approach. For instance, we note that bridging and program transformation approaches require more complex tasks than Gra2MoL, such as writing model-to-model transformations or defining a GPL grammar, whereas in Gra2MoL it is only necessary to create the transformation definition and the target metamodel. With regard to the creation of a dedicated parser, Gra2MoL turns a hard-coding task into the writing of a grammar-to-model transformation definition using a language specially tailored for extracting models. As a consequence, development time is reduced by using Gra2MoL.

3 A Query Language for Concrete Syntax Trees

As stated above, grammar-to-model transformations involve an intensive use of queries to collect scattered information. Therefore, a model extraction approach must provide a powerful query language, which facilitates the access to tree nodes outside the current construct scope (i.e. a rule). Figure 2 illustrates the scattering problem for a simple example of extracting a model element from a PL/SQL procedure. The CST shown corresponds with a procedure declaration which includes a variable declaration and an assignment statement initializing

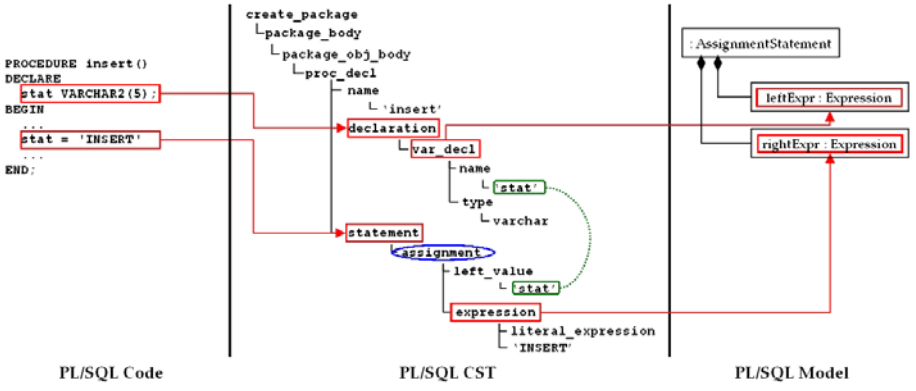


Fig. 2. Example of scattered information. The oval indicates the current scope and the dotted line indicates an identifier-based reference between tree elements.

the declared variable. The `AssignmentStatement` model element represents a PL/SQL assignment which has two attributes to register the right-hand side and left-hand side expressions of the assignment. As can be observed, whereas all the information needed to initialize the right-hand side attribute (`insert` literal) is inside the current scope (depicted as an oval), the information needed to initialize the left-hand side attribute is outside such a scope, because the `stat` variable declaration is referenced by an identifier. Therefore, a query is necessary to resolve this reference to the `stat` variable, by accessing the corresponding `declaration` node and retrieving the variables properties.

We have developed a *structure-shy* query language, inspired by XPath [24], which allows a CST of the source code to be navigated without the need to specify each navigation step. The term "structure-shy" is often used to refer to behavior specifications which are loosely bounded to the data structures on which operations (i.e. queries) are applied. The term "structure-shy" query is used in this sense.

In order to navigate the CST, the nodes are "typed" using the grammar definition, and each tree node registers the name of the grammar element as its type. Figure 3 illustrates the conformance relationships between the CST and the grammar definition, showing a CST for several PL/SQL procedures along with the corresponding fragment of PL/SQL grammar. The conformance rules are those commonly used to create a tree of this kind:

- A non-terminal element corresponds to a tree node. For instance, the `proc_decl` non-terminal element corresponds to the `proc_decl` tree node in Figure 3.
- A terminal element corresponds to a leaf. In Figure 3, the `Name` terminal corresponds to the `Name` leaf.
- A production rule is represented by a node hierarchy whose parent corresponds to the non-terminal element on the left-hand side of the rule, and a child for each grammar element on the right-hand side by applying the

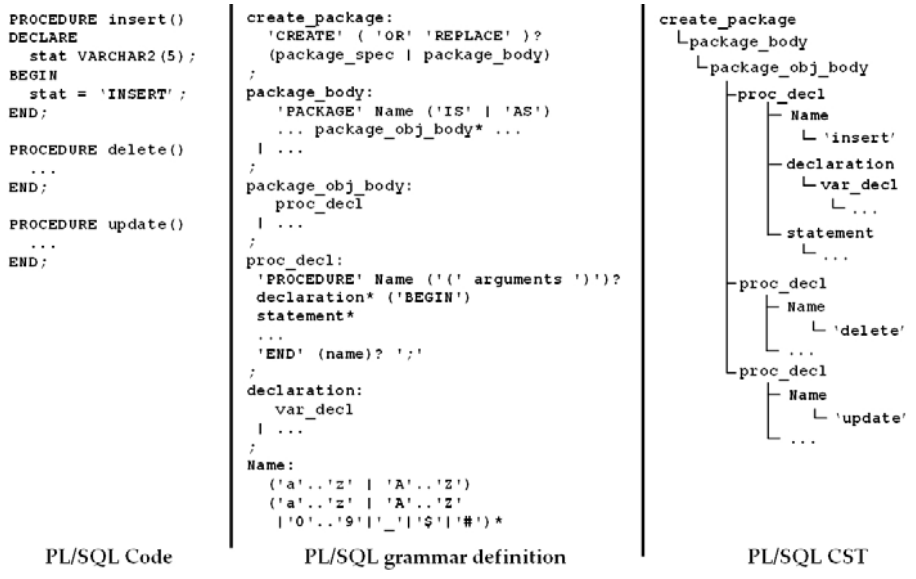


Fig. 3. CST for the PL/SQL grammar

previous rules. In Figure 3, the `proc_decl` production rule is represented by the hierarchy whose root is a `proc_decl` tree node.

A query consists of a sequence of query operations in which each operation includes an operator, a node type and optional filter and access expressions. The EBNF expression for a query operation is:

```
{('/'|'//'|'///') ('#')? nodeType [filterExpression] [accessExpression]}
```

We have defined three operators for querying and navigating over CSTs: `/`, `//` and `///`. The `/` operator returns the immediate children of a node and is similar to dot-notation (e.g. in OCL). The `//` and `///` operators permit the traversal of all the nodes children (direct and indirect), thus retrieving all nodes of a given type. The `///` operator differs slightly from the `//` operator. Whereas the `///` operator searches the syntax tree in a recursive manner, the `//` operator only matches the nodes whose depth is less than or equal to the depth of the first matched node. The `///` operator is, therefore, only used to extract information from recursive grammar structures. These two operators allow us to ignore intermediate superfluous nodes, thus making the query definition easier, since it specifies what kind of node must be matched, but not how to reach it, in a structure-shy manner. The `createFunctionCallStatement` rule defined in Section 5 will show the difference between both operators.

Since a query could return one or more subtrees, the `#` operator is used to indicate the root node from which the information needed can be accessed. This operator must be associated to one and only one query operation of the sequence

```
Locals(cp:create_package) : Sequence(variable_declaration)
post result=if cp.spec.oclIsKindOf(package_body) then
  p.spec.package_obj_body->
    select(pob | pob.oclIsKindOf(procedure_decl))->
    collect(fb | fb.declaration)->flatten()->
    select(ds | ds.oclIsKindOf(variable_declaration))
  else
    Sequence {}
  endif
```

Fig. 4. OCL query for extracting all the variable declarations of every prodecure of the PL/SQL CST shown in Figure 3

of operations forming a query. For instance, in order to extract all the PL/SQL variable declarations defined in every procedure of the PL/SQL CST shown in Figure 3, the following query could be expressed `/create_package//#var_decl`. The same query expressed in OCL is shown in Figure 4. It is worth mentioning how the clarity, legibility and conciseness are improved.

Query operations can also include a filter expression, which is enclosed in curly brackets. A filter expression is a logical expression which is applied to the leaves of the node specified in a query operation. Each operand of a filter expression is a boolean function which checks the properties of a leaf, such as its value or whether it exists. Only those nodes that satisfy the filter expression will be selected. For example, the query `/create_package//#proc_decl{Name.exists && Name.eq('insert')}` will select every `procedure` grammar element with a `Name` leaf and the value of such leaf must be `insert` in the PL/SQL CST shown in Figure 3.

Finally, query operators can also include an access expression enclosed in square brackets, which is used to access to sibling nodes through indexing. For instance, the query `/create_package//#proc_decl[0]` will select the first `procedure` grammar element of the CST in Figure 3, which is the `insert` procedure.

The following section outlines the Gra2MoL domain specific language which integrates the described query language.

4 Gra2MoL

In Gra2MoL, a model extraction process is considered as a grammar-to-model transformation, so mappings between grammar elements and metamodel elements are explicitly specified. According to Figure 1, the input of a Gra2MoL transformation is source code along with the grammar definition it conforms to, a target metamodel and a transformation definition; the output is a model which conforms to the target metamodel.

The language has been designed as a rule-based model transformation language with rules whose structure is similar to those provided in languages such as ATL or RubyTL, with two important differences: i) the source element of a

rule is a grammar element rather than a metamodel element, and ii) the navigation is expressed by the query language described in Section 3, rather than an OCL-based language.

A Gra2MoL transformation definition consists of a set of transformation rules. Each rule specifies the mappings between a grammar element and a target metamodel element and is composed of four parts:

- The *from* part specifies a grammar non-terminal symbol, and declares a variable that will be bound to a tree node when the rule is applied. This variable can be used by any expression within the rule. The *from* part can also include query operations to check the structure to be satisfied by the nodes whose type is the non-terminal symbol.
- The *to* part specifies the target element metaclass.
- The *queries* part contains a set of query expressions which allow information to be retrieved from the CST. The result of these queries will be used in the assignments of the *mappings* part.
- Finally, the *mappings* part contains a set of bindings to assign a value to the properties of the target element.

An example of Gra2MoL is shown and commented upon in Section 5.

4.1 Bindings and Rule Evaluation

A binding construct is used in the *mappings* part to establish the relationship between a source grammar element and a target metamodel element. This construct has very similar syntax and semantics to the binding construct of the RubyTL [12] and ATL [11] languages. A binding is written as an assignment using the operator "=". The left-hand side must be a property of the target element metaclass. The right-hand side can be the variable specified in the *from* part of the rule, a literal value or a query identifier.

The execution of a transformation definition is driven by the bindings. The definitions of rule conformance and well-formed transformation stated for RubyTL in [12] are applicable to Gra2MoL, with simple changes. A metaclass A_m conforms to a metaclass B_m if they are the same or A_m is subtype of B_m, whereas a node type A_n conforms to a node type B_n if they are the same. Every Gra2MoL transformation definition must have an entry point in order to start the transformation execution. The entry point is the first rule of the transformation definition and its mappings are in charge of starting the transformation execution. When a rule is applied on a node, the filter located in the from part is first checked and then, if the node satisfies the filter, an instance of the target metaclass is created, and the rules bindings are executed. In the execution of a binding, three situations may arise according to the nature of the right-hand side.

1. If it is a literal value, the value is directly assigned to the property of the left-hand side.
2. If it is a query identifier, the query is executed and a rule resolving this binding is looked up in the transformation definition, i.e. a rule whose types

of the *from* and *to* parts conform to the types of the right-hand side and left-hand side of the binding, respectively. Whenever a conforming rule is found, it is applied by using the element of the right-hand side of the binding as the source grammar element.

3. If it is an expression, it is evaluated and two situations may arise, depending on whether the result is a node whose type corresponds to a terminal (a leaf) or a non-terminal symbol. If it is a leaf, the result is a primitive type and is directly assigned, otherwise, a rule to resolve the binding is looked up and executed, as was explained in the previous case.

4.2 Implementation

The first step in the execution of a Gra2MoL transformation is to build the CST of the source code. Current implementation of Gra2MoL uses ANTLR grammar definitions. These definitions can be enriched with actions in order to create the CST. However, we are interested in using ANTLR grammar definitions without attached actions for two reasons: (1) to alleviate the grammar developer from the burden of creating the CST programmatically and (2) to promote grammar reuse. We have therefore defined an enrichment process which automatically adds the actions needed to build the CST to the grammar rules.

Gra2MoL uses a metamodel internally to generically represent CSTs of the parsed source code. This metamodel is shown in Figure 5. There are three kinds of elements in a CST model, namely `Leaf`, `Node` and `Tree`. `Leaf` represents a tree node which corresponds to a recognized terminal symbol. `Node` represents a tree node which corresponds to a recognized non-terminal symbol and is composed of one or more children nodes, either of the `Leaf` or `Node` type. The `type` attribute identifies the grammar symbol whose recognition has yielded the tree node creation (this is needed to navigate through the CST, as was explained in Section 3). Finally, `Tree` represents the root node of the tree. The creation of models conforming to this metamodel is driven by the conformance rules explained in Section 3.

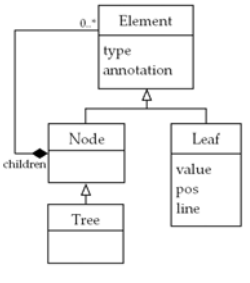


Fig. 5. CST metamodel (MM_{CST})

The execution process of a Gra2MoL transformation is shown in Figure 6(b), together with a schema of the pre-processing step T to enrich the ANTLR grammar in Figure 6(a). Note that 6(b) is the same as Figure 1, except that a parser

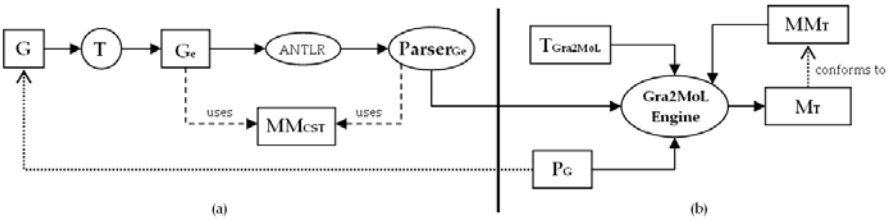


Fig. 6. Gra2MoL implementation

is an input to the Gra2MoL engine to build the CST model. This parser is generated from the grammar (G_e) enriched with actions intended to create CST models conforming to the metamodel MM_{CST} shown in Figure 5.

5 Example

Oracle Forms is an Oracle technology used to design and build enterprise applications in which the business logic and database access is encapsulated in PL/SQL triggers. The source code of such applications is organized in sequences of statements in which each sequence corresponds to a trigger. Gra2MoL has been used within the context of a project to migrate Oracle Forms applications to Java platform, in order to extract models from PL/SQL code. We implemented a Gra2MoL transformation definition to extract models conforming to a metamodel representing a subset of the PL/SQL abstract syntax. This transformation definition consists of 57 rules[1]. An excerpt of this transformation definition will be shown as follows. Figure 7 shows the parts of the PL/SQL grammar and the PL/SQL metamodel considered in this example.

```
rule createPLSQLDefinition
  from create_package cp
  to PLSQLDefinition
  queries
    seqt : /cp//#seq_of_statements;
  mappings
    triggers = seqt;
end_rule

rule createTriggerBlock
  from seq_of_statements seqt
  to TriggerBlock
  queries
    stats : /seqt/#statement;
  mappings
    statements = stats;
```

[1] The complete transformation definition can be downloaded from http://modelum.es/gra2mol

```
seq_of_statements
  :  statement SEMI ( statement SEMI )*
;
statement
  : return_statement
  | function_call
  | ...
;
function_call
  : user_defined_function LPAREN
    ( call_parameter )? RPAREN )?
;
user_defined_function
  : sql_identifier
;
call_parameter
  : (parameter_name '=>' )? nested_expression
    (call_parameter)?
;
```

```
parameter_name
  : identifier
;
return_statement
  : keyRETURN ( plsql_expression )?
;
sql_identifier
  : identifier
;
identifier
    :   ID ( DOT ID )*
;
ID /*options { testLiterals=true; }*/
  : 'A'..'Z'
    ('A'..'Z'|'0'..'9'|'_'|'$'|'#')*
  | DOUBLEQUOTED_STRING
;
```

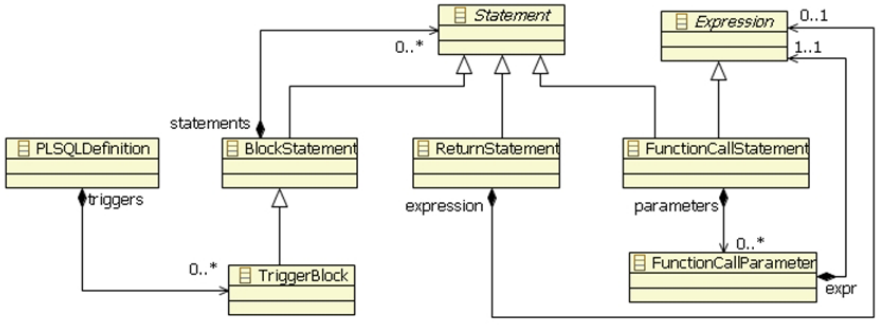


Fig. 7. Excerpt of the PL/SQL grammar and the subset of PL/SQL metamodel used in the example. The non-terminal symbols used in the example are underlined.

```
end_rule

rule createReturnStatement
  from statement/return_statement st
  to ReturnStatement
  mappings
end_rule

rule createFunctionCallStatement
  from statement/function_call st
  to FunctionCallStatement
  queries
    fc     : /statement/#function_call;
    iden   : /fc/user_defined_function//#identifier;
    params : /fc///#call_parameter;
  mappings
    name = iden.ID;
    parameters = params;
end_rule
```

```
rule createFunctionCallParamForFunctionCall
  from call_parameter cp
  to FunctionCallParameter
  queries
    iden  : /cp/parameter_name/#identifier;
  mappings
    name = iden.ID;
end_rule
```

The first rule starts the transformation process by creating an instance of PLSQL Definition. This rule has only one binding whose right-hand side is a query identifier and whose left-hand side refers to the triggers attribute of the `PLSQLDefinition` metaclass. The query is therefore executed and the rules conforming the binding are then looked up and executed. In this example, the `createTriggerBlock` rule would create instances of `TriggerBlock` and would apply the `statements = stats` binding, whose right-hand side is a query identifier and whose left-hand side refers to the statements attribute of the `TriggerBlock` metaclass. In this case, both `createReturnStatement` and `createFunctionCallStatement` rules conform to the binding, but the filter of these rules allows the selection of only one, depending on the direct children of the statement grammar element. The `createFunctionCallStatement` rule illustrates the meaning of the `//` and `///` operators. On the one hand, the `//` operator used in the second query avoids the need to specify every navigation step (i.e. `sql_identifier`) to reach the identifier node. On the other hand, since the `call_parameter` production rule is defined recursively, the `///` allows the CST to be traversed in order to retrieve every `call_parameter` node.

It is worth mentioning how clear and legible the transformation shown above is. Both the implicit rule application driven by the bindings and the format of the queries make it a clean language. Moreover, the query format also contributes towards improving the legibility of the CST retrievals. Figure 8 shows the result of an execution of this transformation definition.

Fig. 8. Result of a Gra2MoL transformation execution

6 Conclusions and Future Work

In this paper, we have presented Gra2MoL, a DSL for extracting models from GPL source code by means of grammar-to-model transformations. Gra2MoL has therefore been designed as a rule-based language inspired by languages such as ATL and RubyTL. A Gra2MoL transformation definition consists of rules which

transform grammar elements into model elements by manipulating the CST of the source code. A powerful language has been defined to navigate and query a CST in a structure-shy manner. Several benefits will be derived from this new approach, in comparison to existing solutions.

With regard to the implementation of a dedicated parser, our approach considerably reduces the development time, and maintainability is also favored. Mappings and queries are not hard-coded in the code of a programming language, but are specified in a clear, concise and legible manner.

We have also compared Gra2MoL to *grammarware*-MDE bridging and program transformation approaches. Since neither of these approaches was devised for the extraction of models from GPL code, both require the performance of difficult tasks. On the one hand, bridging approaches were designed to create DSL, and are therefore not very practical for dealing with GPL. For instance, xText generates low-level models, and model-to-model transformations have to be defined. On the other hand, a program transformation approach could be used but the developer is required to write a target grammar specification and to define a bridge to convert the program generated into a model; Gra2MoL transformations are, moreover, simpler than those transformations expressed by the more commonly used transformational approaches, which use formalisms such as rewriting techniques.

With regard to future work, we are working on several issues such as a modularity mechanism for Gra2MoL transformations and the identification of query patterns to make the query definition independent from the grammar structure. We are also analyzing how to integrate Gra2MoL in the Modisco framework as a mechanism through which to create discoverers. Moreover, since Gra2MoL deals solely with ANTLR grammars, we would like to support other parser generators in order to increase the number of existing grammars that can be reused.

Acknowledgment

This work has been supported by Consejería de Educación y Cultura (CARM, Spain), grant TIC-INF 06/01-0001 and Fundación Séneca (Murcia, Spain), grant 08797/PI/08. Javier Luis Cánovas Izquierdo enjoys a doctoral grant from the Fundación Séneca.

References

1. Architecture-Driven Modernization Roadmap. OMG (2006)
2. ADM Task Force: Knowledge discovery meta-model (kdm). OMG (2007)
3. MoDisco, http://www.eclipse.org/gmt/modisco/
4. van Deursen, A., Visser, E., Warmer, J.: Model-driven software evolution: A research agenda. In: Workshop on Model-Driven Software Evolution (2007)
5. ADM Task Force: Architecture-driven modernization scenarios. OMG (2006)
6. Klint, P., Lämmel, R., Verhoef, C.: Toward an engineering discipline for grammarware. ACM Transactions on Software Engineering Methodology 14(3), 331–380 (2005)

7. Efftinge, S.: openarchitectureware 4.1 xtext language reference (2006), http://www.eclipse.org/gmt/oaw/doc/4.1/r80xtextReference.pdf
8. Wimmer, M., Kramler, G.: Bridging grammarware and modelware. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 159–168. Springer, Heidelberg (2006)
9. Stratego/XT, http://strategoxt.org/
10. TXL, http://www.txl.ca/
11. Jouault, F., Kurtev, I.: Transforming models with atl (2005)
12. Cuadrado, J.S., Molina, J.G., Tortosa, M.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)
13. Jouault, F., Bézivin, J., Kurtev, I.: TCS: a dsl for the specification of textual concrete syntaxes in model engineering. In: GPCE, pp. 249–254 (2006)
14. van Wijngaarden, J., Visser, E.: Program transformation mechanics. a classification of mechanisms for program transformation with a survey of existing transformation systems, Department of Information and Computing Sciences, Utrecht University, Tech. Rep. UU-CS-2003-048 (2003)
15. Andrade, L.F., Gouveia, J., Antunes, M., El-Ramly, M., Koutsoukos, G.: Forms2Net - Migrating Oracle Forms to Microsoft .NET. In: GTTSE, pp. 261–277 (2006)
16. Migrating Visual Basic Applications to VB.NET using the NewCode extension for Microsoft Visual Studio. Newcode (2008)
17. JDT Eclipse project, http://www.eclipse.org/jdt
18. GMT Eclipse project, http://www.eclipse.org/gmt
19. OpenArchitectureWare toolkit, http://www.openarchitectureware.org
20. Kunert, A.: Semi-automatic generation of metamodels and models from grammars and programs. In: Fifth Intl. Workshop on Graph Transformation and Visual Modeling Techniques, E. N. in Theorical Computer Science, vol. 211, pp. 111–119 (2008)
21. Linda Heaton. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. OMG (2005)
22. OCL constraint language. OMG (2006), http://www.omg.org/cgi-bin/apps/doc?formal/06-05-01.pdf
23. van Wijngaarden, J.: Code Generation from a Domain Specific Language, M.Sc Thesis (2003)
24. Xpath, http://www.w3.org/TR/xpath

Challenges in Combining SysML and MARTE for Model-Based Design of Embedded Systems

Huascar Espinoza[1], Daniela Cancila[1], Bran Selic[2], and Sébastien Gérard[1]

[1] CEA LIST, Model-Driven Engineering Labs (LISE)
Point Courrier 94, 91191, Gif sur Yvette, France
{huascar.espinoza,daniela.cancila,sebastien.gerard}@cea.fr
[2] Malina Software Corporation
10 Blueridge Court, Nepean, Ontario, Canada
selic@acm.org

Abstract. Using model-based approaches for designing embedded systems helps abstract away unnecessary details in a manner that increases the potential for easy validation and verification, and facilitates reuse and evolution. A common practice is to use UML as the base language, possibly specialized by the so-called profiles. Despite the ever increasing number of profiles being built in many domains, there is still insufficient focus on discussing the issue of combining multiple profiles. Indeed, a single profile may not be adequate to cover all aspects required in the multidisciplinary domain of embedded systems. In this paper, we assess possible strategies for combining the SysML and MARTE profiles in a common modelling framework, while avoiding specification conflicts. We show that, despite some semantic and syntactical overlapping, the two are highly complementary for specifying embedded systems at different abstraction levels. We conclude, however, that a convergence agenda is highly desirable to align some key language features.

Keywords: model-based engineering, embedded systems, SysML, MARTE.

1 Introduction

The design of embedded systems is a complex process that depends more and more on the effective interplay of multiple disciplines, such as mechanical, electronics, and software engineering. In particular, the lack of a common design language between different disciplines hampers reasoning about system properties. The architecture of a system is particularly vulnerable to bad design choices made in the early design phases, which, unfortunately, often tend to show up later during the integration or construction phases. Designers of one part of the system may make wrong assumptions concerning some other parts resulting in increasing development costs due to long feedback cycles.

The use of models throughout the design process is gaining momentum in addressing these issues [20]. Models allow designers from different disciplines to share knowledge, facilitate design comprehension, and assess system-level trade-offs seeking higher quality and reliability. We subscribe to the view that both system design

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, pp. 98–113, 2009.

and integration will be reduced significantly by the use of a common modelling formalism, even for smaller projects. In particular, we believe that the widespread acceptance of UML (Unified Modelling Language) [16] by industry and the use of UML profiles for domain-specific expressiveness ease the challenge considerably. A profile is the mechanism standardized by the OMG for creating domain-specific modelling languages by refining the concepts of an existing standard language such as UML.

A number of UML profiles have been proposed for modelling embedded systems, both within a standardization context and as research outcomes [12]. In our work, standardization is a crucial concern since it promotes lower overall training costs and helps to reduce the risk of being dependent on a single tool vendor. We particularly focus on two standard UML profiles that cover, as a whole, a broad cross-section of the modelling capabilities required for the embedded system domain. On the one hand, SysML (Systems Modelling Language) [17] provides constructs to specify traceable requirements, structure and behaviour of system blocks, as well as a parametric formalism to specify equation-based analytical models. On the other hand, MARTE (Modelling and Analysis Real-Time and Embedded systems) [18] deals with time- and resource-constrained aspects, and includes a detailed taxonomy of hardware and software patterns along with their non-functional attributes to enable state-of-the-art quantitative analyses (e.g., performance and power consumption).

A major impediment to any kind of real-world application is that a single profile may not be sufficient to capture all aspects in the multidisciplinary domain of embedded systems. A number of industrial and research efforts have started to consider the use of both profiles in a synergistic manner, as described in more detail in Section 5, to cover as much as possible the description of embedded systems at different abstraction levels (e.g., [9] [14] [11] [1]). However, even in the standards world, different profiles may be mutually inconsistent and may overlap in ways that are not fully documented. Hence, it is essential to investigate ways of combining these two UML profiles to avoid conflicts and mismatches.

In this paper, we provide the basis for a comparison between the two profiles. The purpose is to identify some typical scenarios in which their combined usage is of relevant added value in the embedded systems domain and, to provide a convenient starting point for those interested in using both profiles in a complementary manner. One problem is that, because they are constructed for different purposes and follow different design rationales, they tend to define different syntaxes for the same modelling concepts. This issue immediately puts profile users in a dilemma when they try to exploit both profiles in the same system model. Some minimum alignment is necessary to deal with such overlaps. Consequently, another objective of this paper is precisely to encourage the SysML and MARTE standardization task forces to provide a convergence and alignment program for their respective technologies.

The remainder of the paper is organized as follows. Section 2 outlines SysML and MARTE and their respective modelling capabilities. Section 3 introduces some anticipated scenarios that combine concepts from both expressiveness domains. In Section 4, we provide some strategies to properly compare and integrate common modelling constructs. Section 5 discusses contributions and shortcomings of other attempts at combining both profiles. A short discussion and conclusions round out the paper.

2 Background

2.1 UML Profiling Capabilities

Because of the diverse nature of the disciplines needed for designing real-time and embedded system, it is clear that a single modelling language is not adequate to cover all the various concerns involved. Consequently, there has been much discussion about the suitability of UML for such domains compared to custom domain-specific modelling language designed from scratch [9]. The latter approach has the obvious advantage of enabling the definition of a language that is optimally suited to the problem at hand. At first glance, this may seem the ideal approach to modelling language definition, but closer examination reveals that it can have serious drawbacks. If each sub-domain is expressed using a specific language, there is the problem of integrating the various parts of the design so that the full system can be verified, or simply unambiguously understood. Another drawback of domain-specific languages is the availability of and support for industrial-strength tools and training for a such custom language (commercial tool vendors are rarely interested in supporting custom or low-volume languages). This can lead to significant and recurring expenses related to developing and supporting custom tools and providing training for them.

In contrast, although UML was designed to eliminate the accidental complexity stemming from gratuitous diversity, it still provides a built-in mechanism, *profiles*, for creating domain-specific modelling languages (DSML). Profiles are based on specializing the general UML concepts and semantics and can, therefore, take advantage of existing UML tools and expertise. For example, the domain-specific concept of a real-time clock can be derived from the more generic UML concept of an object, by the addition of new attributes and additional semantic constraints. It is even possible to use a domain-specific notation in lieu of the standard UML notation. This kind of reuse can mitigate or even eliminate some of the above drawbacks of custom DSMLs

Another advantage of the profile approach is that a profile can be defined as an *annotation profile*, meaning that it can be overlaid, *non-intrusively*, on an existing model to provide supplemental information and semantics not present in the original model. Such profiles can be applied to create domain-specific views and specializations of an underlying model. This is especially useful in multidisciplinary problems for the ability to capture cross-domain concerns. For example, a UML model can be annotated with information such that it can be analyzed for its schedulability characteristics by domain experts or specialized tools. At the same time (and independently of the schedulability view) a reliability engineer might overlay a reliability-specific view on that same model to determine its overall reliability characteristics. Some parts of the MARTE profile are significant examples of this profile usage.

2.2 SysML and MARTE Modelling Capabilities

SysML and MARTE consider characteristics of the embedded systems domain at different abstraction levels, architectural styles, and particularly for specific purposes or application areas. In this section, we summarize their major modelling capabilities.

SysML is a UML profile "for specifying, analyzing, designing, and verifying complex systems that may include hardware, software, information, personnel, procedures, and facilities" [17]. The so-called *Block* concept is the common conceptual entity that factorizes many different kinds of system elements such as electronic or software components, mechanical parts, information units, and whatever structural entity composing the system under interest. Blocks articulate a set of modelling perspectives enabling separation of concerns during systems design. Eschewing excessive detail[1], we identify the following key contributions of SysML regarding UML:

- *Architecture Organization:* These include modelling concepts to organize system architecture descriptions as defined by the IEEE 1471 standard [6]. Among them, the concepts of *view*, *viewpoint*, and *rationale* are most important.
- *Blocks and Flows:* Block description and internal block diagrams of SysML enable the specification of more generic interactions and phenomena than those existing just in software systems. This includes physical flows such as liquids, energy, or electrical flows. The dimension and measurement units of the flowing physical quantities can be explicitly defined.
- *Behaviour:* Although most behaviour constructs in SysML are similar to UML (interactions, state machines, activities, and use cases), SysML refines some of them for modelling continuous systems and probabilities in activity diagrams.
- *Requirements:* SysML provides an explicit facility for modelling system requirements, along with their traceability with regard to the architecture evolution. These can be specified in either graphical or tabular format.
- *Parametrics:* A perspective called parametric diagram allows SysML users to describe, in a graphical manner, analytical relationships and constraints, such as those described by mathematical equations. Parametric diagrams provide a mechanism for integrating SysML design models with engineering analysis.

MARTE is a UML profile that supports specification of real-time and embedded systems [18]. In addition to functional design, this profile adds constructs to describe the hardware and software (e.g., OS services) resources and defines specific properties to enable designers to perform timing and power consumption analysis. With regard to UML, MARTE adds the following features[2]:

- *NFPs.* The NFPs (Non-Functional Properties) modelling framework provides means to specify semantically well-formed non-functional properties (e.g., throughputs, bandwidths, delays, memory usage), supported by a language to formulate algebraic and time expressions.
- *Time.* A highly refined model of time and timing mechanisms integrates concepts from different sub-domains in embedded systems design, such as causal time, synchronous time, and chronometric time.
- *Software application.* A common model of computation provides semantic support for the real-time object paradigm. This paradigm allows specifying

[1] Further information on SysML can be found via http://www.omgsysml.org/
[2] Further information on MARTE can be found via http://www.omgmarte.org/

applications at a high abstraction level, by delegating concurrency, communication, and time-constraint aspects to a modular unit called *real-time unit* (RtUnit).

- *Components.* The MARTE component model extends UML composite structures and SysML internal block diagrams with a notion of message-based communications. This is intended to support the request-reply/publish-consume communication paradigm.
- *HW/SW Resources.* Software and hardware resources can be described at different levels of abstraction, including their typical services, as found in common OS platforms, and common non-functional properties like power consumption or memory usage.
- *Quantitative Analysis.* A set of pre-defined non-functional annotations enable MARTE models to bridge with state-of-the-art performance and scheduling analysis tools.

3 Scenarios of Combined Usage

To focus our study, we identify a set of representative scenarios in which a combined usage of SysML and MARTE is of relevant added value in the embedded systems domain. Although this set is certainly incomplete, it allows us to drive our comparison in a more focused manner. The intent is to adequately answer the question of what can each profile target best in modelling, and then determine their integration issues.

3.1 Defining Architecture Frameworks

The modelling capabilities of both SysML and MARTE are rich enough for a wide range of design approaches. This has the flexibility for supporting and integrating multiple design perspectives, but also the difficulty of understanding and choosing among a variety of language alternatives. In both cases, there is not a predetermined approach to use the language constructs through the development lifecycle. This means that a consistent modelling framework and methodology should be defined for using these profiles in a particular application domain.

Architecture Organization. In the IEEE 1471 standard (and in the draft of its upcoming update ISO/IEC 42010), the concept of modelling framework is referred as *architecture framework*. An architecture framework "establishes a common practice for creating, organizing, interpreting and analyzing architectural descriptions used within a particular domain of application or stakeholder community" [6]. An architecture framework identifies one or more predefined architectural viewpoints. Viewpoints define how to construct views, which are in turn a representation of a system from the perspective of a set of modeling concerns.

SysML implements IEEE 1471 by providing a set of constructs to organize models. In particular, SysML does not define any specific viewpoint, but it provides means to specify how views are built, and to relate any user-specific view to a given viewpoint. This is aligned with the IEEE 1471 approach that envisages libraries of viewpoints, in order to enable architects selecting those useful for system design at hand.

Although MARTE does not provide any concrete model element to define viewpoints, it has an implicit conception of viewpoints rooted in its design rationale. Indeed, some of the MARTE constructs have been designed to define domain-specific viewpoints (see Section 2.1). Such viewpoints, when applied to a standard UML model, cast that model in a domain-specific way and may also add supplementary information to the model relevant to the viewpoint.

In consequence, there is no language overlapping in this respect. SysML and MARTE can be used in complementary way. While SysML provides means to create viewpoints in a general way, MARTE provides particular viewpoints. However, an open issue is to enable designers of architecture frameworks to build consistent interview rules that ensure meaningful and correct-by-construction models.

3.2 Requirements Engineering

System Usage Scenarios. Requirements engineering is the process by which the requirements for systems and software products are gathered, analyzed, documented, and managed throughout the development life cycle. UML has traditionally been used to document user requirements by means of use case diagrams. Use cases follow a graphical, scenario-based approach. This means that requirements are organized into system usage histories, acting as a user-friendly bridge between technical and business stakeholders.

Although use cases may be formalized to certain degree, for example by using sequence diagrams in order to detail such usage histories, they are often criticized for a number of limitations. For instance, use cases lack well-defined semantics, which may lead to differences in interpretations by stakeholders [22]. They are applied mainly to model functional requirements, but are not very helpful to model non-functional ones. Also, relationships between requirements and the various architectural parts that satisfy those requirements are difficult to trace. SysML and MARTE provide some significant enhancements in these aspects.

Requirements Management/Traceability. SysML requirements diagrams explicitly show the various kinds of relationships between different requirements. This enlarges the spectrum of requirements engineering tools that can interact with UML tools. In effect, the SysML requirements modelling constructs are intended to provide an automated bridge between architectural models and traditional requirements management tools such as, for instance, Requisite Pro, Rectify, or DOORS [1]. The latter provide support for traceability analysis, flow-down, derivation, assignment, among other requirement engineering activities. In particular, requirements tracing is very useful, for example, to identify how requirements are affected by changes, and to prioritize requirements. Traceability also provides a possibility of verifying whether or not all requirements have been fulfilled by the system and sub-system components.

Non-Functional Requirements. On its side, MARTE offers key features to specify non-functional requirements in general and timing requirements in particular. In embedded systems development, non-functional characteristics (e.g., performance, reliability, power consumption) influence a wide range of design decisions [3]. One possible scenario is using MARTE annotations to characterize non-functional constraints in use case diagrams and their underlying sequence diagrams. This provides two important capabilities leading toward more formal requirements specification.

First, non-functional requirements are cohesively specified along with functional requirements. While specifying non-functional aspects is possible with SysML requirements diagrams, their semantic relationship to concrete functional system usages is hard to capture. In particular, the completeness of requirements satisfaction in real-time systems is strongly dependent on the coupling between system function and timing. In MARTE, timing annotations provide semantic definitions closely related to the system behavior. For instance, one may define a jitter constraint in the arrival of an event and identify if such event relates either to a send, receive, or consume occurrence within a sequence diagram. Second, non-functional annotations follow a well-defined textual syntax, which is supported by the MARTE's Value Specification Language (VSL). The main advantages of this level of formalization are the ability to support automated validation, verification, traceability, and, more simply, an unambiguous understanding by stakeholders.

Clearly, SysML and MARTE concepts, articulated by use cases and scenarios, are highly complementary. While scenarios are useful for managing change and evolution, managing scenario traceability across multiple changes becomes increasingly difficult. SysML contributes with constructs to define such traceability relations. Additionally, MARTE completes scenario precision with well-formed non-functional annotations. However, it is important to define clear consistency rules to combine them in a typical development process using different requirements engineering tools.

3.3 System-Level Design Integration

In a typical development process for embedded systems, software and other forms of engineering will be at least partially concurrent. The system is developed by composing pieces that, all or in part, have already been pre-designed or designed independently by different teams specialized in different disciplines. This is often done in vertical design chains such as, for example, in the avionics and automotive industries. Therefore, there is a need for supporting design artefacts by common and standard specification formalisms that will allow plug-and-play of subsystems and their implementation [23].

A model view is a typical abstraction that helps to divide a complex problem into smaller and comprehensible parts. In order to integrate global models, e.g., for performing system-level analysis, we must recombine these smaller parts in a consistent way. UML supports model composition by means of composite structure diagrams. The basic principle is to define usages of model elements in a given context. The idea of composite system models is to describe how information from multiple modelling artefacts and views is to be joined, deployed, or configured. Although there is a linguistic divergence[3], both SysML and MARTE reuse this notion with some particularities. Thus, some aspects need to be taken into consideration for their combined use.

Hierarchy and Composition. To understand the pragmatic problems of SysML-MARTE joint usage, let us consider the scenario of a large development project with engineers from multiple disciplines. It should be carefully decided how the system model will be created by integrating the models from different disciplines. One important issue is the layering and mismatched sub-system hierarchies, which has been

[3] While SysML uses the term "block" for such composition units, MARTE uses "structured component".

comprehensively addressed by Maier [15]. For instance, in multiprocessor software-intensive design, the electronic system perspective typically represents a hierarchy of interconnected processors, each containing software units. From the software perspective, the hierarchy is reversed, as generally illustrated in MARTE examples [18]. At the top is a distributed application, composed of software units that interact through data- and message-based interfaces. Below the application are the operating system (OS) and library layers that support the distributed application. At the bottom of the hierarchy, the hardware (processors and networks) completes the model.

This aspect is important when deciding which kind of modelling constructs will be used to represent hierarchy, allocation/deployment, and composition. For instance, while a composition relationship would be used for the hardware viewpoint, an allocation relationship (supported by both SysML and MARTE) would be preferred by the software designers. Some compatibility or merging rules need to be defined to provide system-level consistency.

While in some cases, engineers from a given discipline would exclusively use either SysML or MARTE, in other cases they would need to combine concepts from both profiles. An integration scenario may consist in starting from a system-level model, probably specified with SysML blocks, and in adding later some additional semantics to some of these blocks, for example by applying MARTE stereotypes. Indeed, the detail level underlying MARTE constructs makes possible to specify some aspects such as concurrency and synchronization mechanisms, as well as resource patterns such as processing resources, communication buses, or power supply devices along with a set of predefined quality attributes. This is especially required in application areas where designers are interested in preparing models to perform simulation, quantitative analysis or product synthesis.

Interfacing/Interaction. A central concern in system and software architecting is to understand the interfaces and interactions between structural elements. The nature of such interfaces and interactions can significantly vary from software to other kind of systems. Looking at the structural aspects, we can see that MARTE adopted the notions of *port* and *flow* from SysML. This may seem very convenient from a perspective of semantic consistency. However, SysML flow ports require careful attention when used to model flows of physical quantities, such as for example energy or torque. Cares must be exercised in defining explicit behaviour on flow transmission. SysML physical flows are often continuous in time, whereas MARTE flows are used to describe data transmission with particular delegation semantics. While providing a precise semantics to flows is currently outside the scope of both profiles, their combined use should define a common "semantic envelope" that could be shared by SysML and MARTE. In this way, composing models from different disciplines will preserve system-level consistency.

3.4 Engineering/Quantitative Analysis

Engineering analysis (SysML term) or quantitative analysis (MARTE term) concern the use of mathematical techniques to study certain quality attributes of the system. They include stress, thermal or fluid analysis in mechanical engineering, and performance or reliability analysis in software engineering. One challenging problem in model-based engineering is to integrate models that are commonly used for system

production or software code generation with the information that is relevant to perform analysis [7]. The goal is to reduce the time required to prepare a design model for performing analysis and to ensure greater accuracy of an analysis model by directly associating it with the actual system model. Both SysML and MARTE provide key contributions in this direction, but some alignment work has to still be done.

Timing Modelling. Beyond the annotation of quality attributes, timing analysis requires a careful semantic definition closely related to the system behavior and the different models of computation and communication [2]. SysML does not extend the UML time model, but a set of preliminary requirements were established by its standardization board, including continuous time models and relativistic effects that can occur in distributed systems. In MARTE, time modelling is a core concern. We can distinguish at least three layers of time constructs:

- In a first layer, time is presented as a set of fundamental notions such as time instant, duration, time bases, or clocks. These provide an unambiguous basis to express further modeling constructs and well-formed value spaces for data types.
- In a second layer, MARTE provides mechanisms to annotate timing requirements and constraints in UML models. One key modeling feature is the concept of *observation*. Observations provide marking points in UML models to specify assertions. Some typical assertions have been embedded in ready-to-use patterns, such as for example jitters.
- In the third layer, time concepts are defined as part of the behavior, not mere annotations. This set of constructs cover both physical and logical time. While the logical time is the basis to understand basic temporal notions, this is further refined to support precedence/dependency in presence of concurrency, and clocked time abstractions to cover synchronous language abstractions (such as those from Lustre, Signal or Esterel).

While the adoption of the two basic layers is certainly useful for system engineering in general, the third layer would need some extensions to include, for example, modelling of the continuous dynamics of systems [12]. This would need to provide means to specify system behaviour in terms of hybrid discrete event and differential algebraic equation systems.

Quantities Values. In SysML, a value property represents a quantifiable characteristic of a block (e.g. energy consumption, surface, and temperature range of a microprocessor). Value properties are defined in block compartments by assigning a name and a value type. A value type is a kind of data type that carries a particular pair consisting of a dimension and a measurement unit.

For its part, MARTE uses its Non-Functional Properties (NFPs) modelling framework. The NFPs modelling framework provides the ability to encapsulate rich annotations within non-functional values. For instance, consider a property named ``latency". Instead of specifying its meaning in an axiomatic way such as: "duration in milliseconds with an accuracy of 0.01 measured by simulation as a mean value", the specification itself include all this information in a normalized syntax. For this purpose, the MARTE data type system includes the required data structure (value, unit, precision, measurement source, etc.) in a predefined library. For example, Duration, Frequency

and Power are typical non-functional data types. Different units of the same physical quantity may be transformed to, or expressed in terms of, existing base units through a given conversion factor and an offset factor.

One of the main issues when trying to combine both profiles is that the modelling approaches to declare and specify quantitative values is quite different. The main difference is that SysML hard coded the qualification of value types with the stereotypes unit and dimension, while MARTE allows for declaring a set of qualifiers as an extendable library. As a consequence, using both modelling mechanisms in the same model may lead to inconsistencies and cumbersome model processing. Alignment of these two modelling styles is a key issue that is being dealt as a joint effort between the MARTE and SysML task forces at OMG.

Beyond syntactical issues, the debate should be centred on providing practical capabilities to both profiles. We believe that at least two key capabilities should be allowed from SysML and MARTE models:

- *Measurement conversion*. Quantities need to be expressed in different measurement units while still allowing tools to convert quantities from one set of units to another.
- *Dimensional analysis*. Physical expressions must guarantee the consistency of equations and solve resulting measurement units and dimensions.

If we look at SysML, it forces tools to be hard-coded with the transformations between measurement units (e.g., from "mm" to "m") because unit definition lacks conversion factors. Furthermore, dimensional analysis is not possible in SysML since dimensions are not defined in terms of basic dimensions and their exponents (e.g., $F = LMT^{-2}$). Conversely, while MARTE supports unit conversion, the notion of dimension has not been considered at all.

Parameters/Expressions. Parameterized expressions are a primary feature in order to prepare models for analyzing performance, risk, costs, and so on [7]. SysML parametric diagrams capture constraints among performance, physical, and other quality-related properties of the system and its environment. Such constraints are specified as equations among value properties. Equations can be specified in a third-party language (e.g., MathML or Modelica). The basic composite modelling entity is the Constraint Block. The relationships between modelling entities within a constraint block are not committed to an 'input' or 'output' role early. Thus, they are called non-causal, as opposed to data flow and control flow approaches. Non-causal models are suitable to enable analytic processing, and can increase the level of integration/automation between design tools and analysis tools.

In addition, MARTE's VSL gives the syntax to formulate algebraic and time expressions. VSL is rooted in OCL. However, VSL was intended to provide more compact expressions. In addition, VSL extends arithmetic and logical expressions with time-related annotations, which can be extended by libraries providing new functions.

We believe that a combined use of SysML parametric diagrams and VSL would provide significant advantages. While parametric diagrams provide a user-friendly formalism to specify non-causal models, VSL provides the textual syntax for constraint expressions. One open issue in VSL is its extension to support special expressions used in system engineering. For instance, differential and integrals, continuous time expressions, and discrete event equations.

4 Combination Strategies

In this section, we outline some issues in combining SysML and MARTE and propose general strategies to integrate both profiles in a single modelling framework. Table 1 summarizes the modelling aspects discussed in Section 3 along with a set of profile combination cases and implementation issues, which are elaborated below.

Table 1. MARTE/SysML Combination Issues

Modelling Concern (from Section 3)	SysML concepts (examples)	MARTE concepts (examples)	Conflicting Combination Cases*	Implementation Issues*
Architecture Organization	view, viewpoint, rationale,…	-	-	predefine library of MARTE viewpoints
Hierarchy/Composition	block, part, allocation	structural component, parts, hw/sw patterns, application-platform allocations	(a)	(1) & (2)
Interfacing/Interaction	port, flow, items	idem SysML + message-based	(a) (c)	(2)
Spectrum of Behavioral Models	rate, continuous, discrete edges, probability,…,	synchronous/asynchronous, causal/real-time	(c), (d)	(1)
System Usage Scenarios	use case, sequence diagrams	use cases, sequence diagrams	common UML concepts	-
Requirements Processing/Trace	requirement, trace relationships, test case	-	-	(1) SysML requirements can be fully imported
Non-Functional Requirements	requirement	nfp constraint, VSL expressions	complementary	(1)
Time Modelling	(UML) time constraints	extended time constraints, clocks, predefined nfp's for time analysis	-	(1) & (2) not all MARTE time notions required
Quantity Values	value property, value type, unit, dimension	nfp, nfp type, unit	(c) (d) overlapping	(2) language alignment required
Parameters/Expressions	constraint blocks, parametric diagrams	VSL expressions	(c) complementary	(1) VSL as expression language

* see text for full explanations

Combination Case. We can generalize typical categories of the combined usage of UML profiles (this is applicable for two or more profiles) as follows:

a) Each language is used for different partitions of the system, in which case they are practically mutually exclusive and conflicts are small or even negligible. For example, SysML is used for mechanical design and MARTE for software design. As shown in Table 1, this category needs special attention when defining the hierarchy/composition and interfacing/interaction constructs during a system-level integration phase.
b) Each language is used for a different level of abstraction. Again, there is not much conflict here. For instance, SysML is used for system domain analysis and MARTE for a detailed design.
c) The languages are used in combination into the same parts of a model (e.g., in the same modelling view) and for the same purpose or concern. For instance, we may use the SysML facilities for continuous behaviour in activity diagrams and the MARTE time annotations to support performance analysis.

d) The languages are used in combination but for different purposes such as, for example, using MARTE annotations to do performance analysis on a SysML model. The UML profiling capabilities of being able to apply many stereotypes to a single model entity is crucial for this kind of usage. There may be some conflict in trying to keep the consistency between MARTE non-functional annotations embedded in stereotype attributes (e.g., performance analysis stereotypes), with other SysML specifications such as block quantity value annotations or block constraint parameters.

Implementation Issues. The above combined cases may result in different combination issues from a tool implementation viewpoint. A supporting toolset that accompanies UML profiles is, strictly speaking, not a part of the language problem. However, the utility of a profile combination is directly related to the maturity of the supporting tools. We identify the following scenarios in combining MARTE and SysML profiles in modelling tools:

1) The simplest solution is to apply the profiles (i.e., the full profile definition) or sub-profiles (i.e., sub-packages stereotyped as profiles) where needed within a model. For example, a SysML user could specify that it requires the full Time Modeling package of MARTE. UML tools can manage this case because of the modularity defined in MARTE (organized in "extension units") and the UML's ability to select only those profile packages that are of direct interest.
2) While one may likely use some concepts of a profile or sub-profile, designers may not want to include the full profile or sub-profile package in their models. For instance, MARTE profile users may want to gain access to SysML concepts of block, but they may prefer to use the MARTE constructs for flows. UML does not allow for applying single stereotypes (contained in a profile) into a model. What is needed is a decoupling/merging mechanism to compose profile concepts and to make it available for profile users. Managing semantic compatibility is a requirement here.

In general, a hypothetical MARTE-SysML modelling tool should allow for filtering appropriate information according to specific users. Some engineering disciplines may be satisfied with a high-level description (e.g., blocks-and-flows description), software developers may want detailed behaviour specifications, while analysis experts may require information of non-functional properties. This aspect is more relevant when more than one stereotype is applied to a single UML model element. For instance, one may consider a SysML Block, as a specific hardware resource by annotating it with the appropriate MARTE stereotype. However, it is considered as a "resource" from a software viewpoint, but not from an electronic viewpoint. This needs a suitable presentation mechanism to show the right stereotype to various stakeholders.

4.2 Combination Clues

Defining a modelling framework that combines SysML and MARTE requires a systematic comparison of the two. We consider that at least the following aspects should be assessed in such work:

1. *Conceptual Domain Coverage*. Beyond syntactical aspects, it is important to begin by assessing both profiles from a conceptual viewpoint. The intent is to

reach an overall understanding of these profiles and determine what application domains are best covered by each. A good starting point is using the conceptual domain models underlying the UML profiles. Conceptual domain models are created as free as possible from considerations related to specific solution technologies so as to not embody any premature decisions that may hamper later language use. Currently, the MARTE specification provides a conceptual domain model in the form of a metamodel with a textual description. On the other hand, SysML directly defined UML stereotypes extending the UML metamodel. Although a conceptual description is provided, a metamodel would significantly help on identifying/comparing conceptualization entities of the targeted domain.

2. *Semantic/Syntactic Overlapping*. The evaluation of related points between both profiles should be clearly identified by defining overlapping semantics (conceptual coverage), abstract syntax (extended UML constructs), and concrete syntax (symbols and terminology). The intent is to determine which aspects of both profiles can be consistently aligned and/or selected to consistently use both profiles. Overlapping aspects must be assessed in the light of one of the language use cases, (c) or (d), identified in Section 3.1. While case (d) needs revisiting the notion of views and viewpoints in the context of UML profiles (see Section 2.1), case (c) requires a more careful treatment of semantic consistency.
3. *Usability/Pragmatics*. Usability issues are concerned with such concepts as ease of use, productivity, and user satisfaction. Once the overlapping concepts are identified and before deciding which profile features to adopt in a given modeling framework, we should identify the effectiveness of different symbols or stereotype names for model understandability, as well as the number of steps needed to accomplish a modeling goal. Of course that may depend on a tools' maturity. However, syntactical design choices can help avoid complicated ways of performing modeling steps or features which invite mistakes.
4. *Expressiveness Limitations*. One fundamental requirement that should drive a useful comparison is completeness and lack of model expressiveness. The evaluation of missing aspects needs to be objective by clearly identifying whether it implies a conceptual, semantic, or attributes insufficiency. This raises the problems of improving and extending both profiles, which is an important goal of our research.
5. *Abstraction/Refinement Levels*. One fundamental difference between SysML and MARTE relates to their ontological considerations. For example, while SysML does not consider any "functional" classification of structural elements (only the generic concept of Block exists), MARTE goes deeper by providing a detailed taxonomy of application and resource structural elements. Using abstract or concrete language concepts will depend on the phase of development, and the kind of model processing (simulation, verification, etc.) required at each level.

5 Related Work

The academic and industrial communities have recently begun to investigate the complementary use of SysML and MARTE to support model-based development of embedded systems.

Among current projects in the embedded systems domain, MeMVaTEx [1] defines a model-based methodology for modelling, validating, and tracing system requirements. It relies on SysML for requirements modelling and on MARTE for modelling timing aspects. Since these aspects are practically independent, their combination is handled methodologically, by providing consistent rules on when and where to apply concepts of the individual profiles. Another project combining these two profiles is INTERESTED [11], which attempts to create an interoperable tool-chain for enhanced rapid design and prototyping of embedded systems. This work aims at a more extensive use of SysML and MARTE. While the first profile serves to describe the high-level architecture organized around functional blocks, the second one provides the standard annotations to enable timing analysis. However, the methodological rules to guide the combined use of both profiles have yet to be established.

Two additional projects were recently started with the objective of adopting SysML and MARTE in the hardware/software co-design field. One of these, the SATURN project [19], proposes to bridge the gap between SysML/MARTE modelling and tools for architecture exploration, simulation and synthesis (in SystemC/VHDL for hardware and C/C++ for embedded software). The main strategy is to adopt most of the constructs of SysML and to integrate MARTE for adding the formal semantics of different models of computation and thus enable system verification. The second project, Lambda [14], intends to reconcile a number of related standards, including SysML, MARTE, AADL, and IP-XACT, to develop a library of broadly used software and hardware platforms.

At the other end of the spectrum, there is very little research literature discussing integrated approaches for system and software modelling based on UML. An example is [10], where the authors evaluate how UML and SysML could be consistently used for both system and software modelling. Perhaps, the main contribution of this work is a mapping between SysML and UML concepts and the identification of the application domains associated with each concept. Unlike this work, we attempt to provide a more rigorous comparison of system and software modelling concerns, and additionally, enrich expressiveness with MARTE features.

With regard to the combination of profiles at tooling level, the authors in [4] introduce a packaging unit called MDATC (which stands for Model-Driven Architecture Tool Component) that serves to collect metamodels and/or profiles, know-how, and required resources in order to support domain-specific activities. Thus, by using MDATC, modelling rules and constraints in the use of multiple profiles can be represented and exchanged in a standard format.

We end this section by highlighting the general problem of composition of languages or profiles. For instance, an aspect oriented approach supporting metamodel composition is proposed in [8]. The authors focus on implementing composition mechanisms for matching and merging model elements that crosscut the dominant structure described in a primary model. The composition directives are implemented in Kermeta, an open-source metamodelling language. Even if language composition between different metamodels is certainly a more difficult problem than combining stereotypes extending the same metamodel, especial care must be exercised. Our study can be inserted in this lively context and viewed as a modest contribution in composition of profiles, with special focus on SysML and MARTE, although in general fragmentation problem is left as an open problem.

6 Conclusions

Because of the varying nature of the disciplines involved in embedded system design, it is clear that a single modelling language, such as for example UML, may not be suitable for all aspects. We believe that the UML profile mechanism is well suited to create domain-specific languages, by providing a common semantic and syntactic foundation while also permitting reuse of the underlying modelling tools. Currently, there are an important number of profiles that may make their usage cumbersome, as they are often created mutually inconsistent and overlapping. In this paper we presented some integration strategies for combining the SysML and MARTE profiles. Both provide essential ingredients to model embedded systems. Our intent is to offer a better understanding of their conceptual domains, and to help in using both profiles in a single model by avoiding semantic and syntactical mismatches.

We presented some typical scenarios in which their combined usage is of relevant added value in the embedded systems domain. In general, using modelling constructs from one or the other profile depends on the expressive power a constructs should provide to practitioners. In a simple usage scenario, the intent may be to aid understanding and to communicate about a system design. As such, it is not necessary to define a detailed description or precise semantics, and basic evaluations of the architecture could be performed. In a more elaborated scenario, however, we may be interested on using powerful analysis tools, simulators, model checkers, product synthesis tools, and the like. In this case, the necessary levels of specification detail and semantic precision are much higher. While both forms of specification have merit, their usage will be driven by the specific needs of a particular development process and its phases through the system lifecycle.

Our future work consists in providing a detailed comparison of SysML and MARTE's semantic and syntax, providing pertinent examples on their combined usage, and suggesting some improvements regarding language mismatches.

Acknowledgments. The work presented here is partially carried out within the System@tic competitiveness cluster projects Lambda and IMOFIS.

References

[1] Albinet, A., Begoc, S., Boulanger, J.-L., Casse, O., Dal, I., Dubois, H., Lakhal, F., Louar, D., Peraldi-Frati, M.-A., Sorel, Y., Van., Q.-D.: The MeMVaTEx methodology: from requirements to models in automotive application design. In: 4th European Congress ERTS Embedded Real Time Software. Toulouse, France (January 2008)

[2] André, C.: Time Modeling in MARTE. In: FDL 2007 Forum on specification and Design Languages, Barcelona, Spain (2007)

[3] Cancila, D., Passerone, R.: Functional and structural properties in the Model-Driven Engineering approach. In: ETFA 2008 (2008)

[4] Bendraou, R., Desfray, P., Gervais, M.-P., Muller, A.: MDA Tool Components: a proposal for packaging know-how in model driven development. Software and System Modeling 7, 329–343 (2008)

[5] Cuccuru, A., Gérard, S., Radermacher, A.: Meaningful Composite Structures - On the Semantics of Ports in UML2. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301. Springer, Heidelberg (2008)

[6] Emery, D., Hilliard, R.: Updating IEEE 1471: architecture frameworks and other topics. In: Seventh Working IEEE/IFIP Conference on Software Architecture WICSA (2008)
[7] Espinoza, H., Servat, D., Gérard, S.: Leveraging Analysis-Aided Design Decision Knowledge in UML-Based Development of Embedded Systems. In: SHARK at ICSE 2008, Leipzig (May 2008)
[8] France, R., Fleurey, F., Reddy, R., Baudry, B., Ghosh, S.: Providing Support for Model Composition in Metamodels. In: Proceedings of EDOC 2007, Annapolis, USA (October 2007)
[9] Gray, J., Tolvanen, J.-P., Kelly, S., Gokhale, A., Neema, S., Sprinkle, J.: Domain-Specific Modeling. In: CRC Handbook of Dynamic System Modeling. CRC Press, Boca Raton (2007)
[10] Hause, M., Thom, F.: Building Bridges Between Systems and Software with SysML and UML. In: INCOSE Intl. Symposium (June 2008)
[11] INTERESTED EU Project: Interoperable embedded systems Tool-chain for enhanced rapid design, prototyping and code generation, http://www.interested-ip.eu/index.html
[12] Johnson, T., Jobe, J., Paredis, C., Burkhart, R.: Modeling Continuous System Dynamics in SysML. In: Proceedings of the IMECE 2007 (November 2007)
[13] Lagarde, F., Espinoza, H., Terrier, F., André, C., Gérard, S.: Leveraging Patterns on Domain Models to Improve UML Profile Definition. In: Fiadeiro, J.L., Inverardi, P. (eds.) FASE 2008. LNCS, vol. 4961, pp. 116–130. Springer, Heidelberg (2008)
[14] Lambda Project, Lambda Libraries for Applying Model Based Development Approaches, Technical Annex (May 2008)
[15] Maier, M.: System and Software Architecture Reconciliation. Systems Engineering Journal, 146–159 (2006)
[16] OMG, Unified Modeling Language, UML™ Superstructure, V2.1.2
[17] OMG, Systems Modeling Language SysML™, V1.0
[18] OMG, UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, Beta 2
[19] SATURN Project: SysML bAsed modeling, architecTUre exploRation, simulation and syNthesis for complex embedded systems, http://www.saturnsysml.eu
[20] Selic, B.: From Model-Driven Development to Model-Driven Engineering. In: Keynote talk at ECRTS 2007 (July 2007)
[21] Selic, B.: A Systematic Approach to Domain-Specific Language Design Using UML. In: ISORC 2007, pp. 2–9 (2007)
[22] Soares, M.S., Vrancken, J.L.M.: A Proposed Extension to the SysML Requirements diagram. In: IASTED International Conference on Software Engineering, Austria (2008)
[23] Sifakis, J.: Embedded Systems - Challenges and Work Directions. In: Higashino, T. (ed.) OPODIS 2004. LNCS, vol. 3544, pp. 184–185. Springer, Heidelberg (2005)

Derivation and Refinement of Textual Syntax for Models

Florian Heidenreich, Jendrik Johannes, Sven Karol,
Mirko Seifert, and Christian Wende

Institut für Software- und Multimediatechnik
Technische Universität Dresden
D-01062, Dresden, Germany
{florian.heidenreich,jendrik.johannes,sven.karol,
mirko.seifert,c.wende}@tu-dresden.de

Abstract. Textual Syntax (TS) as a form of model representation has made its way to the Model-Driven Software Development community and is considered a viable alternative to graphical representations. To support the design and implementation of text editing facilities many concrete syntax and model mapping tools have emerged. Despite the maturity of these tools, users still spend considerable effort to specify syntaxes and generate editors even for simple metamodels. To reduce this effort, we propose to refine a specification that is automatically derived from a given metamodel. We argue that defaults in a customisable setting enable developers to quickly realise text-based editors for models. In particular in settings where metamodels evolve, such a procedure is beneficial. To evaluate this idea we present EMFText [1], an EMF/Eclipse integrated tool for agile TS development. We show how default syntax can easily be tailored and refined to obtain a custom text editor for EMF models and demonstrate our approach by two examples.

1 Introduction

Formal languages are the basis for most activities in computer science (e.g., to describe data or to specify behaviour). They consist of a syntax and a semantics, where the former describes the layout of valid sentences and the latter assigns meaning. Well designed languages usually have a syntax that can be easily understood and that supports the languages' semantics.

For a long time, syntaxes were textual, but during the last two decades graphical representations have become very popular. In particular Model-Driven Software Development has pushed the widespread use of graphical methods to represent (i.e., to model) software. Both types of syntaxes have their pros and cons. Graphical syntaxes are strong in showing relationships, quantities and provide the opportunity to zoom, which is useful to obtain an overview of the presented sentence. In contrast, textual syntaxes have a predefined reading order, which is valuable if sentences are interpreted in sequential order. Furthermore, in model environments that lack a central version controlled model repository textual representations can be easily compared and merged. Thus, graphical and textual

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, pp. 114–129, 2009.

syntax must not be considered alternatives, but rather complements. While each one can perfectly serve one purpose it might be inadequate for another one.

Textual syntax can be either generic or custom. Again, both have their pros and cons. Generic syntaxes are defined on the level of metamodelling languages. Thus, they are either instantly available or can be derived automatically for concrete modelling languages. Custom syntaxes need manual specification effort. The former are also often more verbose and do not reflect the semantics of a language nicely, in contrast to the latter, which are more compact and use symbols that ease reading. To exemplify the difference, consider the two definitions of a UML Transition shown in Listing 1.1.

```
1 Transition { source : State "StateA" target : State "StateB" }      // generic syntax
2 StateA -> StateB                                                   // custom syntax
```

Listing 1.1. Two concrete syntaxes for a UML Transition

Clearly the customised syntax in Line 2 is easier to read, but also requires additional specification effort. If users have metamodels for their languages and want to use textual syntax for their models, the question is, how the benefits of generic syntaxes (low effort) and those of customised syntaxes (easier readability, reflection of semantics) can be combined. This paper presents an approach to achieve this and therefore allows rapid and agile development of tooling for TS. By deriving a default syntax, which can optionally be refined, our tool EMFText aims to reduce the effort needed to develop tool support for TS. As we will show in this paper, not only the initial specification of syntaxes, but also the adaptation of syntaxes to evolving metamodels, can profit from the approach. With this we follow a new direction in TS development, as opposed to previos approaches which we discuss in Section 5, which decreases the effort of syntax development in particular in the initial development phase.

The remainder of this paper is organised as follows: Section 2 describes how models can be represented as text. In particular, we discuss the options that exist both for deriving a default syntax as well as customising it later on. Section 3 provides detailed information about EMFText—the tool that implements our approach. Equipped with this, we present the concrete syntax definition for two modelling languages in Section 4, revealing the strengths and drawbacks. Section 5 compares the presented work with related approaches and Section 6 concludes this paper.

2 Approach

In this section, we first study how concepts used in metamodelling map to concepts used in grammar engineering. Based on this knowledge, we explore how default values for a concrete TS can be automatically derived for a concrete modelling language (i.e., a concrete metamodel). Then we discuss how the derived TS can be further refined and tailored if needed. As one example for the derivation of

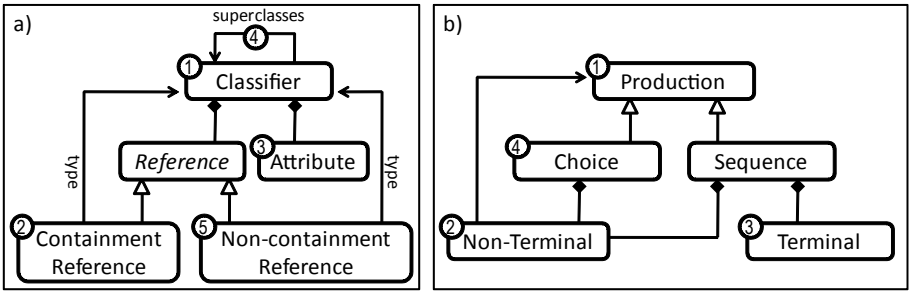


Fig. 1. Concepts of modelling languages (a) and text languages (b)

default syntaxes, we will use the Human-Usable Textual Notation (HUTN) [2]—which is one but not the only possible way of deriving an initial syntax[1].

2.1 Mapping Modelling Concepts to Text Language Concepts

Before we can derive and refine a TS for a concrete modelling language, we must recapitulate which concepts are available in textual syntaxes and how they relate to the concepts used in modelling languages. Typically, textual languages are defined by context-free grammars and modelling languages are defined by metamodels. Hence, a mapping between metamodelling and grammar engineering concepts must be established. We show these concepts side by side in Fig. 1. In the following we first summarise the core concepts of metamodelling (based on [3]) and of grammar engineering (based on [4,5]). Afterwards (Sect. 2.2), we discuss the mappings between the concepts. This discussion is based on [6].

Models defined in modelling languages are composed of elements that in turn consist of attribute values and other contained elements. In addition, cross-references between elements can exist. Which elements are allowed is defined in a metamodel through classifiers (1a). By defining containment references between classifiers, the possible containment relations are defined (2a). Which attribute values can be defined for an element is declared by attributes in the metamodel (3a). Classifiers are also connected through superclass relationships, which expresses exchangeability (4a). Possible cross-references are defined by non-containment references (5a).

Sentences written in a textual languages consist of a sequence of elements, where an element is either represented by a single symbol (i.e., a set of connected characters) or another nested sequence, which again consists of symbols and possibly other nested sequences. The allowed parts of textual languages are defined in a grammar through productions (1b). By referencing other productions through non-terminals in a sequence, the possible nesting of elements is defined (2b). The symbols that may appear in a sequence are defined by terminals (3b). Choices of different non-terminals can be defined to express exchangeability (4b).

[1] See Line 1 in Listing 1.1 for an example of HUTN syntax.

From these core concepts of both language engineering approaches, the following mapping can be derived. (1a–1b) Element types are defined through classifiers on the one side and by productions on the other. (2a–2b) The composition of elements from others is expressed by containment references on the metamodel side. Non-terminals that appear in a sequence express a similar relationship on the grammar side. (3a–3b) Attribute values are also part of a model element and can therefore be expressed by symbols. Consequently the attribute concept is mapped to the terminal concept. (4a–4b) The superclass relationship can be mapped to the alternative concept because both express exchangeability. (5a–2b) For cross-references, there exists no direct correspondence in text. They can however be mapped to symbols as well. In this case, the symbol would represent an identifier that identifies the element to be referenced.

2.2 Derivation and Refinement

After discussing how modelling languages can be mapped to textual syntaxes in general, the question is, what this mapping looks like for concrete modelling languages. Our design objective for such a mapping is to enable rapid and compact text syntax definition. Keeping this in mind, we want to automatically derive a default text syntax and then allow the developer to refine the generated syntax in an intuitive and incremental manner.

Deriving a default text syntax can be performed using a) the language's metamodel and b) assumptions based on standards and best-practices. Hence, these are the two parameters for this derivation process. We will shortly see where one or the other is used to obtain parts of the default syntax. Following the argumentation from Sect. 2.1 the derivation of syntaxes can be studied starting from the elements of modelling languages. For example, each metaclass must be mapped to a production. Thus, we will examine what such concrete default mappings can look like. Once an initial mapping is obtained, it can be refined. Since the options for the design of the default mapping and the customisation are similar, we will discuss both together for each modelling concept.

Classifiers define the types of model elements that can be used in a model. A representation of model elements as text, must disambiguate which class an element belongs to. The most basic way to do so is to use a distinct terminal symbol (keyword) for each classifier. HUTN realises this idea and uses the names of the classifiers from the metamodel as keywords. In subsequent customisation steps these default keywords can be changed. If classifiers can be disambiguated based on other properties of their text syntax, the keywords can be removed during the refinement process.

A *Containment* of model elements is expressed by *nesting* sequences. This is achieved using non-terminals in sequences. These define the children (elements in containment references) of the element defined by the parent sequence. Since the containment relation between metaclasses is fully defined by the metamodel the default nesting can therefore be directly derived. As the tree structures of the metamodel and the grammar must match, the nesting is dictated by the containment relation. However, the order in which contained elements are given

in TS can be refined. HUTN allows an arbitrary order, which can be restricted by further customisation steps.

Attributes are typed properties of classifiers. Classifiers can have multiple attributes. Consequently, the textual representation of an attribute must identify both the attribute itself (e.g., using its name) and its value, as well as, the classifier instance the attribute belongs to. A default approach to have this three things available is to a) define attributes in the context of their classifier (e.g., directly after the keyword), and b) state the name of the attribute in the text together with its value. Thus, two terminals are sufficient to represent an attribute and its value. HUTN adds a third one (a double colon) to ease reading for humans. The default definition for the terminals used for attibute values can be obtained using the type of the attribute. For primitive types this mapping is straightforward. Numbers are mapped to text using their arabic representation, boolean values are `true` and `false`, strings are enclosed in double quotes and characters are surrounded by single quotes. Refining this mapping can be done in different ways. The tokens for the attribute names can be omitted if an order is defined upon the set of attributes or if all attributes have distinct types. Values can be mapped differently (e.g., boolean values can be represented as `yes` and `no`).

The *Inheritance* hierarchy of the metamodel defines that different types of elements are allowed at certain positions. Consequently, *Alternatives* of sequences are allowed at positions where different types of model elements (with a common supertype) may occur. The alternatives can be derived from the inheritance hierarchy and can not be altered (without changing the metamodel).

Non-containment references connect model elements and establish a graph structure. To represent such references in text, names (or identifiers) must be used. To obtain unambiguous identifiers an attribute having some uniqueness constraint (e.g., *name* or *id*) can be used. A default setting is thus to use the values of such attributes to reference model elements. However, sometimes it is not clear which attribute must be used or scoping rules apply. In this case the handling of names must be refined. To do so, both the identification of elements and the referencing rules (e.g., scoping) must be defined.

To summarise, a complete instance of a *Classifier* (i.e., a complete model element with attributes and references) is represented by a *Sequence* of terminal symbols (e.g., keywords and identifiers) and non-terminals. In the particular case of HUTN, an initial sequence definition for a metaclass starts with a keyword that equals the name of the metaclass name followed by terminal symbols and names for the element's attributes and references in an arbitrary order.

3 Overview of EMFText

To first derive and later refine a TS, developers require a tool that 1) performs the derivation, 2) provides the facilities to refine the derived syntax, 3) supports the evolution of the TS along with the metamodel it is based on and 4) generates the tooling to create and manipulate models using the TS. Existing TS tools lack support for one or more of these requirements (see Sect. 5). Therefore, we

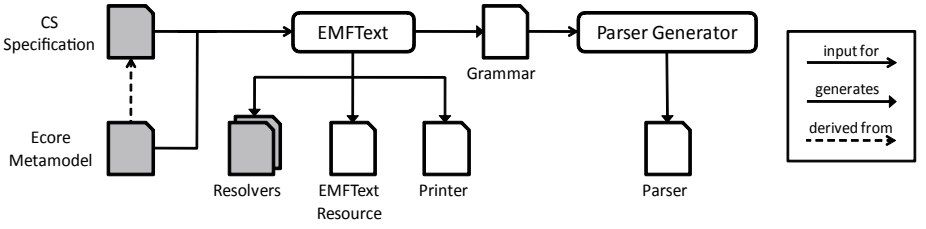


Fig. 2. Overview of Specified and Generated Artefacts

implemented EMFText that supports all the four functionalities in combination. With EMFText we evaluate the theoretical analysis of Sect. 2 on different examples in Sect. 4. This section therefore introduces EMFText, and how it supports the four requirements (cf. Fig. 2).

To provide the refinement capabilities discussed in Sect. 2.2, EMFText provides a specification language called *CS*. An initial CS definition is therefore the result of the derivation performed by EMFText (upper left of Fig. 2). Consequently, an initial CS is obtained without any effort (as we will see in Sect. 4). In CS specifications, the syntax for model elements is defined using concepts from the Extended Backus Naur Form (EBNF). This is, because these concepts from grammar engineering are well-known and are therefore a practical tool for manual refinements. For refinements that go beyond grammar engineering as the mapping of identifiers to cross-references (Sect. 2.1), EMFText generates a Java API. Here a specification of how identifiers are resolved to the respective elements can be implemented (lower left of Fig. 2). Again, to minimise specification effort, a default implementation is provided by EMFText, where identifiers are resolved to elements of the correct type having the correct name or id.

When metamodels evolve over time, the syntax might become incomplete or invalid. In the first case, the textual syntax is incrementally derived for new metamodel elements and syntax of removed metamodel elements is deleted. Developers do not need to spend time to adapt their syntaxes to the changes made to metamodels in these cases. However, changes in existing metaclasses with a refined textual syntax cannot automatically be merged into an existing CS specification. In such situations EMFText informs the developer of necessary syntax refinements by carefully analysing the CS specification. This analysis process, which also ensures correctness during refinement, checks that the CS definition matches the metamodel, by checking that 1) each production (rule) corresponds to a concrete metaclass, 2) that there is a rule for each concrete metaclass and 3) all references defined in the metamodel are present in the syntax. Since CS is itself defined with Ecore and EMFText, all these checks are performed on a CS model (i.e., an instance of the CS metamodel).

Technically, EMFText is implemented as a set of plug-ins for the Eclipse platform [7], tightly integrated with the Eclipse Modelling Framework (EMF) [3]. In addition to the mentioned resolvers, several artefacts are generated by EMFText based on a CS to provide runtime tooling (right side of Fig. 2). First, a

parser is derived that can read text syntax conforming to the CS productions and that creates a model. Second, the counterpart—a printer—that transforms models to text is derived from the same specification. Third, an implementation of the *Resource* interface of EMF is generated through which parser and printer are plugged into EMF's resource management (making them accessible to all EMF-based modelling tools). For the parser generation, EMFText is generally not bound to a specific technology (i.e., additional technologies can be plugged in). However, currently we support only ANTLR [8]. We mention this, because the concrete parsing approach used can influence the CS specification. Therefore, the CS analysis mechanism described above can be enriched with checks specific to the used parser technology. For instance, ANTLR cannot handle left recursion. Thus, a mechanism for automatically resolving direct left recursive rules was provided. Other left recursive rules are detected on the CS level and users are encouraged to refactor their syntax definitions. If a different parser generator is used, the detection of left recursive rules might become obsolete, but other analysis might be needed to make sure the CS meets the criteria of the respective parser generator.

4 Examples

The previous sections introduced the conceptual and technological foundations for development of concrete syntax for modelling languages with EMFText. Here, we exemplify development with EMFText by two well-established modelling languages—feature models and UML state machines—and show how an automatically derived TS can be tailored by several refinement steps. More examples can be found at [9].

4.1 Parsing and Printing of Concrete Syntax for Feature Models

Variability modelling is at the heart of product-line engineering [10]. One widely used notation to express variability between different products in a product line are feature models [11]. The feature metamodel depicted in Fig. 3 was taken from the FeatureMapper [12], a tool for mapping features to models. A central requirement for the development of the FeatureMapper was its compatibility with different existing tools. Thus, it uses a simple and flexible abstraction as feature metamodel which covers a broad range of existing feature model derivatives.

Feature models describe hierarchical tree structures between features and groups of features. These models are usually created using a diagrammatic notation, but as these models can grow for large product lines, an alternative textual representation seems to be beneficial. Hence, providing adequate means for both parsing and printing nested language elements are requirements for our concrete syntax specification.

Generating a Default Concrete Syntax. The starting point in the development of a TS for feature models is the concrete syntax specification language *CS* of EMFText. As discussed in Section 3, CS specifications are tightly coupled

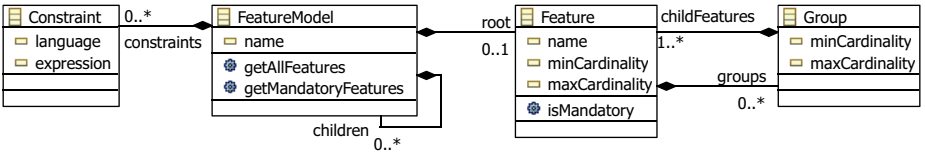


Fig. 3. Metamodel for Feature Models

to one ore more EMF-based metamodels whose structure implicitly defines a grammar skeleton. For every non-abstract metaclass a production rule can be specified. Hence, a production's left-hand side does not only correspond to a non-terminal in a context-free grammar but also to a metamodel element with the same name. Elements on the right-hand side of the production are mapped to features (i.e., attributes or references) of the metaclass. To kick-start the development of a new concrete syntax, EMFText generates a default syntax for every non abstract metaclass based on the conventions introduced by the HUTN standard. HUTN aims at providing a generic, human-readable textual syntax for all kinds of MOF-based metamodels. For this purpose, it derives the syntactical representations of model elements from the properties and relationships of the corresponding metaclass.

An example using the specified syntax for the initialisation of a concrete Feature with the name "SimpleFeature" is shown in Listing 1.2. Every Feature declaration starts with the keyword "Feature". Encapsulated in curly braces initialisations for all properties and references of the Feature in arbitrary order can be given. Every single property initialisation is preluded by a keyword corresponding to the name of the property and a colon.

```
1 Feature { name : "SimpleFeature" }
```

Listing 1.2. Feature declaration using the HUTN syntax

Concrete Syntax Tailoring. We argue that HUTN is not acceptable for a broad audience since it neither provides syntax tailored to a language's application domain nor reflects language-specific semantics. Thus, we propose an optional but recommended tailoring of the generated grammar productions to construct a more usable syntax. The code listing in Fig. 4 compares the initially generated HUTN-based syntax specification with the adapted feature model syntax after tailoring. Lines that have not changed during syntax tailoring span the full width of the listing. Differing lines are shown side by side.

Line 1 assigns a name to the concrete syntax, which can later be used as the file name extension for model instances (here `featuremodel`), refers to a Ecore metamodel registered in the system under the given URI, and defines which metaclass is used as the root element of a model and the start symbol for the parser.

```
1  SYNTAXDEF featuremodel FOR <http://www.tudresden.de/feature> START FeatureModel
2  RULES {
3  FeatureModel ::= "FeatureModel"
4    "{" ( "constraints"  ":" constraints |                    name['"','"']
5    "root"  ":" root |                                        ( "{" "constraints"  ( constraints ";")? "}")?
6    "name"  ":" name['"','"'] )* "}" ;                        root;
7  Feature ::= "Feature"
8    "{" ( "name"  ":" name['"','"'] |                         name['"','"']
9    "minCardinality"  ":" minCardinality[INTEGER] |           ("(" minCardinality[INTEGER] ".."
10   "maxCardinality"  ":" maxCardinality[INTEGER] |           maxCardinality[INTEGER] ")")?
11   "groups"  ":" groups )* "}" ;                             ( groups* )?  ;
12 Group ::= "Group"
13   "{" ( "minCardinality"  ":" minCardinality[INTEGER] |     ("(" minCardinality[INTEGER] ".."
14   "maxCardinality"  ":" maxCardinality[INTEGER] |           maxCardinality[INTEGER] ")" )?
15   "childFeatures"  ":" childFeatures  )* "}" ;              ("{" childFeatures* "}")?;
16 Constraint ::= "Constraint"
17   "{" ( "language"  ":" language['"','"'] |                 language[] ":"  expression['"','"'];
18   "expression"  ":" expression['"','"']  )* "}" ;
19 }
```

HUTN-based Syntax | Tailored Syntax

Fig. 4. CS specification for Feature Models

The rules section (Lines 2–19) contains one production rule for each concrete metaclass. On the right-hand side of each rule we find 1. containment references (e.g., `childFeatures` in Line 15) that define the positions of nestings, 2. attributes (e.g., `name` in Line 6) that define the position of attribute values, and 3. keywords (e.g., "`FeatureModel`" in Line 3) that are pure concrete syntax and have no counterpart in the metaclass.

For attributes, which in contrast to references are followed by square brackets, the allowed symbols as well as the mapping from such a symbol to a value in the model can be customised. The first can be performed by using predefined (e.g., `INTEGER` for `minCardinality` in Line 9) or custom defined token types (can be defined with regular expressions in an extra section). A standard type that allows a sequence of all letters and numbers is used by default if nothing is specified (e.g., `name` in Line 6). The mapping can be adjusted by specifying a prefix and a suffix instead of a token type (e.g., " and " for `name` in Line 4). This causes EMFText to remove the prefix and suffix during parsing and to add them during printing such that they never occur in the model.

The comparison of the generated and the tailored syntax shows that many parts of the generated specification were reused or only slightly changed. In some productions unneeded keywords were deleted, a few productions were changed more invasively. In summary, the automatically generated syntax based on HUTN provided guidance for refinement and tailoring of the syntax specification.

From the tailored CS specification, EMFText derives a context-free grammar which is exported as an ANTLR grammar specification. In accordance to the inheritance structure in the metamodel alternatives are derived to express the interchangeability of a type's subtypes. The ANTLR specification is also annotated with semantic actions to instantiate the model-representation for the

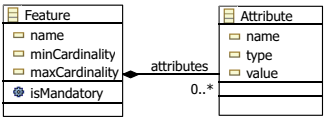


Fig. 5. Extension of Metamodel for Feature Models

parsed expressions. Finally the ANTLR tooling is used to generate the implementation of the lexer and parser. The LOC-ratio between a CS-Specification and the resulting ANTLR grammar is about 1 : 10.

As a second implementation artefact, a pretty printer is generated from the syntax specification. This printer transforms a given model to its textual representation. Often special layout conventions which do not interfere with language syntax and semantics are used to provide a more readable and comprehensive syntax. Two constructs can be used in CS specifications to incorporate these layout conventions. First, the `LineBreak` element (`!<n>`) causes a line break during printing and an indentation in the next line to a level of `<n>`. Seconds, the `Whitespace` elements ♯`<n>`) inserts `<n>` whitespaces during printing. Using these constructs is optional.

Evolving the Concrete Syntax. During the development of the FeatureMapper we faced several situations where the feature metamodel had to be adapted. Such an evolution is a common issue in language development. However changes in a languages metamodel necessitate the co-adaptation of its textual syntax. A concrete example for the feature metamodel was the introduction of `Attribute`s for `Feature`s (cf. Fig. 5).

To adapt the tailored textual syntax for feature models in accordance, the CS specification is automatically extended by EMFText with an additional HUTN-based syntax rule for `Attribute`s. In addition, the syntax rule in Line 3–6 of Figure 4 has to be extended to include `Attribute`s in `Feature`s. We decided not to automate the inclusion of a corresponding containment reference, since the rule was already tailored. Instead, EMFText adds a warning to the CS specification to inform the developer of the necessary syntax refinement. This exemplifies how derivation and stepwise refinement helps the co-adaptation of TS for evolving languages.

4.2 Resolving References in UML State Machines

Our second example defines a textual syntax for UML State Machines. It uses on an existing UML metamodel implementation that can be found at [13]. Listing 1.3 specifies a text syntax for state machines and Fig. 6 shows an exemplary state machine opened in the EMF text editor. Note that the metamodel defines `Vertex` as a superclass of `State`, `FinalState` and the initial state (represented as `Pseudostate` if kind initial). Thus, their productions can be alternatively used for the non-terminal `subvertex` in Line 5. Since the syntax defined in Listing 1.3

```
1  SYNTAXDEF statemachine FOR <http://www.eclipse.org/uml2/2.1.0/UML>
2  START StateMachine
3  RULES {
4    StateMachine  ::= "StateMachine" name[] "{" region "}" ;
5    Region        ::= subvertex* "transitions" "{" transition* "}";
6    State         ::= "state" name[] "{" ("entry" ":" entry)? ("exit" ":" exit)?
7                      "do" ":" doActivity "}" ";";
8    Pseudostate   ::=  kind[] "state" name[] ";";
9    FinalState    ::= "final" "state" name[] "{" ( "entry" ":" entry )?
10                     ( "exit" ":" exit )? "do" ":" doActivity "}" ";";
11   Transition    ::= source[] "->" target[] "when" trigger
12                     ( "do" ":" effect )? ";";
13   Trigger       ::= name['"','"'];
14   Activity      ::= name['"','"'];
15 }
```

Listing 1.3. Concrete syntax for UML state machines

concentrates on state machines, only the `name` of an `Activity` (Line 14) can be specified (and no further details).

In contrast to the metamodel for feature models—which defined a strict tree structure, by only utilising *containment* references (represented by filled diamonds in Fig. 3)—the UML metamodel also utilises *non-containment* references. Non-containment references are used when an element should point to an element that is defined elsewhere.

In a CS specification, terminals can be defined for non-containment references in the same way they are defined for attributes. A token parsed for a terminal represents the *name* of the referenced element. These names have to be resolved to the actual elements. In the UML state machine syntax, the non-containment references `source` and `target` of `Transition` (Listing 1.3, Line 11) are used.

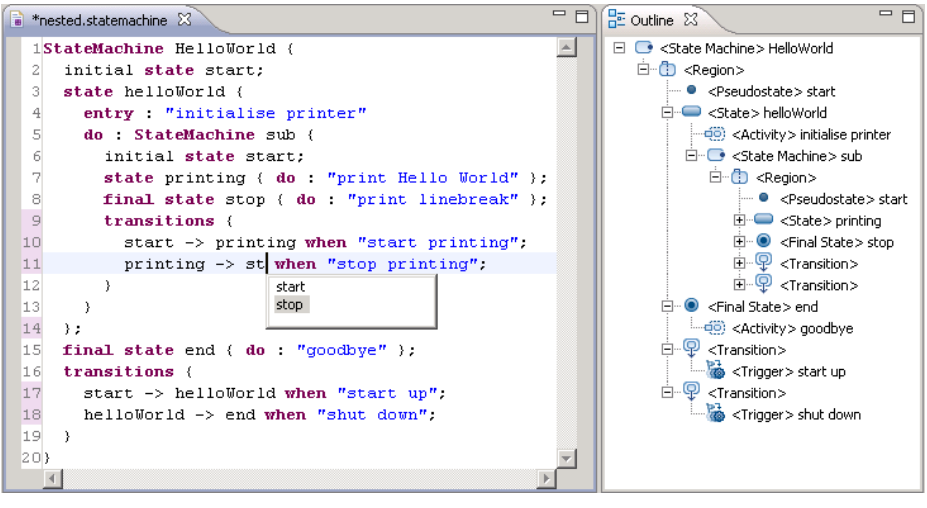


Fig. 6. A UML state machine edited using it's concrete syntax in the EMFText editor

Default Reference Resolving. EMFText provides a default algorithm to resolve references. It uses type information provided by the metamodel to find candidates in the parse tree (e.g., the `target` of a `Transition` must be a `State`). Further, it compares the name of each candidate with the terminal representing the reference to find a match. For that the candidates need to provide a `name` attribute. For every match found, the terminal is replaced with a direct reference to the found element (a `State` in this example). During pretty printing, the references are de-resolved. That is, the referenced object is replaced by a terminal constructed from the `name` attribute.

Tailoring Reference Resolving. By default, a `name` attribute is expected to identify the referenced element unambiguously. For models defined in a large language like UML, this convention does not always apply. For instance, the `doActivity` containment reference of `State` is of type `Behaviour` (Listing 1.3, Line 10). `Activity` and `State` are two subclasses of `Behaviour`. Thus, the UML metamodel for state machines allows to specify the behaviour of a state by embedding a subordinated state machine. In the state machine in Figure 6 for example, there are two `States` named *start* (Lines 2 and 7). UML specific scoping rules apply for the visibility of `States` to avoid name clashes between nested machines. Naturally, the default resolving does not know these rules and therefore requires customisation.

For this purpose, EMFText provides the means to flexibly adapt the semantics for matching a name with potential reference candidates. For every non-containment reference a `ReferenceResolver` with two template methods, `doResolve()` for resolving and `doDeResolve()` for de-resolving, is generated. By default, the methods call their super implementation which implements the default reference resolving behaviour defined above. Hence, a developer can extend or override the default behaviour in this methods. In successive generation steps, the already generated and adapted resolvers are not overwritten.

In the case of our simplified UML state machines two references need to be resolved. Thus, two resolvers: `TransitionSourceReferenceResolver` and `TransitionTargetReferenceResolver` are generated by EMFText. Both can be implemented in a uniform manner. Listing 1.4 shows the implementation of the `doResolve()` method for the latter. Every ReferenceResolver extends the generic type `AbstractReferenceResolver<T>` which gets bound with the type the resolver is meant to resolve (in this case `Transition`). The parameters for `doResolve()` include the `name` to be resolved, the `container` of the unresolved reference (in this case `Transition`), the boolean parameter `resolveFuzzy`, and a `ResolveResult` into which results of the resolution as well as error or warning messages can be placed. In Lines 5–15 only `Vertices` that are defined within the same `Region` as the current `Transition` are considered. This realises the scoping.

If the `resolveFuzzy` flag is turned on, the proxy resolver is asked for all elements that might match the given name. This is used by the text editor to collect suggestions for code completion. Figure 6 shows the completion proposals for transitions starting with “st”. Fuzzy resolution usually differs only slightly

```
1  public class TransitionTargetReferenceResolver extends AbstractReferenceResolver<Transition> {
2    ...
3    protected void doResolve(String name, Transition container,
4            boolean resolveFuzzy, IResolveResult result) {
5      for (Vertex targetCand: container.getContainer().getSubvertices()) {
6        if (resolveFuzzy) {
7          if (targetCand.getName().startsWith(name)) {
8            result.addMapping(targetCand.getName(), targetCand);
9          }
10       } else {
11         if (targetCand.getName().equals(name)) {
12           result.addMapping(identifier, targetCand);
13         }
14       }
15     }
16     if (result.isEmpty()) {
17       result.addError("Vertex " + name + " not defined");
18     }
19   }
20 }
```

Listing 1.4. `TransitionTargetReferenceResolver` implementation

from exact resolution. In the example (Listing 1.4) only `equals()` (Line 11) was exchanged for `startsWith()` (Line 7). If a developer does not care about fuzzy resolution and code completion, the flag can simply be ignored. The default resolvers, however, do implement fuzzy resolution. Thus, code completion is provided out of the box for all cases were the default resolution is not overridden. For de-resolving the default implementation can be kept here. If defined anyway, the code would consist of a statement that reads the `name` attribute of a given `State` and returns it.

Tailoring Token Resolving. EMFText also supports to tailor the mapping of tokens (i.e., strings) to attribute values in the model. For instance, an input string valued `yes` (`no`) can be converted to the boolean value `true` (`false`). As this is a bi-directional mapping (from tokens to attribute values and vice versa), we call the two complementary processes resolving and de-resolving. For each token type, a `TokenResolver` is generated in which additional conversion rules can be implemented. Note that this is only needed for special mappings. Common conversions (e.g., for numbers) are already available.

5 Related Work

As discussed in [14], both generic and custom textual syntax can be beneficial for different application scenarios. Consequently, a variety of different TS tools and approaches exist. Some provide TS based on metamodelling languages (e.g., [14]), while others are explicitly designed for the specification of custom syntaxes. Goldschmidt et al. [15] presented a comprehensive overview of existing approaches and classified them within a multi-faceted classification schema.

In contrast to existing approaches, which either focus on one type of syntax or the other, our approach and the presented implementation (EMFText), was designed with the combined focus on refinement, derivation and evolution. Thus, it combines the strengths of both types of syntaxes. Presenting the differences between EMFText and all other approaches in detail is beyond the scope of this paper. Nonetheless, we will compare EMFText according to the classification in [15] with determined tools that share most of EMFText's concepts and architectural decisions and are thus most related to our work.

Rose et al. [14] presented an implementation of the generic syntax HUTN. The problems adressed by this implementation (rapidly changing metamodels) can be countered with EMFText as well. The tooling generated from a HUTN syntax specification that can be automatically derived by EMFText is very similar to the one provided by [14]. However, we allow users to refine this syntax where appropriate, which is not possible with a pure HUTN implementation. Furthermore, we can automatically generate syntax for new meta classes, keeping the customised syntax for existing ones.

The definition of customised syntax is supported by variety of tools. To our knowledge none of these tools provides support to either derive, refine or evolve syntax definitions. However, there is more differences we want to mention here.

Similar to EMFText, TCS [16] has a tight coupling between the mapping definition and the metamodel, which facilitates the definition of a syntax. It provides a generic editor for registered syntaxes, including syntax colouring, error marking, and navigation to underlying model elements. However, the latter comes at the price of a mandatory base class for all metamodel elements handled by the TCS editor. This is a limitation compared to EMFText's implementation which stores information regarding column and line numbering externally.

Sintaks [17] is built upon the Kermeta metamodelling facility and uses a concrete syntax metamodel to generate transformations between text and models. In contrast to EMFText the mapping between syntax rules and metamodel elements is explicit. The concrete syntax has to be specified via an EMF tree-based editor, which is not as convenient as using our specification language CS.

TEF [18] uses an interpretative approach (in contrast to the generative approach used in EMFText). Additionally, each rule needs to be associated explicitly with classes and properties of the metamodel using metamodel bindings. This differs from the implicit mappings used in EMFText which facilitates much shorter mapping descriptions. TEF supports background parsing and a Model View Controller (MVC) update strategy.

Monticore [19] supports an integrated specification of concrete and abstract syntax. It provides similar editing features as EMFText. Because of its integrated approach, Monticore is well suited to define a metamodel and an editor for a given textual language. This is a clear difference to our approach which targets to easily derive and refine concrete textual syntax for existing metamodels.

XText [20] is tightly integrated with EMF and is based on ANTLR. As Monticore, XText provides an integrated specification language and similar editing

features as EMFText. However, neither it can be used to define more than one syntax per metamodel nor it allows to compose syntax for existing metamodels.

6 Conclusion

In this paper we presented a novel approach for defining textual concrete syntax for modelling languages based on derivation and refinement. We introduced the conceptual foundation for that approach and presented its implementation in the tool EMFText. Finally, we showed usage and applicability of approach and tool on different examples. EMFText allowed us to quickly define several text syntax for two modelling languages. The possibility to start with a generated default syntax but having all the customisation options turned out to be very helpful—in particular for first-time users.

In the future, we plan to extend our text editor and investigate further usages of the concrete text syntax. The tight integration with EMF and Eclipse can be extended to connect to other functionality which profits from textual representations, like diff and merge in version control. We also plan to improve EMFText to support a more declarative specification language for name analysis. The language will provide adequate support to declare different namespaces and scoping rules.

Currently, our tool implementation relies on ANTLR to generate parsers. However this imposes some restrictions: As a recursive descent LL(*) parser generator it does not support general left recursive grammars [4]. Furthermore ANTLR parsers use an extra scanner component to convert input character streams into token streams which may cause conflicts when composing syntax definitions with intersecting token definitions. To improve EMFText in these directions we plan to support alternative backends such that parser generators can be exchanged accordingly. The effort of transforming left recursive grammars into right recursive ones can then be avoided by switching to a tool that can handle left recursive grammars, for example a LR based parser generator [4]. Moreover, a scannerless parsing approach can be applied to prevent conflicts between token definitions [21].

Furthermore, we will define and analyse TS for more languages to find possible improvements to our approach. One direction here is to utilise EMFText for *type-safe* code generation—using model-to-model transformations and EMFText's ability to print models into text. Another, yet related, idea is to extend metamodels to obtain type safe template languages. We have done work into this direction in [22]. The question here is how a TS can be automatically extended along with a metamodel.

Acknowledgement

This research has been co-funded by the European Commission within the 6th Framework Programme project MODELPLEX #034081, by the German Research Foundation within the project HyperAdapt and by the German Ministry of Education and Research within the projects feasiPLe and SuReal.

References

1. TU Dresden: Software Technology Group: EMFText (2008), http://emftext.org
2. Object Management Group: Human Usable Textual Notation (HUTN) Specification. Final Adopted Specification ptc/02-12-01 (2002)
3. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: Eclipse Modeling Framework, 2nd edn. Pearson Education, London (2008)
4. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers – Principles, Techniques, and Tools. Addison-Wesley, Reading (1986)
5. Meyer, B.: Introduction to the Theory of Programming Languages. Prentice Hall, Englewood Cliffs (1990)
6. Kleppe, A.: Software Language Engineering. Pearson Education, London (2009)
7. The Eclipse Foundation: Eclipse Platform (2008), http://www.eclipse.org
8. Parr, T.: ANTLR — ANother Tool for Language Recognition — parser generator (October 2008), http://www.antlr.org
9. TU Dresden: Software Technology Group: EMFText: Concrete Syntax Zoo (2008), http://emftext.org/zoo
10. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, Heidelberg (2005)
11. Kang, K., Cohen, S., Hess, J., Nowak, W., Peterson, S.: Feature-oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Pittsburgh, PA (1990)
12. Heidenreich, F., Kopcsek, J., Wende, C.: FeatureMapper: Mapping Features to Models. In: Companion Proc. of ICSE 2008. ACM, New York (2008)
13. The Eclipse Foundation: EMF-based implementation of UML2 metamodel (2008), http://www.eclipse.org/modeling/mdt/?project=uml2
14. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.: Constructing Models with the Human-Usable Textual Notation. In: Proc. of the MoDELS 2008, Toulouse, France, pp. 249–263 (2008)
15. Goldschmidt, T., Becker, S., Uhl, A.: Classification of Concrete Textual Syntax Mapping Approaches. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 169–184. Springer, Heidelberg (2008)
16. Jouault, F., Bézivin, J., Kurtev, I.: TCS: A DSL for the Specification of Textual Concrete Syntaxes in Model Engineering. In: Proc. of GPCE 2006. ACM, New York (2006)
17. 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)
18. Scheidgen, M.: Textual Modelling Framework, http://www2.informatik.hu-berlin.de/sam/meta-tools/tef/
19. Krahn, H., Rumpe, B., Völkel, S.: Efficient Editor Generation for Compositional DSLs in Eclipse. In: Proc. of DSM 2007, Montreal, Quebec, Canada, Technical Report TR-38, Jyväskylä University, Finland (2007)
20. Efftinge, S., Völter, M.: oAW xText: a framework for textual DSLs. In: Workshop on Modeling Symposium at Eclipse Summit (2006)
21. Scheerder, J., Vinju, J.J., Visser, E.: Disambiguation filters for scannerless generalized lr parsers. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 143–158. Springer, Heidelberg (2002)
22. Henriksson, J., Heidenreich, F., Johannes, J., Zschaler, S., Aßmann, U.: Extending Grammars and Metamodels for Reuse: The Reuseware Approach. IET Software 2(3), 165–184 (2008)

Uniform Random Generation of Huge Metamodel Instances

Alix Mougenot*, Alexis Darrasse, Xavier Blanc, and Michèle Soria**

UPMC Paris Universitas, LIP6, France

Abstract. The size and the number of models is drastically increasing, preventing organizations from fully exploiting Model Driven Engineering benefits. Regarding this problem of scalability, some approaches claim to provide mechanisms that are adapted to numerous and huge models. The problem is that those approaches cannot be validated as it is not possible to obtain numerous and huge models and then to stress test them.

In this paper, we face this problem by proposing a uniform generator of huge models. Our approach is based on the Boltzmann method, whose two main advantages are its linear complexity which makes it possible to generate huge models, and its uniformity, which guarantees that the generation has no bias.

1 Introduction

The size and the number of models is drastically increasing, preventing organizations from fully exploiting MDE (Model Driven Engineering) benefits [9]. Today systems are already composed of hundreds of models whose size is quite often close to the thousand of model elements [13]. Regarding the evolution of system complexity [4], one can easily observe that scalability is the most critical of today's (and tomorrow's) challenges.

Approaches that address scalability issues claim to propose adapted mechanisms that deal with numerous and huge models. However, they lack of a complete large-scale validation. Indeed, as it is very difficult to obtain numerous and huge models, it is not possible to really stress test them.

To face this problem, one possible approach is to gather numerous and huge models into open repositories [8]. The idea is to ask large organizations to populate the repositories by providing their larger models. This approach is however not really convincing because, as said by the authors themselves, "one of the main challenges was to find a good quantity of models". Indeed, only 150 models have been stored in the Moogle repository [8], which correspond to 80 thousands model elements, and is therefore not sufficient to realize large-scale stress test.

In this paper, we propose another approach that consists in a uniform generator of huge models. Indeed, we argue that (1) the generator should be uniform[1]

* This work was partly funded by the french DGA.

** work partially supported by ANR contract GAMMA, n°BLAN07-2_195422.

[1] By uniform we mean that for a finite class of objects $\mathcal{C}$, any object of $\mathcal{C}$ is produced with equal probability $1/\text{card}(\mathcal{C})$.

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, pp. 130–145, 2009.

in order to be used to validate existing approach without introducing any bias and that (2) the generated models should be huge (millions of model elements) in order to measure the scalability of the approaches.

As we will detail in section 5, most of existing approaches that provide generators of models aim at generating constrained models. Their objective is to find, if possible, models that are consistent regarding a set of constraints. Those approaches are based on constraint solvers and hence have difficulties in generating huge models.

Our approach is based on the Boltzmann method [2] whose two main advantages are its linear complexity which makes it possible to generate huge models, and its uniformity, which guarantees that the generation has no bias.

This article is structured as follows. Section 2 presents the Boltzmann method. Section 3 presents our contribution that is to exploit the Boltzmann method to generate huge models. Section 4 then presents our realization, then section 5 presents works related to the problem of models generation and section 6 presents our conclusion.

2 Boltzmann Random Generation of Trees

Our approach is based on the random sampling of combinatorial structures, within the frame of Boltzmann method, as introduced in [2] (see [11] for an up-to-date review of the method's developments and applications). The main feature of this method is uniform generation with linear complexity, thus allowing for generation of much larger objects than was possible before.

Most random generation methods deal with finite classes of objects, usually objects with a given size, for example binary trees of size one thousand. In the case of Boltzmann method, the notion of uniformity is extended to classes of objects for which the cardinality is infinite, like binary trees of any size. The Boltzmann method only guarantees uniformity for structures of the same size, with the constraint that there is a finite number of elements having the same size. For instance, the number of possible binary trees is infinite, but there is a finite number of binary trees for a given size.

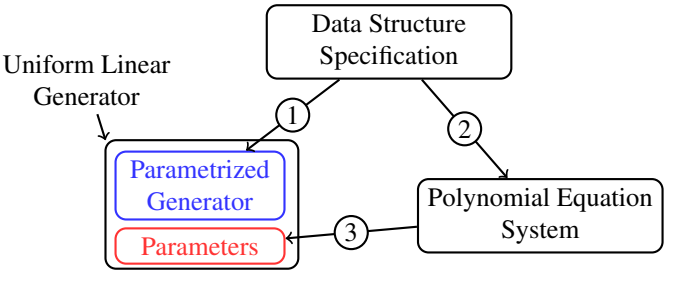


Fig. 1. Boltzmann Method Process Overview

Boltzmann method is generic and can be applied to data whose structure specifications are based on a rich set of constructors, such as disjoint union, Cartesian product, sequences, sets, cycles, etc. It relies on three steps. The first one is the transformation of a `data structure specification` (1) into a `parameterized generator`. The second and third step aim at computing the right parameters for this generator. Step two is the production of a `polynomial equations system` (2) from the `data structure specification` and step three consists in working on the `polynomial equations system` with analytical techniques (these are the domain of analytical combinatorics, described in [5]) in order to compute the actual parameters of the `parameterized generator`, which will make the generator uniform with a linear complexity. By these means, a generator can be automatically compiled from a data structure specification (see figure 1).

This section is continued with a presentation of Boltzmann generation of trees. Section 2.1 presents the notion of *tree specification* which will be used for specifying the structure of the trees to be generated. Section 2.2 describes how to automatically derive the corresponding parametrized generator. Section 2.3 then presents how to compute the actual parameter for making the generator uniform with a linear complexity. Finally, section 2.4 explains why the uniform generator has a linear complexity.

2.1 Tree Specifications

In this paper we use the Boltzmann method to generate trees. A *tree specification* in this context will be a context-free grammar with two terminals (Z and ϵ) and three operators (Seq, $|$ and $*$). Z represents one instancible element (either a leaf or any node) whereas ϵ is the empty element. The unary operator (Seq) is used to specify sequences of an arbitrary size $k \geq 0$. The binary operators ($|$) and ($*$) are used to specify respectively union and product.

The size of a tree T, denoted by $|T|$, will be the number of Z it contains. Remember that for a grammar to be admissible by the Boltzmann method, it must only allow for a finite number of trees of a given size, therefore constructions such as $\mathrm{Seq}(\epsilon)$, which creates an infinity of zero-sized objects, are not allowed. Figure 2 shows three classical examples of tree specifications. First, a binary tree ($\mathcal{T}_1$) which is either a leaf ($\mathcal{L}_1$) or a node ($\mathcal{N}$), with leaves being of one size unit (Z) and nodes being of one size unit that aggregate two binary trees ($Z * \mathcal{T}_1 * \mathcal{T}_1$). Then, one-two tree and a general tree specifications are also given as example.

2.2 General Generator Automatic Construction

In this section we present the transformation that inputs a tree structure specification and returns a corresponding parameterized generator. A parameterized generator is a set of procedures that correspond to each non terminal of the tree structure specification (by convention, we name genT() the procedure that corresponds to the generation of the non terminal $\mathcal{T}$). The parameterized generator can be used to generate any tree that conforms to the input structure specification.

Tree type	A corresponding grammar
Binary trees	$\mathcal{T}_1 = \mathcal{L}_1 \mid \mathcal{N}$
	$\mathcal{L}_1 = Z$
	$\mathcal{N} = Z * \mathcal{T}_1 * \mathcal{T}_1$
One-two trees	$\mathcal{T}_2 = \mathcal{L}_2 \mid \mathcal{U} \mid \mathcal{B}$
	$\mathcal{L}_2 = Z$
	$\mathcal{U} = Z * \mathcal{T}_2$
	$\mathcal{B} = Z * \mathcal{T}_2 * \mathcal{T}_2$
General trees	$\mathcal{T}_3 = Z * \mathrm{Seq}(\mathcal{T}_3)$

Fig. 2. Classical examples of tree specifications

When there is a choice point, for union or sequence in the case of tree specifications, the generator uses its parameters to determine either which element of the union should be generated, and or the length of the sequence to generate. Each choice point must respect a particular *choice probability* in order for the global generator to be uniform. These probabilities are driven by a weight operator, noted w, that will ensure that the generation is uniform. The actual parameters of the generator are the weights of each non-terminal in the specification that, if set correctly, will guarantee the uniformity of the generation and the weight of the terminal Z that, if set correctly, will ensure the linear complexity.

The following rules are then used to build the generation procedures, in addition to each rule we give the corresponding weight operator equation for each construction:

- $A = B$: This construct means that the element A is in fact specified by B. Any generation of A is substituted by the generation of B. $w(A)$ is a generator parameter, and $w(A) = w(B)$.
- $B \mid C$: with this construction, either B or C will be generated. The weights of the elements are used here to control the probability to generate either one or the other. The probability of generating B is $w(B)/(w(B) + w(C))$, and the probability of generating C is symmetric. A pseudo-random number can be used to determine which element should be generated with respect to the given probability. The corresponding weight is $w(B \mid C) = w(B) + w(C)$.
- $\mathrm{Seq}(B)$: this construction independently generates a sequence of B. First the number k of components in the sequence is drawn, following a geometric law ($k = \mathrm{geom}(w(B)) = \lfloor \frac{\ln(random([0,1[))}{\ln(w(B))} \rfloor$), and then k elements of type B are independently generated and returned as a sequence. The weight of such a construction is $w(\mathrm{Seq}(B)) = \frac{1}{1-w(B)}$.
- $B * C$: this construction independently generates both an element B and an element C, and the weight is $w(B * C) = w(B) \cdot w(C)$.
- Z: the Z element in the tree specification corresponds to one tree size unit, which very often corresponds to one node. Wherever there is a Z in the specification, a terminal element is generated. The generation of terminals is not handled by the Boltzmann method, it therefore needs to be provided.

Binary trees: gen$T_1()$ = **if** random() < $w(Z)/w(\mathcal{T}_1)$
then return gen$L_1()$ **else return** gen$N()$
gen$L_1()$ = **return** Ext()
gen$N()$ = **return** Ext(Gen$T_1()$,Gen$T_1()$)

One-two trees: gen$T_2()$ = r := random;
if $r < w(Z)/w(\mathcal{T}_2)$ **then return** gen$L_2()$
elsif $r < w(\mathcal{U})/w(\mathcal{T}_2)$ **then return** gen$U()$
else return gen$B()$
gen$L_2()$ = **return** Ext()
gen$U()$ = **return** Ext(gen$T_2()$)
gen$B()$ = **return** Ext(gen$T_2()$,gen$T_2()$)

General trees: gen$T_3()$ = k := geom($w(\mathcal{T}_3)$); res := [];
for i **from** 1 **to** k **do** res := gen$T_3()$::res **done;**
return Ext(res)

Fig. 3. The generation algorithms for the example grammars

The weight $w(Z)$ is a special parameter of the generator given by the solver (see details below).
- ϵ: ϵ is the empty element, thus nothing will be generated, and $w(\epsilon) = 1$.

Figure 3 shows one generation algorithm for each specification given in figure 2. In this example we name Ext() the constructor of terminals which must be provided by an external source. The explicit values of the weight will be given in section 2.3.

For better understanding of the implementation of such generator, we detail here the construction of the binary tree's generator, the two other ones are given as illustrations.

The specification of binary trees is $\mathcal{T}_1 = \mathcal{L}_1 \mid \mathcal{N}, \mathcal{L}_1 = Z, \mathcal{N} = Z * \mathcal{T}_1 * \mathcal{T}_1$. This recursive specification states that a binary tree is either a leaf ($\mathcal{L}_1$) or a node ($\mathcal{N}$) aggregating two binary trees. The binary tree random generator (noted genT_1) will have to respect the $\mid$ specification; the probability to generate a leaf must be $w(Z)/(w(\mathcal{L}_1) + w(\mathcal{N}))$ (note that as $w(\mathcal{L}_1) + w(\mathcal{N}) = w(\mathcal{T}_1)$, the probability to generate a leaf is simplified as $w(Z)/w(\mathcal{T}_1)$). To generate elements with the right probability we use a pseudo-random generator for real numbers in $]0, 1]$, if the value produced by the pseudo-random generator is smaller than the leaf probability ($w(Z)/w(\mathcal{T}_1)$) a leaf will be produced, otherwise a node is produced using the external binary tree node constructor that inputs two binary trees generated using recursive calls.

Remark 1. If the tree specification has more than one equation, we obtain one generator for each non-terminal, with possible calls to the other non-terminals generators. We thus need to specify one non-terminal as the "root" of the grammar, in order to provide an entry point for the generation.

The goal of the second step of the Boltzmann method is then to compute the actual parameters in order to make the generator uniform with a linear complexity.

2.3 Boltzmann Method

In this section we present how to calculate the weights used by the generator.

Boltzmann method applies to the generation of structured objects, using the powerful tool of *generating functions*. Given a class $\mathcal{C}$ of objects, each object γ having a size denoted by $|\gamma|$, we denote by $C(z)$ its generating function, which is the series $C(z) = \sum_{\gamma \in \mathcal{C}} z^{|\gamma|} = \sum_n c_n z^n$, where c_n is the number of objects of size n in $\mathcal{C}$. In Boltzmann method each object γ is generated with probability $z^{|\gamma|}/C(z)$.

The symbolic method [5] provides a dictionary for translating structural constructions into operators on generating functions: concerning tree constructions, the dictionary reduces to:

$$\begin{array}{rl} \mathcal{Z} \to z, & \epsilon \to 1, \\ \mathcal{C} = \mathcal{A} \mid \mathcal{B} & \to C(z) = A(z) + B(z), \\ \mathcal{C} = \mathcal{A} * \mathcal{B} & \to C(z) = A(z) \cdot B(z), \\ \mathcal{C} = \mathrm{Seq}(\mathcal{A}) & \to C(z) = \frac{1}{1-A(z)}. \end{array}$$

Thus in a tree specification, each line transforms into a corresponding *generating function* equation (which also corresponds to the weight relations from section 2.2), and a system of specifications transforms into a polynomial system of equations.

For computing weights as described in section 2.2, we need to solve such systems of equations for a given value x of variable z: the weight of element Z is set to x, and the weight of a non-terminal C is the value of series $C(z)$ evaluated at $z = x$. The resolution is analytically coherent for $0 \leq x \leq \rho$, where ρ is a special value, called the *singularity* of the system.

Solving polynomial systems of equations is a very complex problem in general, but systems corresponding to specifications do have a structure that can be exploited in the computations. In our implementation we use a combinatorial newton method that gives a very efficient solver [12], that can also be used to calculate an approximation of the singularity ρ.

In figure 4, we show the generating functions for the previously introduced tree specifications and the calculated weights for each of these systems for $z = \rho$.

In each case, using the values of these functions at $z = \rho$, the Boltzmann algorithms of 2.2 derive a linear time generator with the property of *uniformity*: given a size n, two trees of that size have exactly the same probability of being generated. These generators however have the particularity that the generated trees are not all of size n, but have a random size, with a mean value depending on parameter z. We show in section 2.4 how to deal with this aspect, using ρ as the value for z.

Tree type	Corresponding generating functions	Weights
Binary trees	$T_1(z) = L_1(z) + N(z)$ $L_1(z) = z$ $N(z) = z \cdot T_1(z)^2$	$w(Z) = \rho = 1/2$ $w(T_1) = 1$ $w(L_1) = 1/2$ $w(N) = 1/2$
One-two trees	$T_2(z) = L_2(z) + U(z) + B(z)$ $L_2(z) = z$ $U(z) = z \cdot T_2(z)$ $B(z) = z \cdot T_2(z)^2$	$w(Z) = \rho = 1/3$ $w(T_2) = 1$ $w(L_2) = 1/3$ $w(U) = 1/3$ $w(B) = 1/3$
General trees	$T_3(z) = z \cdot \frac{1}{1-T_3(z)}$	$w(Z) = \rho = 1/4$ $w(T_3) = 1/2$

Fig. 4. The generating functions and calculated weights of the example grammars

2.4 Complexity and Generation of Huge Trees

With Boltzmann method, the size of the generated trees is random, with a distribution that depends on the specification and a mean value that goes from 0 to infinity when parameter x goes from 0 to ρ. More precisely the probability for the result to be of size n depends on parameter x and on the singularity ρ, which is attached to the system of equations corresponding to the specification: for large n, this probability is proportional to $n^{-\frac{3}{2}} x^n \rho^{-n}$. Thus the closest is x to the value of ρ, the biggest is the probability of generating large size trees.

As an illustration, in figure 5 we plot the probability of producing a tree of size n in function of n, with different values of x: there are five different curves, corresponding to $x = 0.9\rho, x = 0.999\rho, x = 0.99999\rho, x = 0.9999999999\rho$ and $x = \rho$. In the right part, the curves are plotted with both axes in logarithmic scale, in order to show up the differences. It is quasi-impossible to obtain a tree of size one hundred with a precision of $1/10$ for x/ρ, whereas it is likely to produce a tree of size ten million when ρ is approximated with a precision of $1/10^{10}$.

Boltzmann samplers are particularly efficient if we accept some variability in the size of the generated structures: fixing a target size n and a margin of error δ, generating a structure of size belonging to $[(1-\delta)n, (1+\delta)n]$ can be completed in mean time $O(n)$ (whereas exact size average complexity can be up to quadratic).

In [2], it is showed that in the case of tree sampling, linear time complexity can be achieved by Boltzmann method by using either *pointing* or *singular sampling*. For our implementation, we chose the second approach, consisting in taking ρ as the value of x, and this leads to both issues of computing ρ and rejecting trees of non admissible size. Indeed in the case of singular sampling, the mean size of the generated structures is infinite. We will never generate an infinite object, but there is however a non-trivial probability of generating objects of sizes that we cannot handle. The solution to this problem is simple and consists in aborting the generating process as soon as we pass the upper bound of our target size.

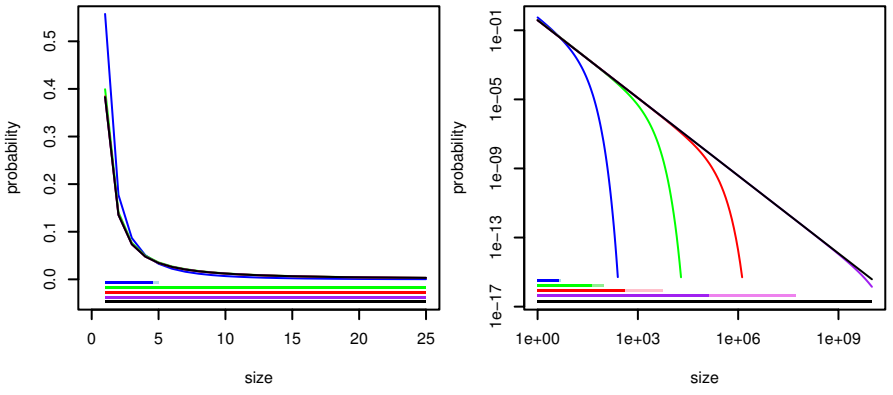


Fig. 5. Probability distribution of sizes for trees generated with Boltzmann method, with a parameter $x = 0.9\rho, 0.999\rho, 0.99999\rho, 0.9999999999\rho$, and ρ. The solid color bars show the range inside which the generators have a guaranteed linear complexity, which in practice extends to the whole colored range. In the second plot, both axes are in logarithmic scale.

As for the second point, the evaluation of ρ is non trivial and will not be detailed in this paper; it uses a dichotomy heuristic and the Newton algorithm of [12].

3 Model Generation Based on Meta Model Specification

We present in this section a scalable, uniform, random model generation process. This process can be used to generate very big models based on their metamodel specification. It makes use of the uniform random trees generation previously presented. We present here how to transform a metamodel specification into a tree specification and how to complete the generated trees to obtain the final instance.

3.1 Running Example

In figure 6 is presented a simple MOF/ECORE [10] metamodel inspired from the classic ECLIPSE EMF[6] Library example. This metamodel will be used throughout this paper to illustrate our approach. It contains four meta-classes: Library, Book, Volume and Compilation where Volume and Compilation inherit the Book abstract meta-class. This metamodel shows three containment relations, from Library to Book, fro Library to Writer and from Compilation to Book, and one relation from book to writer.

3.2 From Metamodels to Tree Specification

Metamodels and tree specifications are not equivalent. The metamodel language is far more expressive than tree specifications. We present here how to interpret

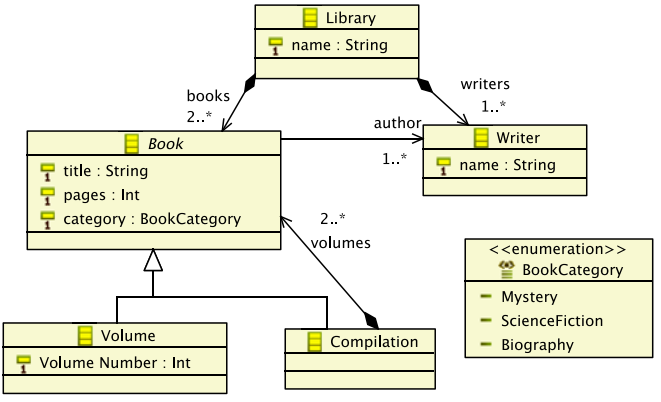


Fig. 6. Running example metamodel diagram

metamodel constructions into tree specification constructions. The transformation we propose is done in three steps where each step refines the output specification tree. Note that the model transformation we define here is not total, some elements in the metamodel will not be translated into the corresponding tree specification. Our approach only generates the core structure of the model.

Identification of base trees. The first step to build the random generator is to identify parts of the metamodel that will correspond to the randomly generated tree. A metamodel is a directed graph that is used to specify other graphs. We need to identify trees in the metamodel graph to be able to generate trees that respect the metamodel specifications. The trees used by the random generator are identified thanks to the containment relationships in the metamodel. The containment relationship offers two advantages, they allow to hierarchically generate the model and are acyclic. For each containment relationship found in the metamodel both source and target meta-classes are created in the tree specification and are equal to Z, i.e. an element with one size unit. Abstract meta-classes are created but are not equal to anything at this stage as they should not be instantiated. Finally, each of the containment relationships adds that source equals its value times the target. In the running example we identified three containment relationships. The result of the transformation on the running example is this tree specification:

$$\begin{aligned} Library &= Z \ * \ Book \ * \ Writer \\ Book &= void \\ Writer &= Z \\ Compilation &= Z \ * \ Book \end{aligned}$$

Inheritance relations. The second step to obtain the tree specification is to handle inheritance relations. The inheritance relation is interpreted as a logical

or. If meta-class B inherits A, the generation of an instance of A can be replaced by the generation of an instance of B. Therefore, are added to the specification rules for each meta-class A that has daughter meta-class B the fact that A is $A \vee B$. If a new meta-class is encountered, it is equals Z. At the end of this stage all tree specification entries must have a value, all abstract meta-classes that are not inherited must be removed from the tree specification system as they can not be instantiated.

In the running example, *Compilation* and *Volume* inherit *Book*. The resulting tree specification is :

$$\begin{aligned} Library &= Z \ * \ Book \ * \ Writer \\ Book &= Volume \mid Compilation \\ Writer &= Z \\ Volume &= Z \\ Compilation &= Z \ * \ Book \end{aligned}$$

Cardinalities. The third step makes sure the cardinalities constraints are respected. To respect lower and upper bound cardinalities the target is multiplied as many times as requested in the tree specification, if the cardinality of a relation A to B is $x..y$ then A is $\vee_{i=x}^{y} B^i$. If the upper-bound is $*$ we use the sequence concept, where $Seq(B)$ denotes an arbitrary long B sequence, note that a sequence may be empty. If the lower bound is 0, the empty element ϵ is used.

If we apply the cardinality constraints to our tree specification, we obtain :

$$\begin{aligned} Library &= Z \ * \ Book^2 \ * \ Seq(Book) * \ Writer \ * \ Seq(Writer) \\ Book &= Volume \mid Compilation \\ Writer &= Z \\ Volume &= Z \\ Compilation &= Z \ * \ Book^2 \ * \ Seq(Book) \end{aligned}$$

Unadapted metamodels. At the end of the transformation, all meta-classes in the metamodel should have a value in the tree specification. If it is not the case, this meta-classes are not accessible, meaning that they can not be randomly generated in a global uniform random generation process, i.e. the metamodel is not suited for our random generation process.

At the end of the transformation, all meta-classes should be linked directly or indirectly with the root of the metamodel. If it is not the case, the given metamodel has more than one root, our random generation process can be used with any of this roots, but only a subpart of the metamodel will be generated.

3.3 Model Final Structure Generation

The tree specification corresponding to the metamodel is used to generate the skeleton of the instance generation as described in section 2, however the Boltz-

mann tree random generator only generates a model core. It needs a mechanism to generate basic relations that are not containments in order to generate instances of the metamodel. To our knowledge, there is no methodology to uniformly generate such structures and keep the overall generation process random (Boltzmann model does not apply well to graphs). Therefore, any generation process for the basic relations that respects the metamodel specification can be used to complete the skeleton. For instance, in [3] is described as stage two and three such a process. In our implementation on UML Class models we implemented a generator that strictly satisfies the lower bound constraints.

4 Validation

In this section we present our implementation of the generator, as well as particularities of the random sampling that were verified in particle uses when generating UML 2.2 Class Models.

4.1 Implementation

We present in this section the application of the generation process to UML class models. From the official UML 2.2 specification we extracted a simplified class metamodel. The figure 7 presents the tree specification corresponding to this metamodel. Note that the generation processed has been tweaked, the probability to generate packages was reduced and the probability to generate operations, properties, literal integers and parameters augmented. This manipulation is later detailed in this section.

We implemented the generator as an Eclipse Plugin which is available online[2]. The plugin can generate UML files containing the class model from a graphical interface shown in figure 8.

We implemented a value generator for the names, literal values, visibility kinds and direction kind in order to produce a valid model. The properties value generation we implemented is constrained in order to only produce valid models. We also implemented a generator for generalization and references that randomly chose a valid target in the generated elements. However, the generation of constrained values is not in the scope of this paper.

Even with a linear complexity in random calls, the actual generation process may be long for big models. Indeed, the meta-classes instantiations and properties value generation is time consuming, in order to avoid the generation of unused elements we propose to use a simulation. When a model of a particular size is to be generated, the current random seed is saved, the algorithm to generate a random instance is run without any instantiation, and if the simulation is successful (the size of the model is correct), the seed is reused to generate the actual model, otherwise the simulation is run again. This optimization does not change the theoretical complexity of the sampling but allows to gain a huge amount of time. In our implementation on UML Class models, there is a factor

[2] See http://meta.lip6.fr for more details.

$$
\begin{aligned}
model &= package \\
package &= 0,01Z * Seq(packageableElement) \\
packageableElement &= package \mid class \mid association \\
class &= Z * Seq(property) * Seq(operation) * Seq(generalization) \\
generalization &= Z \\
property &= 3Z * (valueSpecification \mid \epsilon) \\
association &= Z \\
valueSpecification &= literalBoolean \mid literalNull \mid literalInteger \mid literalString \\
literalBoolean &= Z \\
literalNull &= Z \\
literalInteger &= 2Z \\
literalString &= Z \\
operation &= 2Z * Seq(parameter) \\
parameter &= 3Z * (valueSpecification \mid \epsilon)
\end{aligned}
$$

Fig. 7. UML 2.2 based tree specification for class models

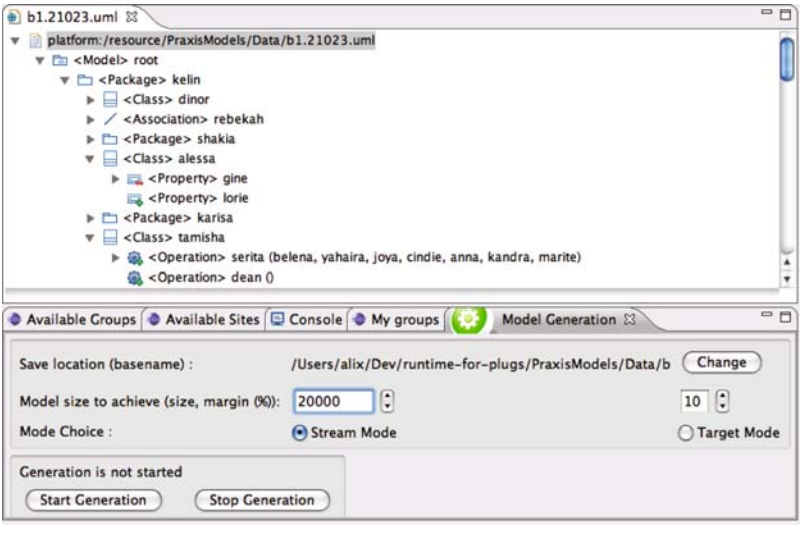


Fig. 8. snapshot of the class model generator

higher than 1000 between the simulation of a valid generation and the same calculation with all meta-classes instantiations.

We present in figure 9 the performances of our prototype. This chart shows that the time to obtain a seed that will produce a valid model (valid size) is very short and its average is linear. However our implementation based on EMF [6] is quite slow and can not reasonably produce models above a size of 250 000 model

Model size (10% margin)	Average simulation time	Building time
100	6.50 ms	44.6 ms
1 000	11.2 ms	154 ms
10 000	91.1 ms	1.61 s
50 000	0.501 s	9.87 s
100 000	0.934 s	26.0 s
200 000	1.79 s	52.8 s
250 000	2.48 s	63.2 s
500 000	4.32 s	not applicable
1 000 000	8.86 s	not applicable

Fig. 9. Prototype's performance chart ran on a MacBook Air

elements due to a huge memory consumption. Therefore we can not provide valid building times for the size 500 000 and one million. It is important to note that the time we provide for obtaining a valid seed is an average for one hundred runs. As the complexity is an mean time complexity, the actual time spent to find a valid seed can significantly vary.

4.2 Generating Instances of a Particular Size

The presented theory states that the mean time to generate a model of a particular size, within a reasonable margin, is linear. The probability to obtain a model of the right size allows us to randomly generate models and only keep the ones with a valid size. As the complexity to generate one model of the wanted size is linear, the complexity to generate a fixed size sampling of uniform models with a size in $[n(1-\delta), n(1+\delta)]$ has a linear complexity too.

It has to be noted, however, that the Boltzmann sampler is very sensible to the value of its parameter, particularly as it approaches its maximal value ρ. Our method depends on taking a parameter equal to ρ, but as ρ can be any real number between 0 and 1, it is not possible to calculate it exactly in most cases and we will use an approximation. The effect of this, illustrated in figure 10, is a "ceiling" in the maximal size attainable by the generator. It is therefore important to make a very precise approximation of ρ to be able to generate very large objects. In figure 10 is represented the distribution of class model sizes generated using different approximations of ρ, it appears that the proportion of big models is affected by the accuracy of ρ's calculation. For instance no model of a size greater than five hundred thousand model elements could be generated with a value of ρ correct up to the seventh digit, but with a precision of fifteen digits, ten million model element is possible to reach in linear time.

4.3 Influencing Generation Output

It is in our opinion very important to be able to characterize the probability distribution of the generated structures, as it allows us to control a possible

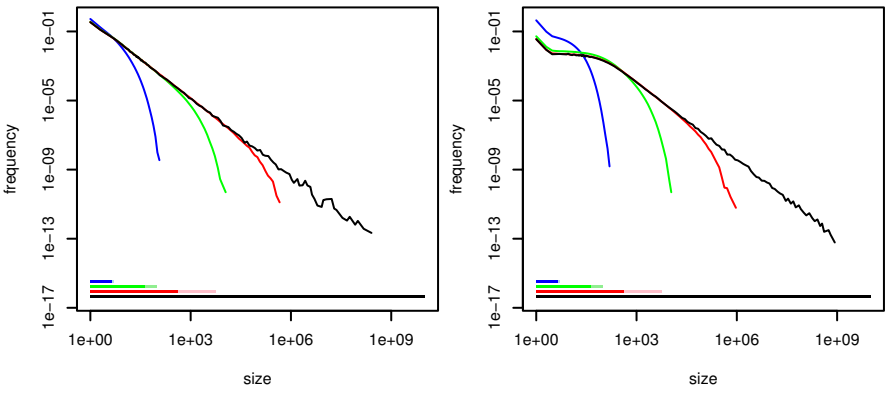


Fig. 10. Distribution of sizes of generated class models by a Boltzmann sampler with a parameter of $0.9\rho, 0.999\rho, 0.99999\rho$ and ρ (ρ calculated with a 10^{-15} precision). The left plot corresponds to a grammar without coefficients, while in the right the exact grammar of figure 7 is used. Note that both axes are in logarithmic scale.

bias. However, the uniform distribution provided by the Boltzmann samplers may not be the best fit to this needs. We might find, for example, that the number of operations in classes in the random class models is not sufficient. We thus need to add ponderations in our specification, in order to influence the frequency of appearance of the different elements, in a way that allows us to calculate the resulting bias.

The extension of the theory of Boltzmann sampling in this direction is work in progress, but there are some elements that we can already use. We allow the definition, for each non-terminal, of a coefficient with a default value of 1 that will influence the frequency of the corresponding element. The influence of the coefficient values to the frequencies is not trivial, as the different frequencies depend on each other, so the good choice of coefficients is done for the moment via trial and error. It is possible to calculate the frequencies given the values of the coefficients, and the complexity of the generation process is not affected. In figure 7 where the tree specification of simplified UML class diagrams is given, the probabilities have been modified, the probability to generate packages was reduced and the probability to generate operations, properties, literal integers and parameters augmented in order to obtain more *realistic* class models.

5 Related Works

Alloy which is a lightweight specification language based on first-order relational logic [7] can be used to generated models. Indeed, its main principle is to compute all models of a fixed size and that correspond to a particular specification. Then Alloy is able of extracting form this set of models the ones that are consistent regarding to a set of specified constraints. Alloy is based on a SAT solver

(the SAT problem belongs to the NP-Hard class) and therefore is not able to produce huge models.

In [3], is presented an algorithm that can generate instances of metamodels. It is based on a transformation of the metamodel structure into a set of graph specification rules. This set of rules is able to generate any skeleton of metamodel instances, and can be coupled with constraint rules in order to respect specific needs. The random process resides in the random election of generation rules. The outputted models can be biased as the choosing of the rule is constrained by the graph specification rule application formalism. Plus, this approach may not scale, the applying of each graph rule has an exponential complexity as it needs to find the existence of a subgraph in the already generated graph which limits the efficiency of this tool (the general problem is NP-Hard).

In [1], a formalism is presented to generate random constrained models. The approach consists in using mutations to derive, from a given instance, random other alike instances. The approach is effective and can handle very huge models since the mutation process is very effective. However, this approach is biased by definition, it needs to input one instance of the model to generate others, therefore the outputted models will have a lot of similarities.

6 Conclusion

In this paper we presented an adaptation of the Boltzmann random sampling theory to metamodel instance generation. The resulting generator has three interesting particularities.

First it is scalable, the complexity of the generating process is linear with the size of the generated structures. And this size is controllable.

Then it outputs uniform samplings for a given size, the probability for any structure of size n to be generated is the same.

And finally, it allows to experimentally change the form of outputted models to meet with specific requirements.

However, metamodels usually come with a set of constraints to precise the specification of its instances, in this paper we did not describe how to generate values for the properties of these instances, however our implementation on class models successfully took this challenge in consideration. In the particular purpose of generating models that satisfy important model constraints, the property generation must be carefully controlled, and possibly a random generation process may not be adapted. Further research in this direction must be done in order to exploit the high performances of the random generation of metamodel instances to constrained models.

References

1. Brottier, E., Fleurey, F., Steel, J., Baudry, B., Traon, Y.L.: Metamodel-based test generation for model transformations: an algorithm and a tool. In: 17th International Symposium on Software Reliability Engineering, 2006. ISSRE 2006, pp. 85–94 (2006)

2. Duchon, P., Flajolet, P., Louchard, G., Schaeffer, G.: Boltzmann samplers for the random generation of combinatorial structures. Combinatorics, Probability and Computing 13, 577–625 (2004)
3. Ehrig, K., Kuster, J., Taentzer, G., Winkelmann, J.: Generating instance models from meta models. In: Gorrieri, R., Wehrheim, H. (eds.) FMOODS 2006. LNCS, vol. 4037, pp. 156–170. Springer, Heidelberg (2006)
4. Feiler, P., Gabriel, R., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Northrop, L., Schmidt, D., Sullivan, K., et al.: Ultra-large-scale systems: The software challenge of the future. Technical report, Software Engineering Institute, Carnegie Mellon University (2006) ISBN 0-9786956-0-7
5. Flajolet, P., Sedgewick, R.: Analytic Combinatorics. Cambridge University Press, Cambridge (2009)
6. T. E. Fondation. EMF (Eclipse Modeling Framework), http://www.eclipse.org/modeling/emf/
7. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. The MIT Press, Cambridge (2006)
8. Lucrédio, D., de Mattos Fortes, R.P., Whittle, J.: Moogle: A model search engine. In: Proceedings of Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, pp. 296–310 (2008)
9. Mellor, S.J., Clark, A.N., Futagami, T.: Guest editors' introduction: Model-driven development. IEEE Software 20(5), 14–18 (2003)
10. OMG. Meta Object Facility (MOF) 2.0 Core Specification (January 2006)
11. Pivoteau, C.: Génération aléatoire de structures combinatoires: méthode de Boltzmann effective. Ph.D thesis, UPMC (2008)
12. Pivoteau, C., Salvy, B., Soria, M.: Boltzmann oracle for combinatorial systems. In: Fifth Colloquium on Mathematics and Computer Science Algorithms, Trees, Combinatorics and Probabilities, DMTCS Proceedings, pp. 475–488 (2008)
13. Selic, B.: The pragmatics of model-driven development. IEEE Software 20(5), 19–25 (2003)

Establishing Correspondences between Models with the Epsilon Comparison Language

Dimitrios S. Kolovos

Department of Computer Science, University of York,
Heslington, York, YO10 5DD, UK
dkolovos@cs.york.ac.uk

Abstract. Model comparison is an essential prerequisite for a number of model management tasks in Model Driven Engineering, such as model differencing and versioning, model and aspect merging and model transformation testing. In this paper we present the Epsilon Comparison Language (ECL), a hybrid rule-based language, built atop the Epsilon platform, which enables developers to implement comparison algorithms at a high level of abstraction and execute them in order to identify matches between elements belonging to models of diverse metamodels and modelling technologies.

1 Introduction

Model comparison is the task of identifying *matching* elements between models. In general, matching elements are elements that are involved in a relationship of interest. For example, before calculating the differences between two models, it is necessary to identify the pairs of corresponding elements in the two models. Similarly, before merging homogeneous models, it is essential that matching elements are identified so that they do not appear in duplicate in the merged model. Finally, in model transformation testing, pairs that consist of elements in the input model and their generated counterparts in the output model need to be identified.

Several approaches to model matching have been proposed in the literature including id-based approaches, which establish matches based on the common non-volatile identity of model elements, signature-based approaches in which comparison between elements is achieved via comparison of their respective signatures, and similarity-based approaches which treat models as typed attributed graphs and compare their elements based on the (often adjusted by user-defined weights) similarity of their features.

As discussed in [1], the main advantage of similarity-based approaches, such as SiDiff [2] and DSMDiff [3], is that they can – relatively – easily be configured to support new modelling languages. However, they also demonstrate two significant shortcomings: they provide limited support – beyond feature-weight setting parameters – for exploiting the semantics and particularities of each individual modelling language in order to improve accuracy and performance, and they cannot be applied to heterogeneous models.

To address these limitations, in this paper we present the Epsilon Comparison Language (ECL), a task-specific model management language built atop the Epsilon Eclipse

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, pp. 146–157, 2009.

GMT component [4], which enables developers to specify precise language-specific algorithms for establishing matches between elements belonging to models of diverse metamodels and technologies in a rule-based manner, and at a high level of abstraction. The rest of the paper is organized as follows. In Section 2 we provide an overview of the background and related work in the field of model comparison. In Section 3 we provide a brief discussion on Epsilon, the platform that underpins the proposed ECL language. Section 4 presents the abstract, concrete syntax and execution semantics of ECL. In Section 5 we presents an example of using ECL to compare UML2 models, and in Section 6 we conclude and provide directions to further work.

2 Background and Motivation

Establishing correspondences between elements belonging to different models in an automated manner is an essential prerequisite for a number of model management tasks such as model differencing, transformation testing and homogeneous and heterogeneous model merging. Given the structured nature of models captured with contemporary metamodelling architectures such as MOF [5] and EMF [6], traditional text-based comparison and differencing algorithms have been shown to be insufficient for model comparison [2]. Algorithms based on persistent unique identifiers, such as [7], are fast but they cannot be applied to modelling technologies that do not support such identifiers or to individually developed models. A signature-based comparison approach is presented in [8] where instead of using persistent identities, the identities used for comparison are instead computed from the features of each element using the Kermeta [9] language. Although this approach addresses the two aforementioned issues to an extent, string-based ids are not meaningful for all types of model elements, as the authors admit.

Another category of approaches, such as SiDiff [2] and DMSDiff [3], treat models as typed attribute graphs and calculate the similarity of nodes based on the combined weighted similarity of their features; this is done in a recursive manner. Such algorithms are typically configurable with regard to the weight assigned to each feature. This is a direct way to indicate the significance of a feature in the context of a comparison. The main advantage of such algorithms is that they are generic and easy to adapt to new metamodels as the customization process typically involves specifying the weights of the features in the new metamodel. However, due to their generic nature, they generally cannot exploit more fine-grained semantics of the metamodel to reduce the number of required comparisons, or to enhance the precision of the results.

In [10], the ATL model-to-model transformation language [11] is used to perform comparison of two models. While this work demonstrates the feasibility of using a general-purpose M2M language for comparing models, comparison transformations are generally verbose[1], as they need to compensate for the fact that M2M languages do not provide tailored constructs for the task of model comparison.

[1] For example, in `http://www.eclipse.org/gmt/amw/examples/library/CreateWeaving.atl`, a significant part of the transformation is dedicated to low-level tasks such as identifying which of the two input models a model element originates from.

3 The Epsilon Platform

Epsilon [4] is a platform that provides infrastructure for implementing uniform, integrated and interoperable model management languages that can be used with models of diverse metamodels and technologies such as EMF and MDR [12]. At the core of Epsilon is the Epsilon Object Language (EOL) [13], an OCL-based imperative language that provides additional features such as model modification, multiple model access, conventional programming constructs (variables, loops, branches etc.), user interaction, profiling, and support for transactions. While EOL can be used as a general-purpose model management language, its primary aim is to be reused for constructing task-specific languages, and therefore it provides respective extensibility points, discussed in [13], that enable developers to easily implement the additional requirements of specific tasks such as model-to-model transformation or model comparison – as discussed in this paper – atop it.

4 The Epsilon Comparison Language

The aim of the Epsilon Comparison Language (ECL) is to enable users to specify precise model comparison algorithms in a high-level rule-based manner, and execute them in order to identify pairs of matching elements between different models. ECL has been implemented as an extension of EOL and therefore it inherits all its syntax and features (e.g. accessing multiple models of diverse metamodels simultaneously, invoking native (Java) code). In this section, the abstract and concrete syntax, as well as the execution semantics of the language, are discussed in detail.

4.1 Abstract Syntax

In ECL, comparison specifications are organized in modules (*ECLModule*). As illustrated in Figure 1, by extending the core EOL library module construct, an ECL module can contain user-defined operations and import other library and ECL modules. In addition to operations, an ECL module also contains a set of match-rules (*MatchRule*) and a set of *pre* and *post* blocks.

MatchRules enable users to perform comparison of model elements at a high level of abstraction. Each match-rule declares a name, and two parameters (*leftParameter* and *rightParameter*) that specify the types of elements it can compare. It also optionally defines a number of rules it inherits (*extends*) and if it is *abstract*, *lazy* and/or *greedy*. The semantics of the latter are discussed in the sequel.

A match rule has three parts. The *guard* part is an EOL expression or statement block that further limits the applicability of the rule to specific elements. The *compare* part is an EOL expression or statement block that is responsible for comparing a pair of elements and deciding if they match or not. Finally, the *do* part is a block of EOL statements that is executed if the *compare* part returns true, in order to perform any additional actions required.

Pre and *Post* blocks are named blocks of EOL statements which as discussed in the sequel are executed before and after the match-rules have been executed respectively.

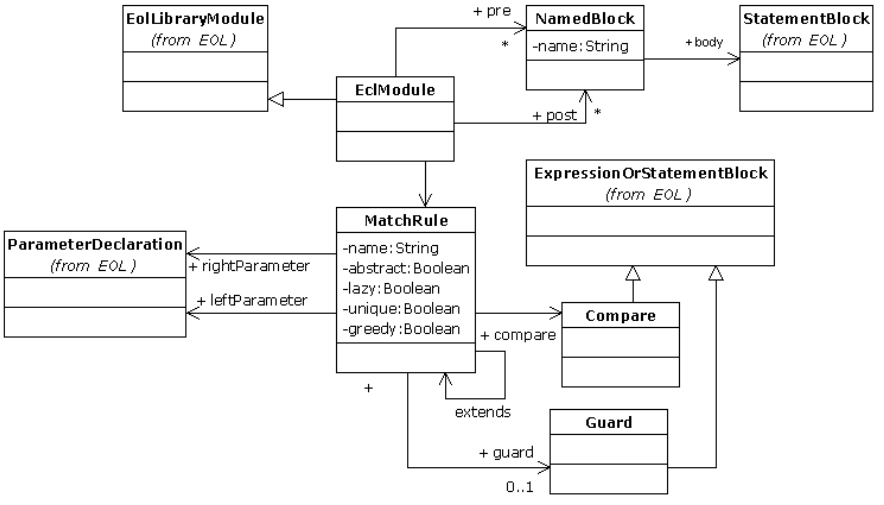


Fig. 1. ECL Abstract Syntax

4.2 Concrete Syntax

The concrete syntax of a match-rule is displayed in Listing 1.1.

Listing 1.1. Concrete Syntax of a MatchRule

```
1  (@lazy)?
2  (@abstract)?
3  (@greedy)?
4  rule <name>
5     match <leftParameterName>:<leftParameterType>
6     with <rightParameterName>:<rightParameterType>
7     (extends (<ruleName>,)*<ruleName>)? {
8
9     (guard (:expression)|({statementBlock}))?
10
11    compare (:expression)|({statementBlock})
12
13    (do {statementBlock})?
14
15 }
```

Pre and *post* blocks have a simple syntax that, as presented in Listing 1.2, consists of the identifier (pre or post), an optional name and the set of statements to be executed enclosed in curly braces.

Listing 1.2. Concrete Syntax of Pre and Post blocks

```
1  (pre|post) <name> {
2     statement+
3  }
```

4.3 Execution Semantics

Rule and Block Overriding. As mentioned above, an ECL module can import a number of other ECL modules. In this case, the importing ECL module inherits all the rules and pre/post blocks specified in the modules it imports (recursively). If the module specifies a rule or a pre/post block with the same name, the local rule/block overrides the imported one respectively. This feature enables developers to reuse and customize the behaviour of existing comparison modules with minimal duplication.

Comparison Outcome. As illustrated in Figure 2, the result of comparing two models with ECL is a trace (*MatchTrace*) that consists of a number of matches (*Match*). Each match holds a reference to the model elements that have been compared (*left* and *right*), a boolean value that indicates if they have been found to be *matching* or not, a reference to the *rule* that has made the decision, and a Map (*info*) that is used to hold any additional information required by the user.

During the matching process, a second, temporary, match trace is also used to detect and resolve cyclic invocation of match-rules as discussed in the sequel.

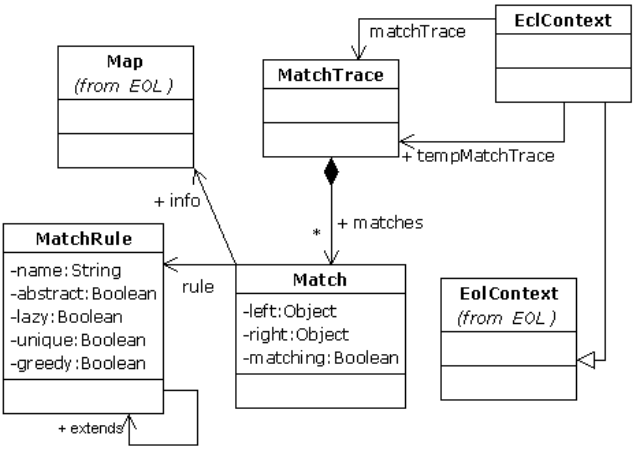


Fig. 2. ECL Match Trace

4.4 Rule Execution Scheduling

Non-abstract, non-lazy match-rules are evaluated automatically by the execution engine in a top-down fashion - with respect to their order of appearance - in two passes. In the first pass, each rule is evaluated for all the pairs of instances in the two models that have a type-of relationship with the types specified by the *leftParameter* and *rightParameter* of the rule. In the second pass, each rule that is marked as *greedy* is executed for all pairs that have not been compared in the first pass, and which have a kind-of relationship with the types specified by the parameters of the rule. In both passes, to evaluate the *compare* part of the rule, the *guard* must be first satisfied.

Before the *compare* part of a rule is executed, the *compare* parts of all of the rules it extends (super-rules) must be executed (recursively). Before executing the *compare* part of a super-rule, the engine verifies that the super-rule is actually applicable to the elements under comparison by checking for type conformance and evaluating the *guard* part of the super-rule.

If the *compare* part of a rule evaluates to true, the optional *do* part is executed. In the *do* part the user can specify any actions that need to be performed for the identified matching elements, such as to populate the *info* map of the established *match* with additional information or explicitly trigger the comparison of additional elements. Finally, a new match is added to the match trace that has its *matching* property set to the logical conjunction of the results of the evaluation of the *compare* parts of the rule and its super-rules.

4.5 The Matches() and doMatch() Built-in Operations

To refrain from performing duplicate comparisons and to de-couple match-rules from each other, ECL provides the built-in *matches(opposite : Any)* and *doMatch(opposite : Any)* operations for model elements and collections.

When the *matches()* operation is invoked on a pair of objects, it queries the main and temporary match-traces to discover if the two elements have already been matched and if so it returns the cached result of the comparison. Otherwise, it attempts to find an appropriate match rule to compare the two elements and if such a rule is found, it returns the result of the comparison, otherwise it returns false. Unlike the top-level execution scheme, the *matches()* operation invokes both *lazy* and *non-lazy* rules.

In addition to objects, the *matches* operations can also be invoked to match pairs of collections of the same type (e.g. a *Sequence* against a *Sequence*). When invoked on ordered collections (i.e. *Sequence* and *OrderedSet*), it examines if the collections have the same size and each item of the source collection matches with the item of the same index in the target collection. Finally, when invoked on unordered collections (i.e. *Bag* and *Set*), it examines if for each item in the source collection, there is a matching item in the target collection irrespective of its index. Users can also override the built-in *matches* operation using user-defined operations [13] with the same name, that loosen or strengthen the built-in semantics for particular model element types.

Of similar functionality is the built-in *doMatch(opposite : Any)* operation with the difference that the results of comparisons triggered by *doMatch* are stored in the maintained match trace, while the results of comparisons triggered by *match* are disregarded. Also, unlike the *match* operation that returns a boolean, *doMatch* is void.

Cyclic invocation of matches(). The built-in *matches* operation significantly simplifies comparison specifications. It also enhances decoupling between match-rules from each other since when a rule needs to compare two elements that are outside its scope, it does not need to know/specify which other rule can compare those elements explicitly.

On the other hand, it is possible - and quite common indeed - for two rules to implicitly invoke each other. For example consider the match rule of Listing 1.3 that attempts to match nodes of the simple Tree metamodel displayed in Figure 3.

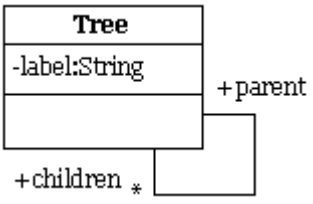


Fig. 3. The Tree Metamodel

Listing 1.3. The Tree2Tree rule

```
1  rule Tree2Tree
2     match l : T1!Tree
3     with r : T2!Tree {
4
5     compare : l.label = r.label and
6        l.parent.matches(r.parent) and
7        l.children.matches(r.children)
8  }
```

The rule specifies that for two nodes to match, they should have the same label, belong to matching parents and have matching children. In the absence of a mechanism for cycle detection and resolution, the rule would end up in an infinite loop. To address this problem, ECL provides a temporary match-trace which is used to detect and resolve cyclic invocations of the match() built-in operation.

As discussed above, a match is added to the primary match-trace as soon as the compare part of the rule has been executed. By contrast, a temporary match (with its *matching* property set to *true*) is added to the temporary trace before the compare part is executed. In this way, any subsequent attempts to match the two elements from invoked rules will not re-invoke the rule. Finally, when a top-level rule returns, the temporary match trace is reset.

4.6 Fuzzy and Dictionary-Based String Matching

In the example of Listing 1.3, the rule specifies that to match, two trees must - among others - have the same label. However, there are cases when a more relaxed approach to matching string properties of model elements is desired. For instance, when comparing two UML models originating from different organizations, it is common to encounter ontologically equivalent classes which however can have different names (e.g. Client and Customer). In this case, to achieve a more sound matching, the use of a dictionary or a lexical database, such as WordNet, is necessary. Alternatively, fuzzy string matching algorithms such as those presented in [14] can be used.

As several such tools and algorithms have been implemented in various programming languages, it is of our best interest to reuse them instead of re-implementing them. For example, in Listing 1.4 we use a wrapper for the Simmetrics Java tool to compare the labels of the trees using the Levenstein algorithm. To achieve this, line 11 invokes the *fuzzyMatch()* operation defined in lines 16-18 which uses the simmterics native tool

(instantiated in lines 2-4) to match the two labels using their Levenshtein [15] distance with a threshold of 0.5. The ability of ECL to invoke native (Java) code is inherited from the core EOL language which underpins it.

Listing 1.4. The FuzzyTree2Tree rule

```
  languagelanguage
1 pre {
2    var simmetrics :=
3       new Native('org.epsilon.ecl.tools.
4          textcomparison.simmetrics.SimMetricsTool');
5 }
6
7 rule FuzzyTree2Tree
8    match l : T1!Tree
9    with r : T2!Tree {
10
11   compare : l.label.fuzzyMatch(r.label) and
12      l.parent.matches(r.parent) and
13      l.children.matches(r.children)
14 }
15
16 operation String fuzzyMatch(other : String) : Boolean {
17    return simmetrics.
18       similarity(self,other,'Levenshtein') > 0.5;
19 }
```

4.7 Exploiting the Comparison Outcome

Developers can exploit the match trace calculated during the comparison process in the post sections of the module or export it into another application or Epsilon program. For example, in a post section, the trace (which is programmatically accessible via the *matchTrace* built-in variable) can be printed to the default output stream or serialized into a model of an arbitrary metamodel. In another use case, the trace may be exported to be used in the context of a validation module that will use the identified matches to evaluate inter-model constraints, or in a merging module that will use the matches to identify the elements on which the two models will be merged.

We have also provided integration with the EMF Compare framework, which can perform comparison of EMF-based models and visualize the results within Eclipse. Due to the modular architecture of EMF Compare, we were able to provide an additional matching engine that executes a user-defined ECL comparison module and visualize its results using the intuitive EMF Compare user interface.

5 Example

In this section we demonstrate an excerpt of an ECL module that can compare UML 2.1 class models. Our aim in this example is not to provide the complete comparison but to use exemplar rules from the module to present the features and advantages of the proposed language.

Listing 1.5. Fragment of UML2 comparison

```
1  rule Class
2     match l : Left!Class
3     with r : Right!Class {
4
5     compare : l.name = r.name
6
7     do {
8        l.ownedOperation.doMatch(r.ownedOperation);
9     }
10 }
11
12 @lazy
13 rule Operation
14    match l : Left!Operation
15    with r : Right!Operation {
16
17    compare {
18
19       var basicMatch := l.name = r.name;
20
21       if (basicMatch) {
22          if (l.class.hasOnlyOneOp(l.name) and
23             r.class.hasOnlyOneOp(l.name)) {
24
25             return true;
26          }
27          else {
28             return l.ownedParameter.
29                matches(r.ownedParameter);
30          }
31       }
32       else return false;
33    }
34
35    do {
36       l.ownedParameter.doMatch(r.ownedParameter);
37    }
38 }
39
40 @lazy
41 rule Parameter
42    match l : Left!Parameter
43    with r : Right!Parameter {
44
45    compare : l.type.matches(r.type)
46
47 }
```

In line 1, the Class rule specifies that it can compare two classes. The *compare* part of the rule specifies that the two classes are to be compared only in terms of their names. If for two classes the *compare* condition holds, the *do* part in line 8 is executed. The *do* part instructs the engine to compare the operations of the two classes using the built-in *doMatch* operation.

In line 13, the Operation rule specifies that it can compare two operations. The rule is marked as *lazy* which means that it will not be invoked by the engine on all the possible pairs of operations in the two models but instead, it will be invoked only when it is needed (via a call to the *matches* or the *doMatch* operations - the latter being the case in line 8). To compare two operations, we exploit the semantics of UML which dictate that there can not be two operations in the same class with the same name and parameter types. First we perform a basic match in which we check that the two operations have the same name and store the result in the *basicMatch* variable in line 19. If the basic comparison succeeds, in line 22 we check if the two classes that contain the operations have other operations with the same name. If there is only one operation with the same name in each class we can disregard all other information and assume that the two operations match. If not, we also need to check that the parameters of the two operations match in line 28. Finally, if the two operations are found to be matching, in line 36 we instruct the engine to establish matches for their parameters. This invokes the lazy Parameter rule in line 41 that compares the parameters based only on their types (as a change in the name is semantically indifferent for operation parameters in UML).

We have used the fully-fledged ECL comparison module (an excerpt of which is discussed above) in order to compare UML 2.1 models obtained using reverse engineering techniques from Eclipse plugins. To visualize the results of the comparison we have used the integration of ECL with EMF Compare as shown in Figure 4.

This short example has demonstrated that ECL can be used to implement fine-grained comparison algorithms that exploit the semantics of the respective metamodel. Also, it illustrates that the use of *lazy* rules in combination with the built-in *matches* and *doMatch* operations can significantly reduce the size and runtime complexity of the comparison. Although this example has demonstrated comparing models of a common metamodel (UML 2.1), it is worth stressing again that ECL can be also used to compare models of different metamodels and modelling technologies, a feature provided by the underlying Epsilon platform.

6 Conclusions and Further Work

We have presented the ECL language, which enables developers to implement rule-based semantically-rich comparisons between models of diverse metamodels. We have demonstrated the advantages of ECL against generic similarity-based algorithms; mainly the ability to explicitly define complex matching criteria and to exploit the semantics of the respective metamodels in order to reduce the number of actual comparisons - and as a result enhance performance. On the other hand, compared to the effort required for customizing a similarity-based algorithm, constructing a custom language-specific comparison with ECL requires substantially more skills and effort. To this end, the decision on whether to use a fine-grained approach such as ECL or a similarity-based approach is not clear for all cases; it depends to the amount of effort that can

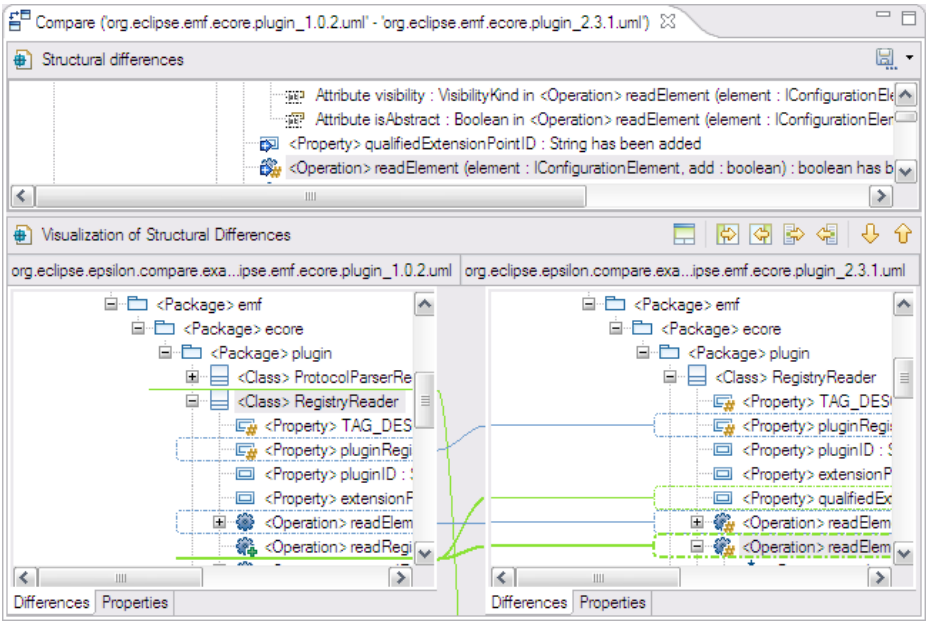


Fig. 4. The results of comparing two UML 2.1 models with ECL within EMF Compare

be assigned to the task, and the required accuracy. In our experience, similarity-based algorithms general perform quite well both; however, there always appear to be corner cases where a more explicit specification of the matching criteria is required. Therefore, modifying a generic similarity-based algorithm, so that it can be overridden partially by a more explicit approach such as ECL in order to handle the corner cases, appears to be a promising direction for further work in this area.

Acknowledgements

The author would like to thank Davide Di Ruscio for providing the reverse engineered models used in the example of Section 5, as well as Richard Paige and Alfonso Pierantonio for discussions that have contributed to the formation of the views reflected in this paper. 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. Kolovos, D.S., Di Ruscio, D., Paige, R.F., Pierantonio, A.: Different Models for Model Matching: An analysis of approaches to support model differencing. In: Proc. 2nd Workshop on Comparison and Versioning of Software Models. ACM/IEEE ICSE, Vancouver (2009) (to appear)

2. Treude, C., Berlik, S., Wenzel, S., Kelter, U.: Difference computation of large models. In: ESEC-FSE: Proceedings of the the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering, Dubrovnik, Croatia, pp. 295–304 (2007)
3. Lin, Y., Gray, J., Jouault, F.: DSMDiff: A Differentiation Tool for Domain-Specific Models. European Journal of Information Systems 16(4), 349–361 (2007); Special Issue on Model-Driven Systems Development
4. Extensible Platform for Specification of Integrated Languages for mOdel maNagement (Epsilon), http://www.eclipse.org/gmt/epsilon
5. Object Management Group. Meta Object Facility (MOF) 2.0 Core Specification, http://www.omg.org/cgi-bin/doc?ptc/03-10-04
6. Eclipse Foundation. Eclipse Modelling Framework, http://www.eclipse.org/emf
7. Alanen, M., Porres, I.: Difference and Union of Models. Technical Report 527, TUCS (April 2003)
8. Fleurey, F., Baudry, B., France, R., Ghosh, S.: A Generic Approach For Automatic Model Composition. In: Proc. 11th International Workshop on Aspect-Oriented Modeling, Nashville, USA (September 2007)
9. Chauvel, F., Fleurey, F.: Kermeta Language Overview, http://www.kermeta.org
10. Fabro, M.D.D., Valduriez, P.: Semi-automatic model integration using matching transformations and weaving models. In: Proceedings of the 2007 ACM symposium on Applied computing, Seoul, Korea, pp. 963–970 (2007)
11. Jouault, F., Kurtev, I.: Transforming Models with the ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
12. Sun Microsystems. Meta Data Repository, http://mdr.netbeans.org
13. Kolovos, D.S., Paige, R.F., Polack, F.A.C.: The Epsilon Object Language. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 128–142. Springer, Heidelberg (2006)
14. Navarro, G.: A guided tour to approximate string matching. ACM Computing Surveys (CSUR) 33(1), 31–88 (2001)
15. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady 10, 707–710 (1966)

Dependent and Conflicting Change Operations of Process Models

Jochen M. Küster[1], Christian Gerth[1,2], and Gregor Engels[2]

[1] IBM Zurich Research Laboratory, Säumerstr. 4
8803 Rüschlikon, Switzerland
{jku,cge}@zurich.ibm.com
[2] Department of Computer Science, University of Paderborn, Germany
{gerth,engels}@upb.de

Abstract. Version management of models is common for structural diagrams such as class diagrams but still challenging for behavioral models such as process models. For process models, conflicts of change operations are difficult to resolve because often dependencies to other change operations exist. As a consequence, conflicts and dependencies between change operations must be computed and shown to the user who can then take them into account while creating a consolidated version. In this paper, we introduce the concepts of dependencies and conflicts of change operations for process models and provide a method how to compute them. We then discuss different possibilities for resolving conflicts. Using our approach it is possible to enable version management of process models with minimal manual intervention of the user.

1 Introduction

Version management of models is a crucial technique for enabling modeling in distributed modeling scenarios and has recently been identified as one challenge in model management [9]. In general, it requires to compute and visualize changes that have been performed on a common source model while creating different versions. Based on this, version management capabilities then have to enable the user to create a consolidated model, by accepting or rejecting changes and thereby modifying the original source model.

A key requirement for consolidation of changed models is that it should impose minimal manual overhead on the user: Otherwise, a straightforward solution would be that the user remodels all changes manually. Nowadays, version management is a common functionality of mainstream modeling tools such as the IBM Rational Software Architect [18]. However, for behavioral models such as process models, inspecting and accepting or rejecting changes can involve quite some overhead if the changes to be dealt with are numerous. One reason for this is that the semantics of behavioral models is usually more complex than for structural models. A straightforward approach to compute all changes on model elements (called elementary changes) and display them is difficult to handle for the user: typically, elementary changes cannot be considered in isolation but must be aggregated to compound changes [16,27].

To enable a high degree of automation within consolidation of changes, it is important to understand dependencies and conflicts of changes. Informally, if two changes are

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, pp. 158–173, 2009.

dependent, then the second one requires the application of the first one. If two changes are in conflict, then only one of the two can be applied. Other than in structural models, in behavioral models changes are often dependent on one another. As a consequence, an approach for computing dependent and conflicting compound changes is required. Further, once conflicts have been computed, techniques for resolving conflicts are needed that take into account the characteristics of the modeling language.

In this paper, we study dependencies and conflicts of compound changes for process models. We first capture each of our compound change operations as a model transformation and then compute critical pairs [3,10,11] which can be used for detecting dependent and conflicting transformations. We then show how the results from critical pair analysis can be encoded as conditions which enable fast checks for dependencies and conflicts. We extend dependencies and conflicts to change sequences and provide a means of breaking up a change sequence into individual subsequences such that they can be dealt with separately in the conflict resolution process. For conflict resolution, we propose several resolution options that take into account characteristics of compound change operations. Using our approach, dependencies and conflicts of compound change operations in change logs can be computed and displayed to the user. In an evaluation we show that our approach leads to considerable less dependencies and conflicts and also to less user intervention for inspecting and resolving changes compared to an approach based on elementary changes.

The paper is structured as follows. First, in Section 2 we introduce our example scenarios that we obtain when performing process modeling in a distributed environment. In Section 3, we discuss how dependencies and conflicts of change operations can be defined and computed. In Section 4, we extend the notion of dependency and conflict to change sequences and in Section 5 we present our approach to conflict resolution. Section 6 reports on tool support and an evaluation of our approach. Finally, we discuss related work and future work.

2 Background

In this section, we introduce our case study motivated by process modeling in the IBM WebSphere Business Modeler [1]. Figure 1 shows an example business process model V from the insurance domain. The language supported by IBM WebSphere Business Modeler has similarities to UML 2.0 Activity Diagrams [22]: Nodes can be *Actions* or *ControlNodes* where *ControlNodes* contain Decision and Merge, Fork and Join, InitialNodes and FinalNodes. Nodes are connected by control flow as it is known from UML Activity Diagrams. In the example in Figure 1, an insurance claim is first checked, then it is recorded and then a decision is made whether to settle or reject it.

In a distributed modeling scenario, the process model V might have been created by the process model representative in an enterprise and then given to two colleagues for further elaboration. During this elaboration period, one colleague creates model V_1 and the other one model V_2. Afterwards, the process model representative is faced with the task of inspecting each change and then either accepting or rejecting it.

A common approach for version management of models is to capture possible operations performed on the model. For behavioral models such as process models, it is

possible to design compound change operations that transform a model from one consistent state into a new consistent state. Following this idea, we have previously proposed compound change operations for process models [16] as follows: *InsertAction*, *DeleteAction* or *MoveAction* operations allow to insert, delete or modify actions and always produce a connected process model as output. Each of the operations consists of several elementary changes such as creating a new action and redirecting source and targets of the edges. Similarly, *InsertFragment*, *DeleteFragment* and *MoveFragment* operations can be used for inserting, deleting or moving a complete fragment of the process model. Here, a fragment can either be an alternative fragment consisting of a Decision and a Merge node, a concurrent fragment consisting of a Fork and a Join node or further types of fragments including unstructured or complex fragments which allow to express all combinations of control nodes [16].

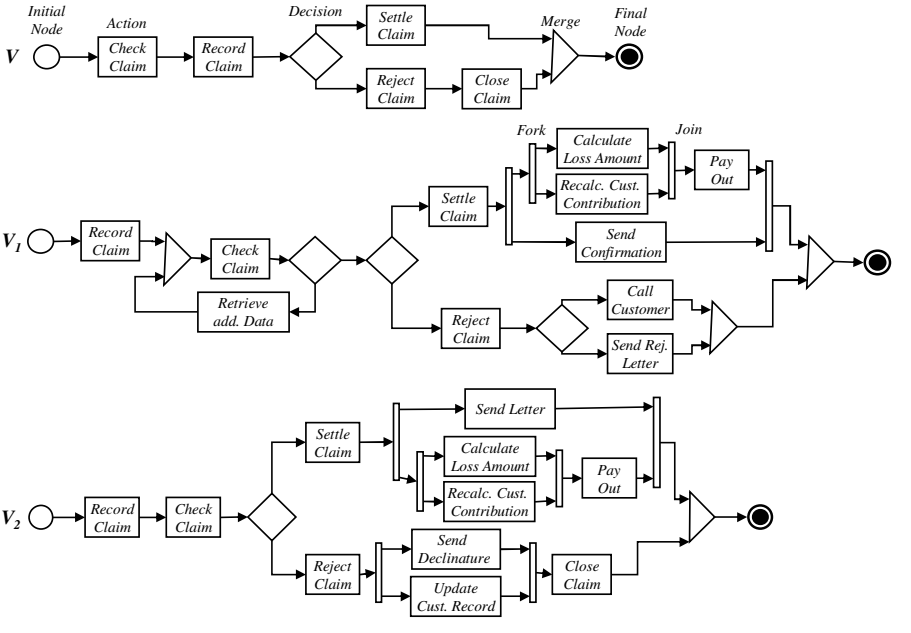


Fig. 1. Example

For the following discussions, we assume knowledge about newly introduced action nodes that are supposed to be identical in different versions, captured by a mapping of identical actions shown in Figure 2.

In addition, we assume that each sequence of change operations is recorded in a change log. This change log describes the change operations performed on the source model to obtain the target model and can either be logged during editing or reconstructed by comparing source and target model, proposed in [16].

Action Mapping (V_1, V_2)
- "Retrieve add. Data" – ""
- "Calculate Loss Amount" – "Calculate Loss Amount"
- "Recalc. Cust. Contribution" – "Recalc. Cust. Contribution"
- "Pay Out" – "Pay Out"
- "Send Confirmation" – "Send Letter"
- "Call Customer" – ""
- "Send Rej. Letter" – "Send Declinature"
- "" – "Update Cust. Record"

Fig. 2. Action mapping between V_1 and V_2

We further assume that this change log is clean, i.e. it does not contain unnecessary change operations that are later in the change log overridden [23]. In Figure 3, two change logs are given: $\Delta(V, V_1)$ describes the sequence of change operations for obtaining V_1 from V and $\Delta(V, V_2)$ describes the sequence of change operations for obtaining V_2 from V. For example, $\mathit{InsertAlt.Fragment}(F_A, \mathit{"Reject\ Claim"}, \mathit{"Close\ Claim"})$ introduces a new alternative fragment called F_A between the nodes *"Reject Claim"* and *"Close Claim"*.

Δ(V, V_1):

< InsertAlt.Fragment(F_A, "Reject Claim", "Close Claim"),
DeleteAction("Close Claim", F_A, Merge^2),
InsertCon.Fragment(F_{C1}, "Settle Claim", Merge^1),
InsertAction("Pay Out", Fork$^1_{FC1}$, Join$^1_{FC1}$),
InsertAction("Send Conf.", Fork$^2_{FC1}$, Join$^2_{FC1}$),
InsertCyclicFragment(F_{Cy}, "Record Claim", Decision),
MoveAction("Check Claim", InitialNode, "Record Claim", Merge$_{FCy}$, Decision$_{FCy}$),
InsertCon.Fragment(F_{C2}, Fork$^1_{FC1}$, "Pay Out"),
InsertAction("Ret. add. Data", Decision$^2_{FCy}$, Merge$^2_{FCy}$),
InsertAction("Calc. Loss Amount", Fork$^1_{FC2}$, Join$^1_{FC2}$),
InsertAction("Call Customer", Decision$^1_{FA}$, Merge$^1_{FA}$),
InsertAction("Send. Rej. Letter", Decision$^2_{FA}$, Merge$^2_{FA}$),
InsertAction("Recalc. Cust. Contrib.", Fork$^2_{FC2}$, Join$^2_{FC2}$) >

Δ(V, V_2):

< InsertCon.Fragment(F_{C3}, "Reject Claim", "Close Claim"),
InsertCon.Fragment(F_{C4}, "Settle Claim", Merge^1),
InsertCon.Fragment(F_{C5}, Fork$^2_{FC4}$, Join$^2_{FC4}$),
InsertAction("Send Letter", Fork$^1_{FC4}$, Join$^1_{FC4}$),
InsertAction("Pay Out", F_{C5}, Join$^2_{FC4}$),
MoveAction("Check Claim", InitialNode, "Record Claim", "Record Claim", Decision),
InsertAction("Send Declinature", Fork$^1_{FC3}$, Join$^1_{FC3}$),
InsertAction("Calc. Loss Amount", Fork$^1_{FC5}$, Join$^1_{FC5}$),
InsertAction("Update Cust. Record", Fork$^2_{FC3}$, Join$^2_{FC3}$),
InsertAction("Recalc. Cust. Contrib.", Fork$^2_{FC5}$, Join$^2_{FC5}$) >

Fig. 3. Change logs $\Delta(V, V_1)$ and $\Delta(V, V_2)$

For the following discussion, we distinguish between two scenarios: In the *single user scenario*, a sequence of change operations is performed on a model V, obtaining model V_1. Afterwards this sequence of change operations needs to be displayed to the user and for each change the user either has to confirm or reject it. In the *multi-user scenario*, two sequences of change operations are performed concurrently on V, leading to V_1 and V_2. Afterwards, all change operations are reconsidered and either rejected or confirmed.

Requirements for both scenarios are that the application of changes should be automatic and involve minimal user interaction. This requires that the change operations can be executed automatically and requires the validity of their parameters: If the parameters are invalid then a change operation becomes non-applicable. For this purpose, it is important that dependencies between change operations in the change sequences are known: The rejection of one operation can turn other operations non-applicable. For example, the rejection of an *InsertAlt.Fragment* operation leads to the non-applicability of all operations operating on this fragment. In addition, in the multi-user scenario, conflicts need to be identified because it is impossible to apply both operations that are in conflict without adaptation. For example, $\mathit{InsertAlt.Fragment}(F_A, \mathit{"Reject\ Claim"}, \mathit{"Close\ Claim"})$ and $\mathit{InsertCon.Fragment}(F_{C3}, \mathit{"Reject\ Claim"}, \mathit{"Close\ Claim"})$ are in conflict because either an alternative or a concurrent fragment is inserted at the same position. This means that once one of the change operations has been chosen the other one becomes non-applicable. In the following, we first provide a concept for dependencies and conflicts of change operations and then proceed to conflict resolution.

3 Dependencies and Conflicts of Change Operations

In this section, we establish the notions of dependencies and conflicts of change operations and discuss how to compute them. We first formalize change operations using graph transformations and then compute potential dependencies and conflicts of change operations.

3.1 Metamodel and Change Operations

Change operations can be formalized over a process model metamodel as has been done previously for other model transformation rules. We assume a business process model defined by the simplified metamodel shown in Figure 4 consisting of nodes connected by edges. Nodes can be *Actions* or *ControlNodes* or *Fragments*. *ControlNodes* contain Decision and Merge, Fork and Join, InitialNodes and FinalNodes. We assume that the metamodel is restricted by constraints and in particular that for *Actions*, only at most one incoming and outgoing edge is allowed. Fragments are an extension that allow us to represent a decomposition of the process model which can be computed using existing algorithms [26]. Fragments can be used for various analysis purposes such as control and data flow analysis but are also beneficial in the context of version management because they allow to detect and specify compound changes [16].

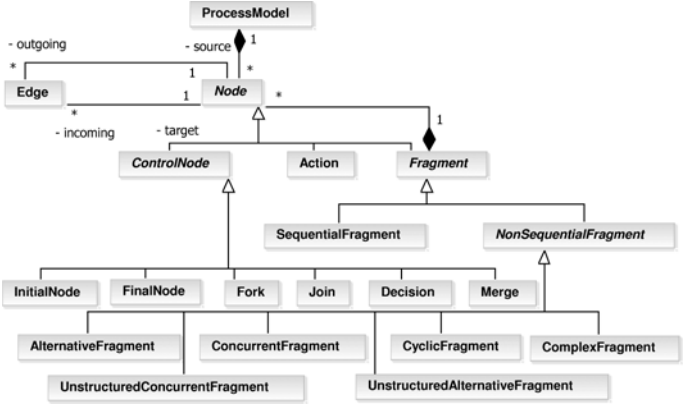


Fig. 4. Metamodel for process models

Each change operation c on a model V can be viewed as a model transformation rule which can be formalized as a typed attributed graph transformation rule [11,14,21] where the type graph represents the metamodel. A typed graph transformation rule $p : L \to R$ consists of a pair of typed instance graphs L, R such that the union is defined. A graph transformation step from a graph G to a graph H, denoted by $G \overset{p(o)}{\Longrightarrow} H$, is given by a graph homomorphism $o : L \cup R \to G \cup H$, called occurrence, such that the left hand side is embedded into G and the right hand side is embedded into H and precisely that part of G is deleted which is matched by elements of L not belonging to R, and, that part of H is added which is matched by elements new in R. For a rule p, an inverse rule p^{-1} can be constructed that inverts the transformation defined by p.

The change operations used in Figure 3 are specified as graph transformation rules in Figure 5. Here, new elements and deleted elements are visualized using dashed lines

and dotted lines, respectively. The *InsertAction* operation inserts a new *Action* between two existing nodes and also reconnects the process model such that it stays connected. For this purpose, the left hand side of the rule matches a fragment f and two nodes a and b connected by an edge e. It then creates a new *Action* x and a new edge $e2$ and redirects the target of the edge $e1$ to be the new *Action*. In a similar way, *DeleteAction* and *MoveAction* delete an action or move an action, respectively. Fragment operations are used for inserting, deleting or moving a fragment of the process model. Note that fragments can be concurrent fragments or alternative fragments or of a further type. Details about the fragment structure are left out here for simplification.

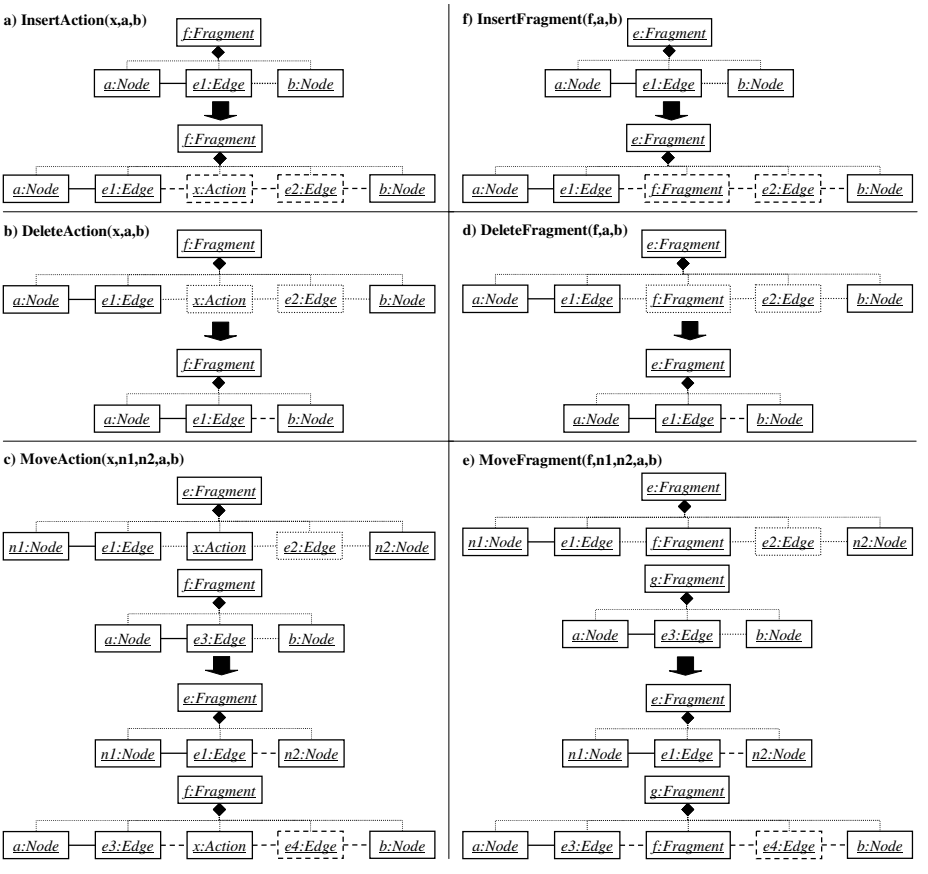


Fig. 5. Specification of operations dealing with Actions and Fragments

3.2 Dependencies and Conflicts of Changes

For graph transformation, dependencies and conflicts have been defined [5,10,21]: Formally, given a sequence of graph transformations $G \overset{p_1(o_1)}{\Longrightarrow} H_1 \overset{p_2(o_2)}{\Longrightarrow} X$, $H_1 \overset{p_2(o_2)}{\Longrightarrow} X$ is *(weakly sequential) independent* of $G \overset{p_1(o_1)}{\Longrightarrow} H_1$ if the occurrence $o_2(L_2)$ is already

present before the application of p_1. This is the case if $o_2(L_2)$ does not overlap with objects created by p_1. If in addition p_2 does not delete objects that are needed for the application of p_1, then p_1 and p_2 can be exchanged and are called sequentially independent.

Formally, given two graph transformations $G \overset{p_1(o_1)}{\Longrightarrow} H_1$ and $G \overset{p_2(o_2)}{\Longrightarrow} H_2$, $G \overset{p_1(o_1)}{\Longrightarrow} H_1$ is *(weakly parallel) independent* of $G \overset{p_2(o_2)}{\Longrightarrow} H_2$ if the occurrence $o_1(L_1)$ of the left-hand side of p_1 is preserved by the application of p_2. This is the case if $o_1(L_1)$ does not overlap with objects that are deleted by p_2. If the two transformations are mutually independent, they can be applied in any order yielding the same result. In this case we speak of *parallel independence*. Otherwise, if one of two alternative transformations is not independent of the second, the second will disable the first. In this case, the two steps are *in conflict*. According to the Local Church Rosser Theorem [5][1], parallel independence of two transformation steps induces their sequential independence and vice versa (with adapted occurrences).

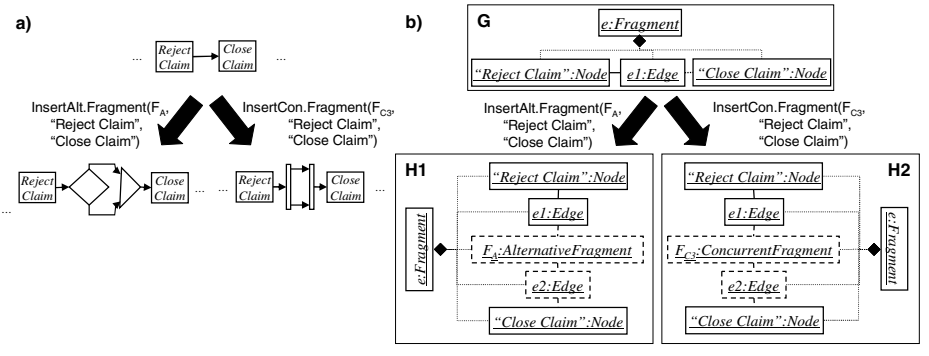


Fig. 6. A conflict between two changes

Often, we are not only interested to know whether two particular transformation steps are parallel or sequentially independent but also whether two transformation rules are parallel or sequentially independent. Related work (e.g. [10]) already discusses the notion of *potential* conflicts and dependencies. Given two rules p_1, p_2, a potential conflict or dependency occurs if there exist transformation steps such that a conflict or sequential dependency occurs. Given two rules p_1 and p_2, the computation of potential conflicts and dependencies can be done using critical pairs. A critical pair is a pair of transformation steps $H_1 \overset{p_1(o_1)}{\Longleftarrow} G \overset{p_2(o_2)}{\Longrightarrow} H_2$ which are in conflict and with the property that G is minimal.

Critical pairs of two rules p_1 and p_2 can be computed by overlapping the left hand sides of p_1 and p_2 in all possible ways such that there exists at least one object that is deleted by one of the rules and both rules are applicable. Figure 6 a) shows two conflicting changes in concrete syntax from our example and Figure 6 b) shows the

[1] The Local Church Rosser Theorem has been proven for typed attributed graph transformation in [8].

critical pair for this situation. Here, both changes insert a fragment at the same position in G. If one of the changes is applied, the other one will not be applicable anymore.

In the following, we discuss the results of critical pair analysis obtained for the different scenarios. In the single-user scenario it is important to know which changes can be independently rejected/confirmed. This can be achieved by studying compound operations for sequential independence. The idea is here to determine when compound operations are sequentially independent based on the parameters they have. As an example, an *InsertFragment* operation followed by an *InsertAction* operation into the fragment leads to a dependency. This means that if the *InsertFragment* operation is rejected, also the (dependent) *InsertAction* operation needs to be rejected.

In order to compute the sequential dependencies between compound changes, given two rules p_1 and p_2, we compute critical pairs of p_1 and p_2^{-1} and p_1^{-1} and p_2 [10]. The critical pairs obtained are then encoded by specifying conditions on the parameters of the operations and captured in a dependency matrix[2]. An excerpt of the dependency matrix is shown in Figure 7 (a)[3] specifying dependent configurations for *InsertAction* operations. For example, $InsertAction(X1, A, B)$ and $InsertFragment(F2, C, D)$ are sequentially dependent if $C = X1 \vee D = X1$.

In the multi-user scenario, given two rules p_1 and p_2, we compute the critical pairs of p_1 and p_2 for all combinations of change operations. Critical pairs obtained are then encoded by specifying conditions on the parameters of p_1 and p_2, partially shown in a conflict matrix in Figure 7 (b)[5] for our *InsertAction* operations.

(a) Sequential Dependencies	InsertAction (X2,C,D)	DeleteAction (X2,C,D)	MoveAction (X2,oQ,oT,nQ,nT)	InsertFragment (F2,C,D)
InsertAction (X1,A,B)	[IA(X1), IA(X2)]: C = X1 v D = X1	[IA(X1), DA(X2)]: X2 = X1 v (D = X1 & X2 = A) v (C = X1 & X2 = B)	[IA(X1), MA(X2)]: (nQ = A & nT = X1) v (nQ = X1 & nT = B) v (X2 = A & oT = B) v (oQ = X1 & X2 = B) v (oQ=A & X2=X1 & oT=B)	[IA(X1), IF(F2)]: C = X1 v D = X1

(b) Conflicts	InsertAction (X2,C,D)	DeleteAction (X2,C,D)	MoveAction (X2,oQ,oT,nQ,nT)	InsertFragment (F2,C,D)
InsertAction (X1,A,B)	(C = A & D = B)	(C = A & X2 = B) v (X2 = A & D = B)	(nQ = A & nT = B) v (X2 = A & oT = B) v (oQ = A & X2 = B)	(C = A & D = B)

Fig. 7. Configurations of compound operations that lead to sequential dependencies **(a)** and conflicts **(b)**

4 Dependencies and Conflicts of Change Sequences

Until now we have studied dependencies and conflicts of change operations in isolation. We now extend our concept of dependencies and conflicts to change sequences in order to deal with change logs as introduced above.

[2] We used the AGG tool [25] to partially compute and validate the entries of the matrices. However, AGG does currently not support inheritance in the type graph which required a simplification of rules.

[3] The complete dependency and conflict matrices are given in [15].

4.1 Dependencies of Change Sequences

For the following discussion, we assume that a change sequence $\Delta = \langle t_1(o_1), .., t_n(o_n)\rangle$ consists of a sequence of transformation steps t_i at an occurrence o_i such that the transformation $G = S_0 \overset{t_1(o_1)}{\Longrightarrow} S_1..S_{n-1} \overset{t_n(o_n)}{\Longrightarrow} S_n = H$ exists. Informally, a change sequence Δ can be considered as a concatenation of model transformations and represents a change log as introduced before. As a shorthand, we also write $\Delta = \langle t_1, .., t_n\rangle$.

Given a change sequence $\Delta = \langle t_1, .., t_n\rangle$, we are interested in sequential dependencies because these are the changes that cannot be resolved in any order. Potential dependencies that can occur between two changes have been captured in the dependency matrix, shown partially in Figure 7. Based on this, a given change sequence $\Delta = \langle t_1, .., t_n\rangle$ can be broken up into subsequences c_i such that the following holds:

- each subsequence c_i consists of a sequence of change operations $t_i \in \Delta$, i.e. $c_k = \langle t_l, .., t_r\rangle$ with the property that t_i is not dependent of any change operation not contained in c_k, and
- for two subsequences c_k and c_l, the change operations contained are disjoint.

These subsequences can be computed as follows: Given a Δ, we compute for each pair of compound changes t_i and t_j sequential dependencies. Thereby we check whether operations t_i and t_j with their concrete parameters form a critical pair according to the dependency matrix given in [15]. If t_i and t_j are dependent, they belong to the same subsequence.

The dependency matrix can only indicate a sequential dependency between two operations whose signatures overlap. There are cases where a sequential dependency exists and signatures do not overlap. These dependencies will be detected in a transitive way. For instance, the sequential dependency of *InsertAction*(*"Calc. Loss Amount"*, $Fork^1_{FC5}$, $Join^1_{FC5}$) on *InsertConcurrentFragment*(F_{C4}, *"Settle Claim"*, $Merge^1$) (fragment F_{C4}) will be detected transitively since the insertion of the action *"Calc. Loss Amount"* is dependent on *InsertConcurrentFragment*(F_{C5}, $Fork^2_{FC4}$, $Join^2_{FC4}$) (fragment F_{C5}) which is in turn dependent on the insertion of fragment F_{C4}.

In the end, each t_i belongs to exactly one subsequence and the operations in different subsequences are sequentially independent. According to the Local Church Rosser Theorem this induces parallel independence for operations in disjoint subsequences as well.

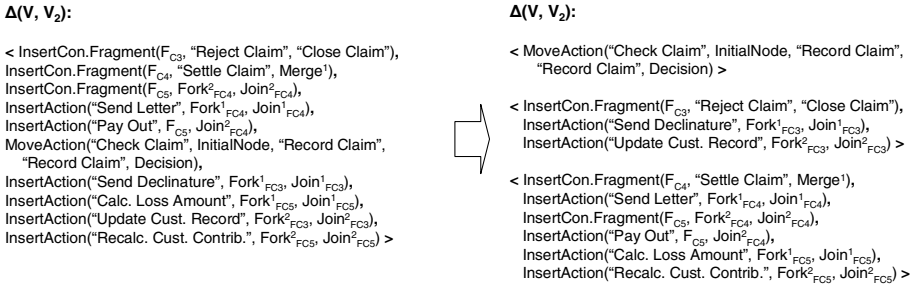


Fig. 8. Independent subsequences in $\Delta(V, V_2)$

Figure 8 shows the sequence of changes applied on our example model V in order to obtain V_2 and the decomposition of these changes into parallel independent subsequences. The parallel independent subsequences for change sequences are important for several reasons: firstly, they show which changes are dependent which is important for the single-user scenario. Secondly, for the multi-user scenario, the parallel independent subsequences will be used for computing conflicts.

4.2 Conflicts of Change Sequences

Given two change sequences $\Delta_1 = \langle t_1, .., t_n \rangle$ and $\Delta_2 = \langle s_1, .., s_m \rangle$, we first compute the parallel independent subsequences of each change sequence as described previously. Given two subsequences $c_k = \langle t_i, ..t_j \rangle \in \Delta_1$ and $d_l = \langle s_m, .., s_n \rangle \in \Delta_2$ we are then interested in conflicts because these must be taken into account when rejecting or accepting changes in the multi-user scenario.

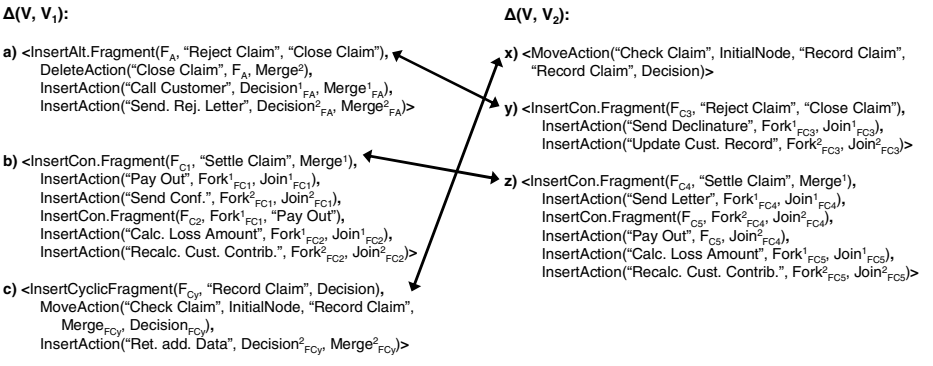


Fig. 9. Computation of conflicts between subsequences of change sequences

Conflicts can be computed based on the results of critical pair analysis which determines potential conflicts, shown partially in Figure 7. For computation of conflicts, the operations in two change sequences are analyzed pairwise for conflicts. Figure 9 shows the result of this computation for our example. Here, three conflicts occur, indicated by the arrows. Once conflicts have been determined, conflicts need to be resolved. For this, different options exist that will be discussed in the next section.

After resolving a conflict between c_k and d_l, new conflicts between Δ_1 and Δ_2 can occur. For identifying these, we recompute conflicts after each conflict resolution. Optimizations of this procedure where recomputation of conflicts is restricted to certain subsequences is left for future work. Resolving a conflict can also lead to less conflicts if an operation together with its subsequence is rejected and its dependent operations also become non-applicable. In the following section, we will elaborate on conflict resolution.

5 Conflict Resolution

In this section, we discuss the different options for conflict resolution. For the following discussion, we assume that two change sequences Δ_1 and Δ_2 exist that have been

divided into parallel independent subsequences as previously explained. For a given conflict, conflict resolution can consist of (at least) the following choices:

- *selection of the subsequence to adopt*, meaning that the complete other subsequence is discarded and not considered further,
- performing a *combination of the two operations or unifying the two operations*. The operations in conflict have a similar type or are structurally very similar. In such a case, the conflict can be resolved by performing one operation and establishing a mapping between the elements used. If the operations cannot be unified directly, i.e. one operation inserts a fragment with six branches, the other one with only two branches, then a common superset or subset can be chosen.
- *both operations are performed* by modifying one or both operations, leading e.g. to a sequential or parallel insertion of fragments or actions.

The choice which type of conflict resolution to adopt is made by the user, usually based on his or her domain knowledge of the models, and cannot be automated.

In many cases, the decision about conflict resolution influences the change operations that are dependent on the conflicting operations. In the case of combination using unification, the parameters of the dependent operations have to be recomputed by replacing the unified parameters of the conflicting operations. In the case of a combination by introducing a new operation, this also yields to recomputation of parameters.

In the case that one of the two subsequences is adopted and the other one is discarded, it is important to know about possible conflicts that occur within the adopted subsequence. By adoption, all the operations inside the subsequence will also be adopted, meaning that in case of a conflict this type of conflict resolution will be chosen for contained operations as well.

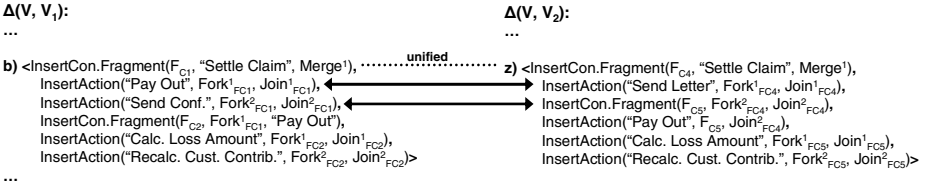


Fig. 10. Unification of two conflicts

In our example, a conflict which is likely to be resolved by a unification is the conflict between subsequence b) and z) shown in Figure 10. After the unification of *InsertCon.Fragment*$(F_{C1}, ...)$ and *InsertCon.Fragment*$(F_{C4}, ...)$ the parameters and conflicts for the changes that are dependent on the unified changes are recomputed. In this case, F_{C1} and F_{C4} are unified as well as the nodes $Fork_{FC1}$ and $Fork_{FC4}$ and $Join_{FC1}$ and $Join_{FC4}$. This leads to two additional conflicts between *InsertAction*$(Pay\ Out, ...)$ and *InsertAction*$(Send\ Letter, ...)$ as well as *InsertAction*$(Send\ Conf., ...)$ and *InsertCon.Fragment*$(F_{C5}, ...)$, because due to the unification the dependent changes are now applied in the same concurrent fragment and their parameters overlap.

Conflict resolution entails the application of one or both conflicting, possibly modified or adapted, change operations. After this, conflicts between the following operations are recomputed and displayed to the modeler, leading to a iterative resolution process.

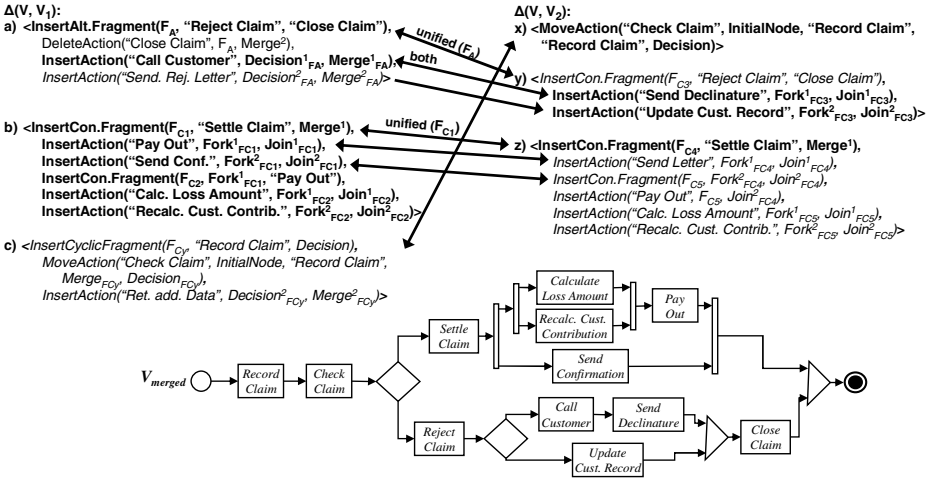


Fig. 11. A possible merged process model based on V and the modifications made in V_1 and V_2

Figure 11 illustrates one possible resulting process model V_{merged} based on the modifications made in V_1 and V_2. In order to visualize the conflict resolution process, applied compound changes are printed in bold letters and rejected changes in italic letters. We start the conflict resolution by unifying the conflict between subsequence a) and y) and applied $InsertAlt.Fragment(F_A, ...)$ on model V. By the unification, all occurrences of $Fork^*_{FC3}$ and $Join^*_{FC3}$ in the signatures of the remaining operations are substituted by $Decision^*_{FA}$ and $Merge^*_{FA}$. Thereby, two new conflicts between the *InsertAction* operations in subsequence a) and y) arise. In case of the conflict between $InsertAction("Call\ Customer", ...)$ and $InsertAction("Send\ Declinature", ...)$, we select both operations for application, leading to a sequential insertion of the two actions. In the other case, we apply $InsertAction("Update\ Cust.\ Record", ...)$ and reject $InsertAction("Send\ Rej.\ Letter", ...)$. Finally, we apply $DeleteAction("Close\ Claim", ...)$. Further, we resolve the conflict between b) and z) by unification as described previously and then apply only operations in b). For the resolution of the conflict between subsequence c) and x), we decide to adopt only subsequence x) and rejected all operations contained in subsequence c). This example shows that using our approach it is possible to resolve conflicts between change sequences in an iterative way with minimal manual intervention such that a consolidated process model is constructed.

6 Tool Support and Evaluation

In this section, we report on tool support and evaluation of our approach. The dependency and conflict detection approach has been implemented as a prototype for IBM

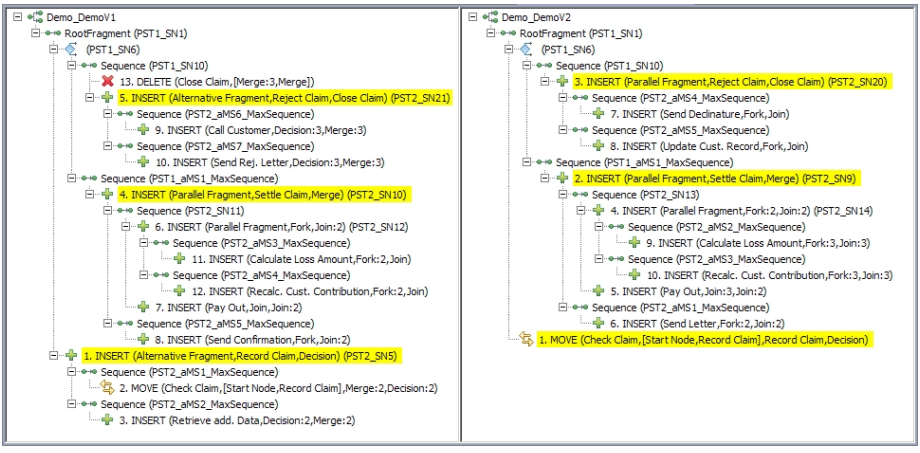


Fig. 12. Business Process Merging Prototype in the IBM WebSphere Business Modeler

WebSphere Business Modeler. Figure 12 shows a screenshot of the extension with the example and computed conflicts.

One goal of our evaluation was to show that our approach leads to less conflicts and dependencies than an approach relying on elementary change operations. Another goal was to show that our approach then also leads to less required user intervention than an approach based on elementary operations. Figure 13 provides an overview of our results. Detailed results of our evaluation and the case study can be found in [15].

	Elementary Changes		Compound Changes	
	Δ(V, V1)	Δ(V, V2)	Δ(V, V1)	Δ(V, V2)
# of Change Operations	42	33	13	10
# of Dependencies	45	36	10	7
# of initial Conflicts	23		3	
# of Work Units for Sample Resolution	22		7	

Fig. 13. Evaluation results for approaches based on elementary and compound operations

We can distinguish between application of operations, conflict examination and conflict resolution. On average, the number of elementary operations is three times the number of compound operations which means that for application of operations the user intervention triples (unless further optimizations are implemented for the elementary operations). The relation of conflicts for the elementary and compound operations cannot be estimated. In our example, we obtain the number of conflicts as indicated in the table. The user intervention required for conflict resolution depends on the support given by the modeling tool. In [15] we give a detailed comparison of the required user intervention for one conflict resolution example. The results of this (measured in work units, see Figure 13) show that the user intervention again almost triples.

Our evaluation has also shown that compound operations can be used to realize advanced functionality such as change operation unification which is difficult to realize for elementary operations, unless they are grouped again to compound operations. In addition, compound operations enable to always create a connected and well-formed model during conflict resolution whereas using elementary operations elements often have to be reconnected manually.

7 Related Work

Mens et al. [21] analyze refactorings for structural conflicts using critical pair analysis. They first express refactorings as graph transformations and then detect conflicts using the AGG tool [25]. Hausmann et al. [10] analyze functional requirements in a use-case driven software development approach for conflicts and dependencies. Further approaches including critical pair analysis include work by Mens et al. [20] for transformation dependency analysis. Lambers et al. [17] study rule sequences and formulate sufficient criteria for their applicability. All of these approaches are similar to ours with regards to the analysis of syntactic conflicts and dependencies and the formalization using graph transformation. However, there are also differences: Firstly, we analyze process model refactorings and elaborate on breaking up change sequences into independent ones. Further, our analysis is performed after the changes have been made for resolving conflicts whereas in the related work conflicts should be avoided up front.

Another area of related work is concerned with model composition and model versioning. Alanen and Porres [2] describe an algorithm how to compute elementary change operations in a similar setting as ours. Kolovos et al. [13] describe the Epsilon merging language which can be used to specify how models should be merged. Kelter et al. [12] present a generic model differencing algorithm. All these approaches aim at providing generic support for merging different models but do not focus on dependencies and conflicts of change operations. In contrast to these approaches, we provide a selection of conflict resolution techniques which is language-specific to process models, showing that there is a need for these domain-specific approach to dependency and conflict detection. As such, our approach can be categorized as an operation-based, tree-based and syntactic approach to software merging [19]. In the IBM Rational Software Architect [18] or using the EMF Compare technology [7], dependencies and conflicts between versions are computed based on elementary changes. The underlying conflict analysis can be customized by self-defined algorithms which can make use of our approach for establishing a conflict and a dependency matrix.

Cicchetti et al. [4] have recently proposed a metamodel for representing conflicts which can be used for specifying both syntactic as well as semantic conflicts. One key difference to our work is that we do not specify conflicts for compound operations but we compute them by using the critical pair approach. Finally, within the process modeling community, Dijkman [6] has categorized differences of process models in the context of process integration where models do not originate from a common source model. Rinderle et al. [24] have studied disjoint and overlapping process model changes in the context of the problem of migrating process instances but have not considered dependencies and conflicts between change sequences and different forms of conflict resolution.

8 Conclusion and Future Work

When modeling in a distributed environment, changes performed on models can be conflicting or sequentially dependent. In order to consolidate different models, conflicting changes must be computed and manually resolved. In this paper, we have shown how change operations can be analyzed for conflicts and dependencies. Based on this, we presented an approach for breaking up a sequence of change operations into subsequences such that change operations from different subsequences are independent. Our approach allows to make dependencies explicit and resolve conflicts in a versioning scenario with special language-specific conflict resolution choices. Our evaluation has shown that our approach leads to less user interaction than using elementary change operations.

There are several directions for future work: Firstly, we would like to validate our approach also for other behavioral models such as statecharts where compound change operations need to be designed and then analyzed for conflicts and dependencies in a similar way. Another area of future work is to take into account the semantics of process models in order to be able to identify those syntactic conflicts which do not represent a semantic conflict.

References

1. IBM WebSphere Business Modeler, http://www.ibm.com/software/integration/wbimodeler/
2. Alanen, M., Porres, I.: Difference and Union of Models. In: Stevens, P., Whittle, J., Booch, G. (eds.) UML 2003. LNCS, vol. 2863, pp. 2–17. Springer, Heidelberg (2003)
3. Bottoni, P., Schürr, A., Taentzer, G.: Efficient Parsing of Visual Languages based on Critical Pair Analysis and Contextual Layered Graph Transformation. In: VL 2000, pp. 59–60. IEEE Computer Society, Los Alamitos (2000)
4. Cicchetti, A., Di Ruscio, D., Pierantonio, A.: Managing Model Conflicts in Distributed Development. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 311–325. Springer, Heidelberg (2008)
5. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic Approaches to Graph Transformation Part I: Basic Concepts and Double Pushout Approach. In: Rozenberg, G. (ed.) Handbook of Graph Grammars and Computing by Graph Transformation. Foundations, vol. 1, pp. 163–245. World Scientific, Singapore (1997)
6. Dijkman, R.: A Classification of Differences between Similar Business Processes. In: EDOC 2007, pp. 37–50. IEEE Computer Society, Los Alamitos (2007)
7. Eclipse Foundation. EMF Compare, http://www.eclipse.org/modeling/emft/?project=compare
8. Ehrig, H., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graph transformation. In: Ehrig, H., Engels, G., Parisi-Presicce, F., Rozenberg, G. (eds.) ICGT 2004. LNCS, vol. 3256, pp. 161–177. Springer, Heidelberg (2004)
9. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: Briand, L.C., Wolf, A.L. (eds.) International Conference on Software Engineering, ISCE 2007, Workshop on the Future of Software Engineering, FOSE 2007, Minneapolis, MN, USA, May 23-25, pp. 37–54 (2007)
10. Hausmann, J.H., Heckel, R., Taentzer, G.: Detection of conflicting functional requirements in a use case-driven approach: a static analysis technique based on graph transformation. In: Proceedings ICSE 2002, pp. 105–115. ACM, New York (2002)

11. Heckel, R., Küster, J.M., Taentzer, G.: Confluence of Typed Attributed Graph Transformation. In: Corradini, A., Ehrig, H., Kreowski, H.-J., Rozenberg, G. (eds.) ICGT 2002. LNCS, vol. 2505, pp. 161–176. Springer, Heidelberg (2002)
12. Kelter, U., Wehren, J., Niere, J.: A Generic Difference Algorithm for UML Models. In: Liggesmeyer, P., Pohl, K., Goedicke, M. (eds.) Software Engineering 2005, Fachtagung des GI-Fachbereichs Softwaretechnik, 8.-11.3.2005 in Essen. LNI, vol. 64, pp. 105–116. GI (2005)
13. Kolovos, D.S., Paige, R., Polack, F.: Merging Models with the Epsilon Merging Language (EML). In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 215–229. Springer, Heidelberg (2006)
14. Küster, J.M.: Definition and validation of model transformations. Software and Systems Modeling 5(3), 233–259 (2006)
15. Küster, J.M., Gerth, C., Engels, G.: Dependent and Conflicting Change Operations of Process Models. IBM Research Report RZ 3727, IBM Zurich Research Laboratory (2009), http://www.zurich.ibm.com/~jku/Papers/rz3727.pdf
16. Küster, J.M., Gerth, C., Förster, A., Engels, G.: Detecting and Resolving Process Model Differences in the Absence of a Change Log. In: Dumas, M., Reichert, M. (eds.) BPM 2008. LNCS, vol. 5240, pp. 244–260. Springer, Heidelberg (2008)
17. Lambers, L., Ehrig, H., Taentzer, G.: Sufficient Criteria for Applicability and Non-Applicability of Rule Sequences. ECEASST 10 (2008)
18. Letkeman, K.: Comparing and merging UML models in IBM Rational Software Architect: Part 3. A deeper understanding of model merging. IBM Developerworks (2005), http://www.ibm.com/developerworks/rational/library/05/802_comp3/
19. Mens, T.: A State-of-the-Art Survey on Software Merging. IEEE Trans. Software Eng. 28(5), 449–462 (2002)
20. Mens, T., Van Der Straeten, R., D'Hondt, M.: Detecting and resolving model inconsistencies using transformation dependency analysis. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 200–214. Springer, Heidelberg (2006)
21. Mens, T., Taentzer, G., Runge, O.: Analysing refactoring dependencies using graph transformation. Software and System Modeling 6(3), 269–285 (2007)
22. Object Management Group (OMG). The Unified Modeling Language 2.0 (2005)
23. Rinderle, S., Jurisch, M., Reichert, M.: On Deriving Net Change Information From Change Logs - The DELTALAYER-Algorithm. In: Kemper, A., et al. (eds.) BTW 2007. LNI, vol. 103, pp. 364–381. GI (2007)
24. Rinderle, S., Reichert, M., Dadam, P.: Disjoint and Overlapping Process Changes: Challenges, Solutions, Applications. In: Meersman, R., Tari, Z. (eds.) OTM 2004. LNCS, vol. 3290, pp. 101–120. Springer, Heidelberg (2004)
25. Taentzer, G.: AGG: A Graph Transformation Environment for Modeling and Validation of Software. In: Pfaltz, J.L., Nagl, M., Böhlen, B. (eds.) AGTIVE 2003. LNCS, vol. 3062, pp. 446–453. Springer, Heidelberg (2004)
26. Vanhatalo, J., Völzer, H., Leymann, F.: Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition. In: Krämer, B.J., Lin, K.-J., Narasimhan, P. (eds.) ICSOC 2007. LNCS, vol. 4749, pp. 43–55. Springer, Heidelberg (2007)
27. Weber, B., Rinderle, S., Reichert, M.: Change Patterns and Change Support Features in Process-Aware Information Systems. In: Krogstie, J., Opdahl, A.L., Sindre, G. (eds.) CAiSE 2007 and WES 2007. LNCS, vol. 4495, pp. 574–588. Springer, Heidelberg (2007)

Enabling Automated Traceability Maintenance through the Upkeep of Traceability Relations

Patrick Mäder[1], Orlena Gotel[2], and Ilka Philippow[1]

[1] Department of Software Systems
Ilmenau Technical University, Germany
{patrick.maeder,ilka.philippow}@tu-ilmenau.de
[2] Department of Computer Science
Pace University, New York, USA
ogotel@pace.edu

Abstract. Traceability is demanded within mature development processes and offers a wide range of advantages. Nevertheless, there are deterrents to establishing traceability: it can be painstaking to achieve initially and then subject to almost instantaneous decay. To be effective, this is clearly an investment that should be retained. We therefore focus on reducing the manual effort incurred in performing traceability maintenance tasks. We propose an approach to recognize those changes to structural UML models that impact existing traceability relations and, based upon this knowledge, we provide a mix of automated and semi-automated strategies to update these relations. This paper provides technical details on the update process; it builds upon a previous publication that details how triggers for these updates can be recognized in an automated manner. The overall approach is supported by a prototype tool and empirical results on the effectiveness of tool-supported traceability maintenance are provided.

Keywords: Automated traceability maintenance; Model-driven engineering; Change management; Rule-based traceability; Traceability update.

1 Introduction

Establishing traceability on a project can be time consuming. Even with the support of emerging automated techniques there is still substantive manual work involved [1]. Providing for traceability is intended to support change management and many other software development tasks but, to leverage this investment, it is necessary to have an accurate and ready to use set of traceability relations. Traceability needs to be maintained while project artifacts are evolving.

By traceability maintenance, we refer to the modification and/or enhancement of existing traceability relations after changes to artifacts to ensure their continued correctness and accuracy. We highlighted a general approach for automated traceability maintenance in a previous paper [2]. The approach targets model-driven software development using UML and comprises two key tasks:

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, pp. 174–189, 2009.

(i) recognizing changes to models in terms of the broader development activity being undertaken; and then (ii) updating the impacted traceability relations to restore the traceability. The technical details underlying the automated recognition of development activities have been described in an earlier paper [3]. In this companion paper, we focus on the technical details associated with traceability upkeep and explain how the necessary updates are implemented.

The paper is organized as follows. In Section 2, we provide high-level details about our approach and describe the context of software development for which it has been developed initially. In Section 3, we describe our process for updating existing traceability relations after the recognition of development activities. We also describe how necessary updates can be propagated. In Section 4, we discuss our initial validation through an experiment that compares the effort and quality associated with tool-supported traceability maintenance, based upon the approach described, versus manual efforts. We end the paper with a summary of related and future work.

2 Motivation and Scope

Within this section, we discuss the problem of maintaining traceability relations and present an overview of our solution to this typically manual task. We first outline the context that we work within.

2.1 Model-Driven Software Development

With model-driven software development using the UML, a variable number of abstraction layers (i.e., models) can be created to document a problem and its solution, from the initial requirements through to the final implementation [4]. As the elements of these models describe the same system, there are benefits in establishing explicit traceability relations between models to handle change. Few industrial projects implement traceability in this fashion though, due to the perceived and actual costs, and struggle to find the right balance as project artifacts evolve [5]. Traceability maintenance can be a time consuming proposition.

2.2 Approach and Tool Support

Our approach is founded upon the following assumptions: (1) while evolving any kind of UML model, it is possible to capture the elementary change actions and salient information regarding the properties of the changed element; (2) one can understand the intention of these elementary change actions within the context of a chain of related change actions on an element and so determine the wider development activity; and (3) knowledge of an intentional development activity provides the information necessary for pre-existing traceability relations to be updated following the changes. Our approach therefore records all the changes to model elements and uses this information to find a match within a set of predefined patterns of recurring development activities. A match will instigate the requisite traceability update actions.

Changes to a UML model can be classified into three elementary types: adding new elements, deleting elements and modifying existing elements. All elementary changes that are not recognized as part of a wider development activity are handled as new additions or deleted elements of the model. For new elements, we support the developer in the creation of traceability relations. For deleted elements, we discard associated traceability relations if necessary. To restore overall traceability within a set of interrelated models, we support the propagation of required changes to models.

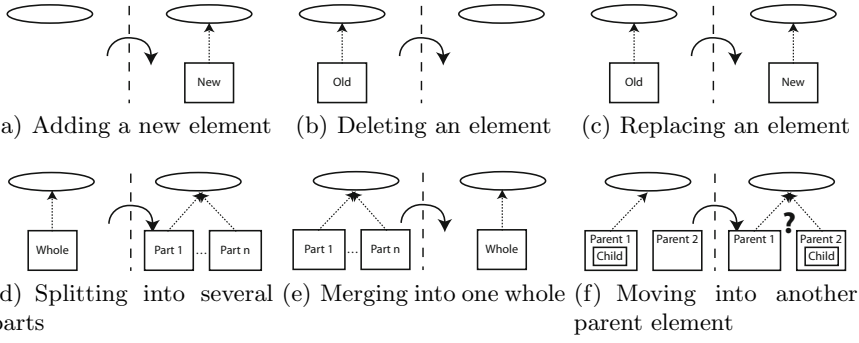
Currently, the approach is restricted to the analysis of changes to those models described via structural UML diagrams (e.g., class, object, composite structure, package and component diagrams). We focus on UML models as these are the de facto standard for model-based software development and a wide range of existing CASE tool support is provided. Furthermore, UML offers the possibility of capturing the full range of software development artifact via its diagrams and offers basic support for traceability relations as part of its meta-model [6]. We are currently investigating how to extend the approach to support the behavioral diagrams of the UML and also to handle additional types of model. More details are described in earlier papers ([2], [3]) and the approach is supported by a prototype tool called *traceMAINTAINER* [7].

3 Traceability Update Process

We restrict our focus to post-requirements traceability [8] and hence to the consequent modeling activities and artifacts of the software development lifecycle. The development process therefore consists of activities that incrementally transform a requirements specification into the final implementation (e.g., the platform specific model or source code) in a forward engineering manner. Each of these activities is applied to or influenced by various input artifacts and creates new or improved output artifacts. Creating an explicit traceability relation between these artifacts can capture these dependencies. We represent our traceability relations as stereotyped dependency relationships, as defined by the UML meta-model. The direction of a dependency relation points, by default, from the dependent model element towards the independent model element. This directionality is intended to convey semantics and, in our case, to assist traceability update and change propagation, but does not prevent bi-directional use or navigation of the traceability relation itself.

3.1 Types of Model Changes

Focusing on post-requirements traceability relations and restricting directionality, requirements can be treated as sinks of a directed acyclic graph, sourced by elements of implementation, test and so on. The nodes of this graph represent related elements in different models and the arcs represent traceability relations. Each related requirement spans such a tree. Changes to models that impact traceability are the possible changes to the traceability graph and are combinations of the elementary changes given in Section 2.2. In this section, we



(a) Adding a new element (b) Deleting an element (c) Replacing an element

(d) Splitting into several parts (e) Merging into one whole (f) Moving into another parent element

Fig. 1. Different change types with a focus on the necessary traceability updates

describe the types of change we distinguish at the model level and explain their relation to the change of the traceability graph. Figure 1 depicts all the change types and their traceability implications. Figure 2 depicts the underlying graph that is the subject of the examples in the sub-sections below.

Adding a New Element. This change enhances a model by adding a new element (see Figure 1(a)). While creating new traceability relations on the element, new nodes will be added to the traceability graphs of the associated requirements.

Recognition of change: Each change to a model is retained in a buffer of our *traceMAINTAINER* tool until the wider development activity has been identified. If we have not been able to assign the creation of a new model element (i.e., ADD event) to a wider development activity, we treat the change as an enhancement of the model. This means that the element is a new additional part of the model and may require the creation of a new traceability relation. The necessity to establish new traceability relations depends upon the project's traceability information model.

Required traceability update: After adding a new element, it might be necessary to relate the element to independent elements that are the reason for the addition. The traceability information model specifies what elements are allowed/intended to be linked for a project, so it is possible to suggest or even require the creation of one or more traceability relations on the new element. In cases of ambiguity, we add a tag to such elements for the developer to examine when convenient. On exiting a CASE tool, a dialog is provided to alert the user to any tagged elements that are yet to be connected within the traceability graph. The dialog provides possible counterparts for new traceability relations on the element according to the traceability information model. Also, the last traced element of that type will be offered, as it is common that several proximal changes belong to the implementation of the same new requirement. Recently changed elements are more highly ranked and we are currently investigating the possibility of using existing information retrieval techniques to provide a better ranking of candidate relations to aid this task.

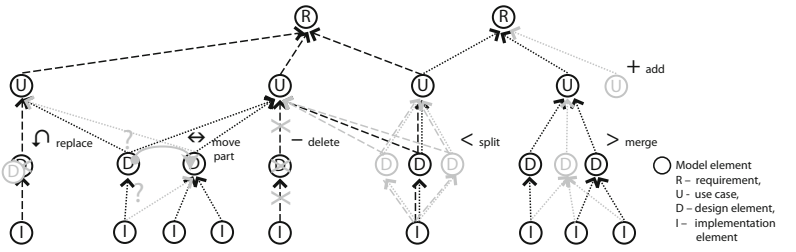


Fig. 2. Example of two overlapping traceability graphs implementing two requirements

Impact on existing traceability: This change type has no impact on the existing traceability relations but, if new elements are not related, the results of impact analysis and change propagation are likely to be wrong.

Deleting an Element. This change arises when an element is removed from a model without being replaced (see Figure 1(b)). While removing an element, its relations will also be deleted, so nodes from the traceability graphs are removed.

Recognition of change: This is similar to ADD events. For each autonomous DEL event that is removed from the *traceMAINTAINER* buffer we assume that the intention behind the activity is to remove the element from the model.

Required traceability update: If an element has been removed it is necessary to remove its traceability relations and examine the resulting impact. Using *traceMAINTAINER*, we store the traceability relations in an additional repository with the advantage that we are able to find out about an element's relations even after its deletion, but with the necessity to delete these inconsistent relations explicitly.

Impact on existing traceability: The deletion of inconsistent traceability relations is sufficient for relations to independent elements. For incoming links from dependent elements, however, it is necessary to check whether these elements are still valid and required or not. All these traceability relations will be kept and receive a *suspect* tag and the deletion of the model element will be added as the rationale for this status. Developers need to make decisions on suspect tags.

Replacing an Element. An element may be replaced by another element for a number of reasons (see Figure 1(c)). It can be refined or generalized (e.g., replacing an association by an aggregation or composition and vice versa, or replacing a class by an interface and vice versa). The element can also be converted into a different type (e.g., converting a class into a component and vice versa). Replacing an element requires restoring all former traceability relations on the new element. This means that the nodes of the traceability graphs have to be replaced.

Recognition of change: The replacement of an element is a wider development activity comprised of several elementary changes. At a minimum, the deletion

of the old element and addition of the replacing element are present. We use predefined rules and a rule engine within *traceMAINTAINER* to recognize such development activities, as described in an earlier paper [3].

Required traceability update: In general, the update after a replacement activity requires the restoration of all hanging traceability relations that were related to the impacted model element on the newly created replacing element. Restoration here means to delete all hanging relations of the old element and to recreate them on the new element.

Impact on existing traceability: The replacement element may have an impact on the elements of dependent models requiring an update. Our change propagation mechanism is described in Section 3.2.

Spitting or Merging Elements. An element may be split into two or more elements and two or more elements may be merged into one resulting element (see Figures 1(d) and 1(e)). Examples for split and merge are the extraction of a class's attribute into its own class and vice versa, or the refinement of an unspecified association into two unidirectional ones and vice versa. The splitting or merging of elements requires the duplication of existing traceability relations to these elements. This leads also to the split or merge of nodes within the traceability graph.

Recognition of change: As with replacing an element, split and merge are wider development activities comprised of several elementary changes. We provide rules to recognize these development activities within the rule catalog of *traceMAINTAINER*.

Required traceability update: As for the replacement of elements, there are many ways within a typical CASE tool to get to the same result. This variety is important for split activities since traceability update may require copying all the traceability relations that still exist on the modified element to all additional new elements, with or without the removal of the original. The situation is similar for merging. One of the parts might be modified into the resulting whole or all parts deleted prior to creating a new whole. The traceability update requires consolidation of all traceability relations of all the parts on to the resulting whole.

Impact on existing traceability: The impact is similar to replacing an element. The split or merge may have an impact on related elements of dependent models and require change propagation (Section 3.2).

Moving an Element. An element or one of its parts may be moved into another context, so into a different element (see Figure 1(f)). Examples would be moving a class into another package or moving an attribute into a different class. Changing the context of an element may require copying or moving the traceability relations on the element's parent to the new parent. This change is similar to adding/deleting a node to the traceability graph.

Recognition of change: As discussed for replacing, splitting and merging, the moving of an element is also a wider development activity and can be recognized by predefined rules within *traceMAINTAINER*.

Required traceability update: It is possible to perform this change type in different ways. The element might be dragged and dropped in a project browser from one element to another. In this case, no update of traceability relations on the element itself is necessary. The element may also be deleted and recreated at the new location. In this case, it is necessary to restore all the traceability relations of the deleted element on the newly created element. The more challenging aspect of the traceability update after moving activities such as these is that of changing the context. It might be the case that the moved element was part of a related element. In such a case, it is likely that these relation(s) are impacted by the move, so we provide a dialog to the developer that shows all the traceability relations on the former parent element (old context) and offers (for each relation) to delete it from the old parent and also to create it on the new parent. This approach offers the developer the possibility of either leaving, copying or moving each traceability relation.

Impact on existing traceability: This change type requires propagation, as per Section 3.2.

3.2 Change Propagation

The main purpose of our work is to maintain the ongoing relevance of traceability relations once they have been established and to reduce the manual effort required to do this. It is for this reason that we distinguish between incoming and outgoing relations on a model element. An outgoing relation points towards an independent element; an incoming relation relates from a dependent element. We use this information specifically for the propagation of changes between different models and/or different levels of abstraction. The related elements in such a context usually have an ancestor/successor type of association. If the ancestor element changes, it is likely that this has an impact on related dependent elements, but it should have no impact on the independent related elements. Rather than make assumptions, the developer may be alerted to re-examine such impact. We therefore propagate changes, by default, only on incoming traceability relations to dependent elements. Nevertheless, the developer may also choose to propagate only to independent elements or, in certain cases, to all related elements. The possiblity to propagate changes also to independent elements can be helpful in situations where a model is changed without changing the more abstract model first so that both models stay aligned.

By propagating changes, we mean that we set all incoming and/or outgoing traceability relations of a changed model element to a *suspect* status and require the developer to manually inspect and, if necessary, resolve inconsistencies. We propagate all changes to an element, even those that did not take part in a wider development activity. This makes sense in a forward engineering process where changes will be propagated to subsequent elements whose creation was

influenced by the changed element. To help resolve possible inconsistencies, we capture the recognized change type as a tag on the traceability relation while setting its status to suspect, as mentioned earlier.

We provide a mechanism to resolve relations whose state has been set to suspect. If an element has been modified that has outgoing relations with suspect status then we propose to remove that state after the change. This mechanism reflects our assumption that a developer working on a model element will usually resolve all open issues and ensure consistency to the independent model. Empirical studies are required to examine developers' activities in more detail to ensure the approach is reflective of practical needs.

4 Validation

In this section, we outline an experiment undertaken to explore the following research questions regarding traceability upkeep. Note that validation of the change recognition process has been described elsewhere [3].

Research Question 1. Does use of the *traceMAINTAINER* prototype (a tool that implements the approach and updates traceability relations using automated and semi-automated strategies [7]) reduce the manual effort necessary for maintaining traceability relations? While no manual effort would be a desirable target, we seek evidence of a reduction greater than the time necessary to configure and learn how to interact with *traceMAINTAINER* to make its use worthwhile.

Measure: The manual effort for traceability maintenance refers to the time the developer spends on this task. It comprises: thinking about the maintenance task (including recognizing it), navigating within the models and performing the required changes. Measuring the time taken for these sub-tasks is problematic given the lack of access to thought processes and the inter-weaved nature of many tasks, so attempting to gain this data would interfere with the task itself. As an indicator of effort, we record the number of performed changes to the set of traceability relations, along with time spent responding to *traceMAINTAINER* dialogs.

Research Question 2. Do the traceability updates performed by *traceMAINTAINER* result in a set of traceability relations of comparable quality to those when maintained purely manually?

Measure: Determining the quality of a set of traceability relations depends upon having an agreed baseline. Three types of changes to the traceability relations are then distinguished: changes that have been performed correctly Δ_c (according to the baseline); changes that have been performed incorrectly Δ_i; and changes that have not been performed Δ_m. To be able to compare the quality and number of changes amongst subjects we compute two measures that are commonly used to evaluate approaches dealing with uncertainty in recognition processes, precision and recall. Precision tells us about the quality of the

performed changes, $Q_P = \Delta_c/(\Delta_c + \Delta_i)$ while recall tells us about the number of necessary changes performed, $Q_R = \Delta_c/(\Delta_c + \Delta_m)$.

4.1 Experimental Set-Up

Hypothesis Formulation: Our experiment has one independent variable (the use of *traceMAINTAINER*) and two treatments ($tM, no{-}tM$). It has five dependent variables, on which treatments are compared:

C_m	Number of manually performed changes to the set of traceability relations.
C_a	Number of automated changes to the link-set.
C_{UI}	Number of user interactions with *traceMAINTAINER*.
Q_P	Precision of performed changes (correct changes/(correct+incorrect changes)).
Q_R	Recall of necessary changes (correct changes/(correct+missing changes)).

Null hypotheses:	Manual changes	$H_0 : C_m(tM) \geq C_m(no - tM)$
	Precision	$H_0 : Q_P(tM) = Q_P(no - tM)$
	Recall	$H_0 : Q_R(tM) = Q_R(no - tM)$

Development Project: The experiment was conducted on the UML models for a mail-order system, a completed project implemented in Java by the first author of this paper. The project artifacts include UML models on three levels of abstraction: requirements, design and implementation. The models provide information to a level of detail that one would expect at the end of the design phase, including use case diagrams, interaction diagrams and class diagrams. The model elements are listed in Table 1(a). The set of traceability relations for this project (referred to as the project link-set) relates the three models and consists of 214 traceability relations. The initial linking was undertaken according to a traceability information model for the project. This states that only relations between requirements/analysis and analysis/design are valid (i.e., no intra model relations and no requirements/design). The relations are always directed from the dependent to independent model. Use cases and classes have to be related by at least one relation. Attributes, methods, components and packages can be related to any other element as long as the rules above are followed.

Modeling Tasks: Three maintenance tasks were to be performed on these models in a fixed order, adding new features of practical value that would impact large parts of the system. Although the underlying source code was to be made available within the implementation model, the tasks only required changes to the analysis and design models. It was estimated that it would take 2 to 3 hours to complete all the tasks. Subjects were permitted to perform the tasks according to their ideas and experiences to capture a realistic spread of different solutions to the same problem. This means that the solutions are not comparable per se.

The tasks comprised: (1) Enhance the system's functionality to distinguish private and business customers and to handle different properties for them. Enhance it also to handle foreign suppliers (including currencies and taxes).

Table 1. Development project and subject information

(a) Project models and elements

	RQs	Analysis	Design
Use case diagrams	3		
Class diagrams	1	6	6
Package diagrams		1	1
Activity diagrams	7		
State charts	3		
Sequence diagrams		5	
Package		5	5
Class		41	63
Attribute		73	150
Method		124	280

(b) Prior experience of subjects

	Mean	SD
Programming [years]	11,79	7,58
Languages [count]	4,06	1,70
Projects [count]	2,47	1,29
Projects [days]	448,88	397,20
UML [1-4]	2,80	0,91
CASE tools [1-4]	3,16	0,85
Sparx EA [1-4]	2,81	0,69
Traceability [1-4]	2,05	0,75

(Scale [1-4] – 1 is low and 4 is high)

(2) Convert two parts of the system into separate components. (3)Enhance the functionality to categorize different products.

Subjects: The subjects comprised 16 computer science students with a wide range of experience in UML and model-based software engineering. All the students were taking a course on software quality and were either in the 4^{th} or 5^{th} year of their diploma (Masters comparable). The subjects were partitioned into two groups of 8 (tM and $no-tM$), to equally distribute expertise based upon prior experience (see Table 1(b)).

Experimental Procedure:

1. All subjects completed a questionnaire to capture their background and experience. The answers were used to divide the subjects into two groups.
2. All subjects were asked to install the CASE tool to be used (Sparx Enterprise Architect) and to follow a tutorial one week prior to the experiment.
3. All subjects completed a second questionnaire to capture the effort they spent on learning the CASE tool.
4. All subjects spent 30 minutes on a lesson explaining the general structure of the project's UML models and the purpose of the system. The subjects were introduced to the advantages and problems of traceability, the project's traceability information model and how to maintain traceability relations manually within the CASE tool. The subjects of the tM group also received an introduction to the purpose and required responses to requests for user interactions when using *traceMAINTAINER*.
5. All subjects received the description of the three tasks along with a questionnaire that would be used to gather information about the work completed on each task. Only the subjects of the $no-tM$ group were asked to maintain the traceability relations along with undertaking the modeling activities.
6. After 120 minutes, the subjects of the tM group were asked to stop the modeling work and to manually maintain the traceability relations.
7. After 150 minutes, all the subjects stopped work.

8. Each subject participated in a short final interview to gather data on the perceived usefulness of traceability to the tasks, the problems they experienced with traceability maintenance and suggestions they had on improving tool support for traceability.

Data Gathering: Data were gathered via the three questionnaires, as described above. For both groups, a log file was created by *traceMAINTAINER* containing all the elementary changes performed by the subject, all changes to the link-set and information about how often the subject navigated the models using traceability relations. For the *tM* group, a log of all recognized development activities, the user decisions on interactions and all automatically performed traceability updates was also created. The models of all the subjects were available for analysis and the participants of the *tM* group were asked to save their model before the 30 minute manual traceability maintenance period.

4.2 Results

Univariate analyses of the dependent variables were performed to test the hypotheses both individually for each task and across all tasks. For all dependent variables C_m, C_a, C_{UI}, Q_P and Q_R, two-sample t-tests were performed. The level of significance for the hypotheses tests was set to $\alpha = 0.05$. We provide p-values in the t-test columns of Table 2(a) and 2(b). We had to exclude one subject from each group from the analysis, because they did not provide a minimal solution to each modelling task. This precondition was required in order to compare results between all subjects.

Table 2. Descriptive statistics

(a) Change actions and interactions

Task	Var	Treat	Mean	SD	% diff	t-test
All	C_m	no-tM	36.3	12.8	-82%	0.00
		tM	6.6	3.8		
	C_a	tM	59.6	34.7		
	UI	tM	9.0	4.1		
1	C_m	no-tM	18.6	9.5	-87%	0.00
		tM	2.4	1.8		
	C_a	tM	36.2	23.9		
	UI	tM	2.0	2.0		
2	C_m	no-tM	7.7	4.9	-90%	0.01
		tM	0.8	1.8		
	C_a	tM	13.0	3.3		
	UI	tM	6.2	4.9		
3	C_m	no-tM	10.0	4.7	-66%	0.04
		tM	3.4	4.7		
	C_a	tM	12.4	14		
	UI	tM	0.8	0.8		

(b) Precision and recall of changes

Task	Var	Treat	Mean	SD	% diff	t-test
All	Q_P	no-tM	79.5	25.7	21%	0.19
		tM	95.9	4.3		
	Q_R	no-tM	71.3	27.6	11%	0.61
		tM	78.8	19.0		
1	Q_P	no-tM	78.9	36.5	21%	0.34
		tM	95.7	6.0		
	Q_R	no-tM	78.0	36.3	-6%	0.82
		tM	73.3	32.7		
2	Q_P	no-tM	83.3	31.0	19%	0.29
		tM	98.9	2.5		
	Q_R	no-tM	59.5	35.3	51%	0.1
		tM	90.0	14.1		
3	Q_P	no-tM	81.6	37.5	17%	0.44
		tM	95.6	6.5		
	Q_R	no-tM	76.2	35.8	-11%	0.67
		tM	67.8	27.3		

4.3 Discussion

Research Question 1. When looking at the number of manual changes C_m to the link-set over the three tasks, the tM group performed far fewer changes (82%) than the $no-tM$ group (see Table 2(a)). This difference is statistically significant. However, it is evident that the $no-tM$ group performed only half as many changes (36.3) than the tM group's combined total of manual and automated changes (6.6+59.6=66.2). There are two reasons for this. First, *traceMAINTAINER* recognizes small incremental change activities and updates traceability relations immediately (in the background) after recognition. This means that the link-set reflects each detour of the developer, in contrast to manual maintenance where the update is typically performed after completing the whole task, resulting in fewer changes. Second, manual maintainers often chose to perform only the minimum required changes to comply with the traceability information model.

The time to undertake a manual change could not be measured precisely because it is not clear when the developer starts to think about a change task. To estimate this indirectly, we counted the manually created relations, manually deleted relations and user interactions. Based upon several measurements with different subjects' data, we correlated the comparable effort of the three direct measures as follows: $T_{create} \approx 2 * T_{delete} \approx 2 * T_{UI}$. Using this, we compared no-tM ($T_{create} + 0.5 * T_{delete}$) and tM ($T_{create} + 0.5 * T_{delete} + 0.5 * T_{UI}$) to gain an approximation of the saved effort by using our approach, as shown in Table 3.

Table 3. Manual effort of the tM group, compared with the $non-tM$ group

Task	All	1	2	3
Approx. effort tM	-71%	-82%	-48%	-67%

Research Question 2. The values of Q_P and Q_R, information about the precision and recall of changes to the link-set (see Table 2(b)), show that the tM group reached a value of over 95% precision for all tasks, with a low standard deviation, 21% higher than the value for the $no-tM$ group. Among all the changes performed by *traceMAINTAINER* we found no incorrect ones; all the incorrect changes within the tM group were manually performed changes. The values for recall are lower than those for precision and, except for Task 2, comparable between both groups. Values of recall lower than 100% indicate that more changes to the link-set would have been necessary. This means that the approach performs updates with high quality (high precision), but does not perform all the necessary updates. However, any differences between the two groups for both measures are not statistically significant. The quality of the changes performed by both groups is comparable.

4.4 Threats to Validity

External Validity: From a task perspective, the reported experiment is realistic. We had young professionals working on a real project, using commercial tools

and implementing demanding tasks. Nevertheless, it is hard to draw conclusions to a wider population without more studies. The results reflect more a tendency that shows the potential of our approach. There are threats associated with the short time the subjects spent on the experiment given the task complexity. However, we did not want to set trivial tasks with obvious changes. A high-level task description enabled there to be a variety of ways to solve the problem, but demanded effort to analyze and evaluate the data (55k lines of log messages and 16 different models). Without sophisticated techniques, it would be complicated to run an experiment lasting much longer.

Internal Validity: Internal validity is concerned with establishing a causal relationship, here between the use of *traceMAINTAINER* and the number of manual changes to the link-set. Subjects were randomly assigned to the groups to balance expertise, but in order to have comparable results among participants we had to exclude two subjects from the experiment since they did not solve the modeling tasks sufficiently. The potential influence of the facilitators was addressed by providing an initial briefing and task description in written form only. The difference in the material between groups was marginal, the addition being how to react on user interaction for the *tM* group. None of the subjects had any prior knowledge about the approach nor did they know the experimental goals.

Construct Validity: Construct validity refers to having established correct operational measures for the constructs being studied. To investigate the effect of our approach on the effort for maintaining traceability relations after the evolution of related UML models it is necessary to use the UML as intended. The UML offers an open set of description techniques with many ways to apply them. In this experiment, we used six types of diagram at different levels of detail, our subjects had state-of-the-art education in UML development and we used a widely distributed CASE tool. From the debriefing interview, we learned that almost all the subjects felt immediately familiar with the tool after their prior tutorial. The examination of the resulting models showed that, except for two cases (explained above), all the subjects were able to edit and enhance the UML models in a manner comparable to industrial practice. To investigate the quality of changes to the link-set, the main problem with comparing traceability is the lack of an agreed standard. We therefore provided a traceability information model to give guidance on how to establish traceability for the project. We did the initial creation of traceability relations according to the model and required our subjects to do likewise. In order to gain comparable results, we made further restrictions as to the minimal number and direction of relations (see Section 4.1).

5 Related Work

The goal of our approach is to support the maintenance of already established traceability relations. We are not concerned with creating an initial set of relations, which is mostly the domain of techniques based on information retrieval and data mining, and are concerned more with incremental additions to an

evolving set. There is related work on maintaining traceability relations, supporting inconsistency management and change propagation between models.

Maletic et al. [9] describe an XML-based approach to support the evolution of traceability relations between models. The authors describe a traceability graph and its representation in XML, independent of specific models or tools. They discuss the issue of evolution and propose to evolve traceability along with the models by detecting syntactic changes at the same level and type as the relations. However, the authors do not discuss how to detect these changes in depth nor how to update the impacted traceability relations.

Murta et al. [10] describe an approach called ArchTrace that supports the evolution of traceability relations between architecture and implementation. The use of xADL for the description of architectures and Subversion for the versioning of source code is required. The authors trigger a set of eight policies on committing a new version of an artifact. These policies mostly ensure the update of existing traceability relations on artifacts to new versions within the version control system and further restrict the creation of new relations on old artifacts. The authors do not discuss the recognition of structural changes to supported models nor how to update relations in this case.

Mens et al. [11] describe an extension to the UML meta-model to support the versioning and evolution of UML models. The authors classify possible inconsistencies of UML design models and provide rules to detect and resolve these. They transform the models into a supported format, apply their rules and suggest model refactorings based on the results. While the authors discuss the necessity for traceability management and change propagation while UML models are evolving, they provide no support for this.

Cleland-Huang et al. [12] present an approach called event-based traceability (EBT). The authors link requirements and other artifacts of the development process through publish-subscribe relationships. Changes to requirements are categorized by seven kinds and events are raised according to kind. Events are published to an event server that sends notifications about the change to subscribers. Change is propagated through sending messages to stakeholders. The notification contains information to support the update process of the dependent artifacts. This facilitates manual maintenance. The EBT approach is similar to our approach, though the authors mainly focus on requirements models and do not discuss how to detect and resolve changes to additional UML models.

Olsson and Grundy [13] describe an approach where they extract key information from different artifacts (requirements specifications, use cases and tests) into abstracted representational models. The developer can then create explicit relations between the abstract elements. Some implicit relations can be defined automatically (e.g., consistently named users within different artifacts). Through this mechanism, changes can be propagated. Some changes can be resolved automatically (e.g., changing the name of a user). For others, developers are informed so they can take action. Like Olsson and Grundy, we also propagate the change of a traced model element and maintain the traceability relations of evolving

model elements, in some cases automatically and in others with limited developer interaction. In contrast, we do not need to extract the data from the models first and provide the propagated information about change within the model.

Grundy et al. [14] review existing approaches to handle inconsistencies, between analysis, design and implementation specifications. They also outline requirements for effective inconsistency management and provide exemplars to demonstrate and evaluate their approach. We tried to follow their requirements with our approach, so the necessity to propagate changes immediately as they occur and mechanisms to inform the developer about inconsistencies.

Traceability is supported by many commercial requirements management tools, enabling the tracing of requirements to other artifacts in the software development life cycle. One example, IBM's RequisitePro, allows developers to relate requirements kept within the tool to other tools in the product suite, such as Rational Software Modeler. While these tools support UML explicitly, there is limited support for the automated creation or maintenance of traceability relations at fine-grain levels. To integrate the approach of this paper, it would be necessary to write a tool-specific adapter to generate the necessary events, and to be able to create and delete the traceability relations. We have found this to take from several days to two weeks based on the particular tool.

6 Conclusions and Future Work

This paper addresses the problem of traceability decay by presenting an approach for the maintenance of traceability relations. The approach is currently limited in scope such that it focuses on restoring traceability following changes to related elements within structural UML models while undergoing model-driven software development. The paper provides a set of potential change types and described the necessary update to existing traceability relations that each type demands. The identification of these change types becomes possible by applying development activity recognition rules to elementary change events captured while working within a CASE tool [3] and the approach is supported by a prototype tool called *traceMAINTAINER* [7].

By recognizing each elementary change to a model, it is much easier to solve small incremental problems associated with maintaining traceability. Through our rules and identified change types, this is what we do and we achieve encouraging results. We have conducted preliminary studies to examine the effectiveness of the traceability update process in practice, in terms of effort saved and quality of the end results. We are currently starting a larger longitudinal industrial case study to evaluate our work further. Within that study we plan to gain more statistical data on the cost/benefit trade-off of the approach for practical application.

Acknowledgments. This work is part funded by DFG grant Ph49/7-1. The authors thank Johannes Langguth for his work in preparing and analyzing the experiment and all the students who were involved.

References

1. Hayes, J.H., Dekhtyar, A., Sundaram, S.K.: Advancing candidate link generation for requirements tracing: The study of methods. IEEE TSE 32(1), 4–19 (2006)
2. Mäder, P., Gotel, O., Philippow, I.: Rule-based maintenance of post-requirements traceability relations. In: Proc. 16th Int'l Requirements Eng. Conf., Barcelona, Spain (September 2008)
3. Mäder, P., Gotel, O., Philippow, I.: Enabling automated traceability maintenance by recognizing development activities applied to models. In: Proc. 23rd Int'l Conf. on Automated Software Engineering ASE, L'Aquila, Italy (September 2008)
4. Lano, K.: Advanced systems design with Java, UML, and MDA. Elsevier Butterworth-Heinemann, Amsterdam (2005)
5. Egyed, A., Grünbacher, P., Heindl, M., Biffl, S.: Value-based requirements traceability: Lessons learned. In: Proc. 15th Int'l Req. Eng. Conf., pp. 115–118 (2007)
6. Object Management Group Framingham, Massachusetts: OMG Unified Modeling Language Specification (Version 2.1.2) (November 2007)
7. Mäder, P., Gotel, O., Kuschke, T., Philippow, I.: traceMaintainer – Automated Traceability Maintenance. In: Proc. 16th Int'l Requirements Eng. Conf., Barcelona, Spain (September 2008)
8. Gotel, O.C.Z., Finkelstein, A.C.W.: An analysis of the requirements traceability problem. In: First Int'l Conf. on Req. Eng. ICRE, pp. 94–101. IEEE CS Press, Los Alamitos (1994)
9. Maletic, J.I., Collard, M.L., Simoes, B.: An xml based approach to support the evolution of model-to-model traceability links. In: Proc. TEFSE 2005, pp. 67–72. ACM, New York (2005)
10. Murta, L.G.P., van der Hoek, A., Werner, C.M.L.: Archtrace: Policy-based support for managing evolving architecture-to-implementation traceability links. In: 21st Int'l Conf. on Automated Software Engineering ASE, September, pp. 135–144 (2006)
11. Mens, T., van der Straeten, R., Simmonds, J.: A framework for managing consistency of evolving UML models. In: Yang, H. (ed.) Software Evolution with UML and XML, pp. 1–30. IGI Publishing, Hershey (2005)
12. Cleland-Huang, J., Chang, C.K., Christensen, M.J.: Event-based traceability for managing evolutionary change. IEEE TSE 29(9), 796–810 (2003)
13. Olsson, T., Grundy, J.: Supporting traceability and inconsistency management between software artefacts. In: Int'l Conf. Software Eng. and Appl. (November 2002)
14. Grundy, J.C., Hosking, J.G., Mugridge, W.B.: Inconsistency management for multiple-view software development environments. IEEE TSE 24(11), 960–981 (1998)

Temporal Extensions of OCL Revisited

Michael Soden and Hajo Eichler

Department of Computer Science, Humboldt University Berlin
Unter den Linden 6, 10099 Berlin, Germany
{soden,eichler}@ikv.de

Abstract. Temporal extensions of OCL have been proposed in the literature in order to express dynamic system properties of UML models. This paper reviews previous work on Temporal OCL based on dynamic, state-oriented behaviour specifications and gives a more general definition for *Linear Temporal OCL* (LT-OCL) for languages that are defined using MOF metamodels in conjunction with operational semantics. The definitions presented in this paper intend to pave the way for precise semantics of temporal OCL constraints of languages defined by other metamodels than UML.

1 Introduction

The Object Constraint Language (OCL) has become a de-facto standard for expressing static constraints over object-oriented UML design models [1]. In conjunction with the formalization of the dynamic semantics of UML, there have been proposals to extend OCL towards *temporal logic* to precisely specify dynamic properties of model behaviour. Driven by the need of various domains that adopt UML as a specification language, several variants of 'Temporal OCL' have been studied based on state-oriented dynamic semantics of UML models [2] [3] [4] [5]. A general problem, however, is the missing precise semantics of UML in conjunction with its semantic variation points which leads in practice to assumptions on the platform that will execute the modelled system. For this reason, formalizations of OCL including temporal extensions are usually bound to a fixed notion of *system states*. Moreover, application of OCL in the context of domain specific languages (DSLs) that are defined over other metamodels than UML including their own behavioural semantics are not considered.

This paper explores an approach to define linear temporal constraints for languages defined over MOF metamodels in conjunction with operational semantics. We use the M3Actions [6] [7] to give models executable behaviour through action semantics and define the interpretation of *Linear Temporal OCL* (LT-OCL) formulas over states induced by these definitions. For this purpose, the Essential OCL (EOCL) subset of the standard [1] is extended in a similar way than previous work from Ziemann et al. [2]. Thus, the semantics of temporal constraints can be applied for DSLs solely by a language metamodel including behaviour definition. We demonstrate the approach along with a timed state-machine language

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, pp. 190–205, 2009.

that models the steam-boiler control system [8]. Originated from the Dagstuhl-Meeting in 1995, the system specification has become quickly a widely known reference example in the area of safety-critical systems and verification methods.

The remainder of this paper is organized as follows. Section 2 introduces the concepts of Linear Temporal Logic in general as basis for assessment of our and related work discussed in section 5. Syntax and semantics of LT-OCL are introduced in section 3 along with an overview on the action semantics used in subsection 3.2. Afterwards, the approach is applied to an example in section 4 before section 6 draws some conclusions and summarizes future work.

2 Linear Temporal Logic

Linear Temporal Logic (LTL) is usually defined as an extension of propositional logic by *temporal operators*. Semantics are given as interpretation over sequences of states, typically in terms of Kripke structures that specify in which states propositions hold [9]. In the extent of this paper, we have to deal with an enhanced variant of many-sorted (first-order) predicate logic with temporal operators which we will briefly introduce in this section.

The various definitions of temporal predicate calculi found in literature can be roughly classified based on their access to time [10]. While *explicit* time access introduces time as a special sort (commonly with additional strict monotonic property), *implicit* modelling of time introduces a set of temporal operators. In order to express future directed[1] statements, commonly used operators are *always* (denoted $'\Box'$), *next* (denoted $'\bigcirc'$), *until* (denoted $'\mathcal{U}'$) and *eventually* (denoted $'\Diamond'$). For example, one can express a simple liveness property of a system *'all messages being sent are received'* by:

$$\Box(\forall msg(send(msg) \rightarrow \Diamond(receive(msg)))) \tag{1}$$

From a syntactical point of view, this expression assumes that there exists some predicates *send* and *receive* over a sort that is compatible with variable *msg*. Note how the variable *msg* is bound by the $\forall$ quantifier and identifies a set of objects which — whenever the predicate *send* is true — must fulfill the predicate *receive* some time *later*. A more strict message receival would be specified by the formula:

$$\Box(\forall msg(send(msg) \rightarrow \bigcirc(receive(msg)))) \tag{2}$$

Due to the next operator, an immediate receival of messages is required *in the next state*.

For the precise semantics of such formulas, *a model* is required over which formulas are interpreted and the notion of truth is defined. Traditionally, Kripke structures provide a universe of states, transitions and a labeling function that specifies which propositions hold when. The temporal semantics at its core are defined by evaluation of formulas over state sequences which are paths through

[1] in opposite to past directed operators, cf. also to the discussion in section 5.

the Kripke structure. For our purpose, we define the semantics by an evaluation over state sequences $\pi = s_0, s_1, s_2, \ldots$ directly. Thereby, all functions/predicates are interpreted on a (set of) states as specified by the temporal operators. For the example (1), the state sequence

$$s_0 : send(m_1) \longrightarrow s_1 : send(m_2) \longrightarrow s_2 : receive(m_2) \longrightarrow s_3 : receive(m_1)$$

is a valid model since all messages are received even though in a different order. However, this sequence is not a model for formula 2. We refrain from a formal definition of syntax and semantics for this example and call upon the intuition of the reader. Instead, we'll focus on the definitions of temporal operators for OCL in the next section.

3 Linear Temporal OCL

This section defines syntax and semantics of LT-OCL expressions. Conceptually, we introduce three new unary operators `next`, `always` and `eventually` plus a binary operator `until`. Intuitively, they correspond to the temporal operators '$\bigcirc$', '$\Box$', '$\Diamond$' and '$\mathcal{U}$' as outlined in section 2. Since formulas of LT-OCL must be interpreted in the context of a metamodel and its dynamic behaviour specified by action semantics, we define the semantics over a generic trace metamodel that conceptually replaces the state sequences.

3.1 Syntax Extensions for Temporal Operators

The concrete syntax of OCL is defined by means of an attributed EBNF grammar including a mapping to the abstract syntax defined as MOF metamodel (cf. section 9 of [1]). Thus, we define the extensions of LT-OCL by extending the grammar rules and the abstract syntax metamodel in the following.

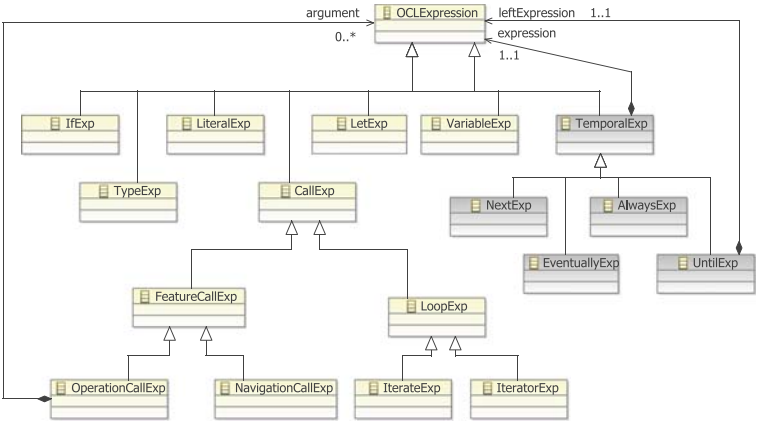


Fig. 1. Abstract Syntax extensions for temporal operators

For the four temporal operators, the OCL metamodel extension is shown in figure 1 (extensions marked grey). The changes to the OCL grammar are as follows. On top level, we add a new production rule `[G]` for temporal expressions:

```
[A] OclExpressionCS ::= PropertyCallExpCS
[B] OclExpressionCS ::= VariableExpCS
[C] OclExpressionCS ::= LiteralExpCS
[D] OclExpressionCS ::= LetExpCS
-- [E] OclExpressionCS ::= OclMessageExpCS
-- not in Essential OCL
[F] OclExpressionCS ::= IfExpCS
[G] OclExpressionCS ::= NextExpCS | AlwaysExpCS |
                        EventuallyExpCS | UntilExpCS
```

The new four non-terminals are then defined by the production rules:

```
[A] NextExpCS ::= 'next' '(' OclExpressionCS ')'
[B] AlwaysExpCS ::= 'always' '(' OclExpressionCS ')'
[C] EventuallyExpCS ::= 'eventually' '(' OclExpressionCS ')'
[D] UntilExpCS ::= '(' OclExpressionCS[1] ')' 'until'
                              '(' OclExpressionCS[2] ')'
```

Furthermore, the OCL specification uses a synthesised attribute `ast` to provide access to the abstract syntax objects returned by each rule. In that way, the AS-tree construction and mapping is defined. To exemplify this mechanism, we define the type and tree assignment for `NextExpCS` and `UntilExpCS`:

```
NextExpCS.ast : NextExp
[A] NextExpCS.ast.expression = OclExpressionCS.ast
[...]
UntilExpCS.ast : UntilExp
[D] UntilExpCS.ast.leftExpression = OclExpressionCS[1].ast
[D] UntilExpCS.ast.expression = OclExpression[2].ast
```

Intuitively, as can be seen from the definitions above, the `ast` instantiates the metamodel classes `NextExp`, `UntilExp`, etc. and the property `expression` is set to the constructed value for `OclExpressionCS`. For the `until` expression, the left hand side is assigned to `leftExpression` while the right hand side is stored as `expression`. As additional constraint for all four temporal expressions, we have to limit expressions to type *Boolean*:

```
NextExpCS.ast.expression.type.name = 'Boolean'
AlwaysExpCS.ast.expression.type.name = 'Boolean'
EventuallyExpCS.ast.expression.type.name = 'Boolean'
UntilExpCS.ast.leftExpression.type.name = 'Boolean'
```

3.2 Semantics of LT-OCL

As outlined in section 1, the semantics of LT-OCL expressions is dependent on a concrete metamodel and its operational semantics. While metamodels specify

how models are structured, operational semantics define the models' behaviour and as a direct consequence the notion of *states* of the models. For this purpose, we use the M3Actions framework that supports the definition of action semantics for MOF metamodels [11]. The framework has been successfully applied to model a number of languages including aspects of SDL, ASM, C# or Petri-Nets ([6] [7]). The current implementation for EMF is on the verge of becoming the basis for the Model Execution Framework (MXF) of Eclipse [12].

M3Action framework. The core of the M3Actions framework is the *MAction* language that defines behaviour with a graphical syntax similar to UML Activities/Actions, but with a precise update semantic defined for instances of MOF metamodels [13]. Action flows specifying the operational behaviour are hooked into a metamodel as specific *MOperations*. Elementary actions include for example an OCL query action to navigate the model, an assign action to update properties, a create action to instantiate a meta-class, an invocation action to trigger other flows, and so on.

Flows are always executed in the context of a `MThread`. While parallel execution is supported through forking at invocation actions, we explicitly constrain the operational semantics to be 'single-threaded' for LT-OCL[2].

One key aspect in the M3Actions architecture is the differentiation of static model structures (abstract syntax of a language) from its evolving runtime configurations: the *runtime model*. Runtime models are considered to be instances of the abstract syntax which can be expressed by an explicit *instanceOf* relation between meta-classes. We will explain more details upon the language and architecture along with the example of the steam-boiler application in section 4.

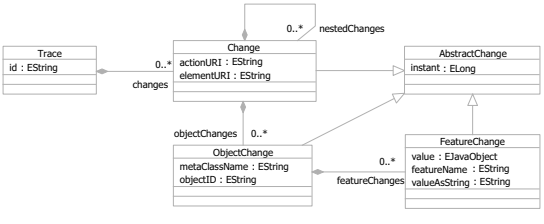


Fig. 2. Generic trace metamodel defining model changes

State Sequences by Execution Traces. For the design of a behavioural language the notion of *observable states* is key when it comes to the analysis of execution runs. We define the notion of *states* for an executed model as the states of its runtime model. Since actions change the runtime model, a new state is reached whenever an action (that updates the model) has finished execution.

States manifest themselves in *execution traces*, which can be recorded as instances of a generic trace metamodel shown in figure 2. The structure of a trace consists of an instance of class `Trace` as top level element created per running

[2] cf. also to the discussion on parallelism in section 5.1.

`MThread` (cf. section 3.2) and a contained list of changes. Such `Change` objects in turn consist of `ObjectChanges` and `FeatureChanges` holding the delta values of a change. Since every object is tagged by a unique (simulation) time stamp number `instant` (cp. class `AbstractChange`), all manipulations are globally ordered. Although the trace contains only changes, one can consider each change to define a new state[3].

In most cases, observable states of the designed language do not match the 'micro-states' of the operational actions. In order to control the creation of states in the trace, MActions provide two concepts:

1. *Atomic Groups* combine multiple actions into a new *atomic* one that is prevented from being interrupted. They produce only a single change-object in the trace[4].
2. *State Generating Transitions (SGTs)* provide a way to determine when a new state is reached. In other words, traversing an SGT force all previous changes to be dumped into the trace.

From the application of these concepts it follows that the recorded traces reflect the observable states of the executed model. In other words, to create or restore a state explicitly, one has to apply all changes in the order given by `instant` to the initial (empty) runtime model[5]. In the following, we refer to these states as the states *induced* by a sequence of changes recorded in an execution trace.

Definition 1 (Induced States). *Let* $\gamma_0, \ldots, \gamma_n (n \in \mathbb{N})$ *be the globally ordered sequence of instances of class* `Change`*, where* n *denotes the value of attribute instant. We denote an object model* $\sigma_i(\gamma_i)$ *with* $i \leq n$ *to be the* induced state *of* γ_i *constructed by applying all changes* $\gamma_0 ... \gamma_i$ *to the empty runtime model. Synonymously, we refer to this constructed runtime model of an induced state as the* snapshot *at instant* i*.*

Yet we have an intuitive notion of system states as models in the sense of MOF's instance model. In order to define the semantics for LT-OCL, we require a more formal definition of this object model. In the OCL standard, the semantics are defined by a mapping of the abstract syntax onto a domain model, both expressed in UML (cf. [1], section 10). We follow the formal approach included as (non-normative) Appendix A of the OCL standard and reuse those definitions of the *object model* consisting of:

$$\mathcal{M} = (\textsc{Class}, \textsc{Att}_c, \textsc{Op}_c, \textsc{Assoc}, \text{associates}, \text{roles}, \text{multiplicities}, \prec) \qquad (3)$$

where Class is a set of classes, $\textsc{Att}_c$ the set of operation signatures that associate a class c to its attributes, $\textsc{Op}_c$ the operations of class c, Assoc the set of association names, *associates* a mapping of association names to participating

[3] we will return to the beneficial value of knowing the change in section 5.1.
[4] except if nested actions are connected by SGTs.
[5] more precisely, we could define a transformation that derives a state sequence from this change-based trace model.

classes, *roles* and *multiplicities* the mapping of role names and multiplicities to association ends and $\prec$ a function defining the generalization hierarchy of the classes (cp. [1], appendix A, section A.1.1.7). Furthermore, the standard defines a *system state* for a model as structure $\sigma(\mathcal{M})$ as follows:

$$\sigma(\mathcal{M}) = (\sigma_{CLASS}, \sigma_{ATT}, \sigma_{ASSOC}) \tag{4}$$

This tuple consists of the set of objects σ_{CLASS}, a value to attribute mapping σ_{ATT} and a finite set σ_{ASSOC} containing the links between objects[6]. In combination with definition 1, we interpret all runtime snapshots $\sigma_i(\gamma_i)$ in the following as objects reflected in a structure $\sigma_i(\mathcal{M})$ at moment in time i.

Finally, we reuse the syntactical expressions $Expr_t$ (set of all OCL terms), assuming the extensions introduced in section 3.1 have been applied. Hence, an interpretation of expressions is performed in an extended evaluation environment $\varepsilon = (\pi, \beta, i)$ consisting of a sequence of states π, a variable projection $\beta : Var_t \times \mathbb{N} \rightarrow I(t)$ that derives the values from a state at an instant of time i. For readability, we abbreviate $\varepsilon = (\pi, \beta, i)$ as ε_i if π and β are clear from the context.

Definition 2 (Semantics of LT-OCL expressions). *Let $\pi = \sigma_0, \ldots, \sigma_n$ $(n \in \mathbb{N})$ be a sequence of induced states, $e \in Expr_t$ an LT-OCL expression and Env the set of environments $\varepsilon = (\pi, \beta, i)$. Then evaluation of a LT-OCL expression is defined by the function $I[[e]] : Env \rightarrow I(t)$ as follows:*

i. Standard OCL expressions evaluate as described in the semantics part of [1], Appendix A, Definition A.14 - A.30, [7] *with the enhanced environment $I[[e]](\varepsilon)$ that carries the time instant i. For example, this implies:*
   (1) Variable assignment retrieves the value from the environment at instant i: $I[[v]](\varepsilon) = \beta(v, i)$
   (2) Iterator expressions iterate over a consistent collection that was retrieved at instant i
   (3) Evaluation of operation call expressions are considered to take no time and be side-effect free. Hence, evaluation is consistent at instant i, etc.

ii. 'next' expressions evaluate in the next state of the path:

$$I[[next(e)]](\varepsilon) = \begin{cases} true & \text{if } I[[e]](\varepsilon_{i+1}) = true \\ false & \text{if } I[[e]](\varepsilon_{i+1}) = false \\ \bot & otherwise \end{cases}$$

iii. 'until' force the expression e_1 evaluate to true in all states until the second expression e_2 becomes true:

$$I[[(e_1)\ until\ (e_2)]](\varepsilon) = \begin{cases} true & \text{iff } \exists k.i < k \text{ so that } \forall i \leq u < k : I[[e_1]](\varepsilon_u) = true \\ & \text{and } \forall j \geq k : I[[e_2]](\varepsilon_j) = true \\ false & \text{iff for one of } i \leq u < k\ I[[e_1]](\varepsilon_u) = false \text{ or} \\ & I[[e_2]](\varepsilon_j) = false \\ \bot & otherwise \end{cases}$$

[6] cp. [1], appendix A, Definition A.12 (System State).
[7] we refrain from repeating the altered definitions in the extent of this paper because of limited space.

iv. 'always' expressions are true iff they hold in all states of the path:

$$I[[always(e)]](\varepsilon) = \begin{cases} true & if\ \forall i \in 0,...,n.I[[e]](\varepsilon_i) = true \\ false & if\ \exists i \in 0,...,n.I[[e]](\varepsilon_i) = false \\ \bot & otherwise \end{cases}$$

v. 'eventually' expressions iff there exists a state along the path where it evaluates to true:

$$I[[eventually(e)]](\varepsilon) = \begin{cases} true & if\ \exists i \in 0,...,n.I[[e]](\varepsilon_i) = true \\ false & if\ \forall i \in 0,...,n.I[[e]](\varepsilon_i) = false \\ \bot & otherwise \end{cases}$$

We say an LT-OCL expression $e \in Expr_t$ ***is true on*** *π if $I[[e]](\pi, \beta, 0) = true$.*

As can be seen from the definition, the semantics of an LT-OCL formula is defined in the scope of a single trace, which reflects one execution run of a model. Therefore, we denote a LT-OCL expression $e \in Expr_t$ as true with respect to a model behaviour, iff e is true on *all* possible traces. This corresponds directly to an implied Kripke structure, which consists of all possible states that a might be entered during model execution.

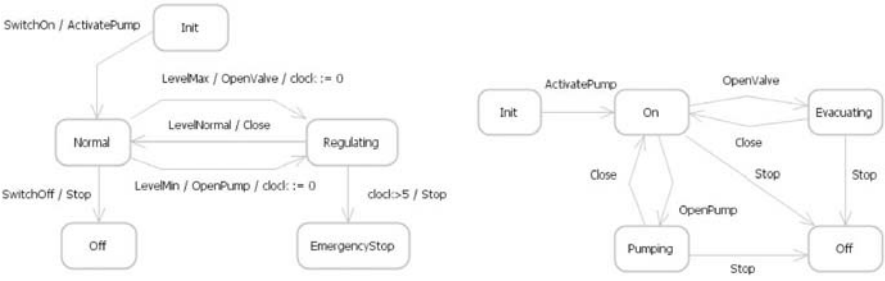
LT-OCL application context. The Essential OCL subset defined in the standard does not make an explicit statement about the contextual element to which an OCL expression can be attached (cp. [1]). As consequence of the package merge of the MOF metamodel, we assume that the `context` definitions remain valid. For usage of LT-OCL expressions, we restrict the scope of application to invariants of classifiers.

4 Example: SteamBoiler Control System

In this section we illustrate the application of LT-OCL along with a well-known example: the steam-boiler control system [8]. Within this paper, we consider a simplified version of the steam-boiler consisting of one *controller program* and one *pump*[8] that are modelled using a state-machine language. During normal operation, the task of the controller is to regulate the water supply of the steam-boiler tank in order to maintain the water level in a certain range while the water is evaporating. This is achieved by switching the pump/valve on or off[9]. Important for proper system operation is the safety constraint of the steam-boiler which says that the water level must not leave the normal range for more than 5 seconds. Otherwise, the system must be halted. Figure 3 shows the state-machine model of both components.

[8] the original system specification defines four pumps, an additional maintenance-mode and diagnostic capabilities for failure detections which are omitted here for simplicity, cf. to [8] for details.

[9] originally, the valve is only used for initial evacuation of the tank.



(a) Controller (b) Boiler

Fig. 3. The steam-boiler control system as two state machines: 3(a) controller program, 3(b) pump

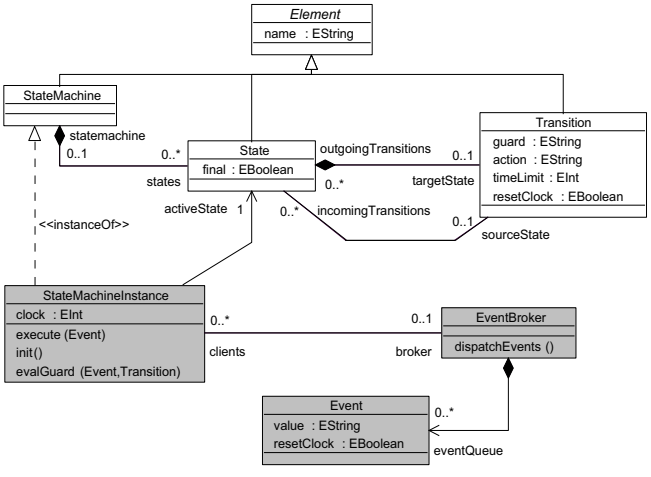


Fig. 4. Simple StateMachine Metamodel

The state machines are defined by the metamodel shown in figure 4. We use the standard UML notation as syntax for states and transitions with some refinements for labels at transitions. Labels are made up of triples of the form:

```
<guard> / <action> / <clock-reset>
```

where `<action>` and `<clock-reset>` are optional. The values of `<guard>` and `<action>` are mapped directly to the corresponding attributes of meta class `Transition` with the following exceptions:

(1) `<clock-reset>` might only be the value `'clock := 0'` which maps to `resetClock=true` (otherwise `resetClock` is `false`)
(2) any `<guard>` starting with `'clock'` simply describes the value of the `timeLimit` attribute

These limitations have been introduced solely for reasons of simplicity of the example[10]. Conceptually, each state-machine has a single, local clock. All clocks are continuously increasing over time in discrete steps. A transition may reset the machine's clock through `clock := 0` or react upon changes only by specifying a time limit as guard condition, describing *when* the transition shall fire.

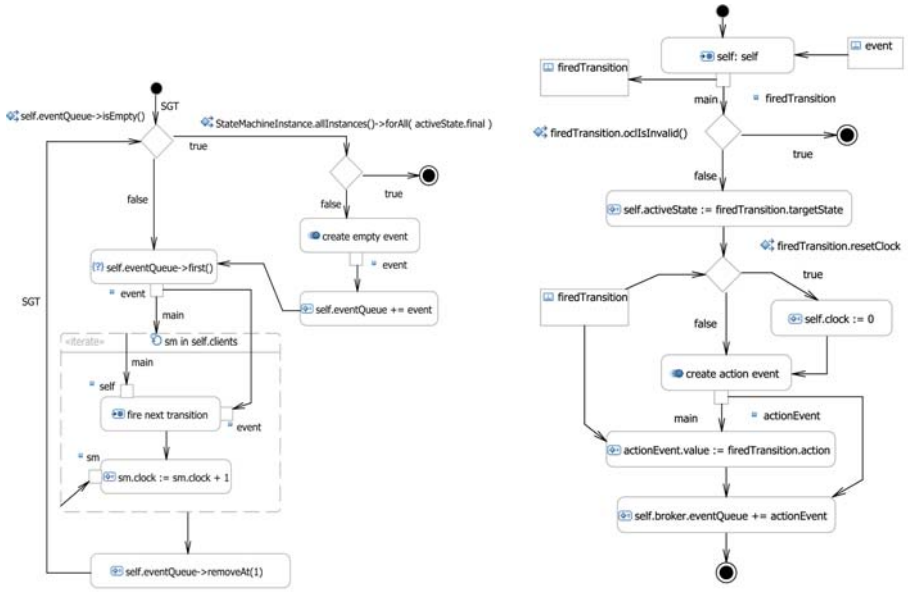
Intuitively, the controller in figure 3 will operate in `Normal` mode upon receiving a `SwitchOn` event which subsequently emits an `ActivatePump` event to start the pump unit. Whenever a `LevelMin` or `LevelMax` event is received describing the occurrence of a water level measurement exceeding the limits, the controller starts to actively regulate the level by sending a `OpenValve` or `OpenPump` event. At the same time, the machine's clock is reseted to measure the time spend in state `Regulating`. If a `LevelNormal` event is received to indicate resuming of a normal water level, the controller will `Close` the pump/valve and continue normal operation. In the case that 5 seconds have passed and no `LevelNormal` event was received, the system is halted according to the specification with an `EmergencyStop`. Note that the system definition is incomplete: no component emits the measuring events `LevelNormal`, `LevelMin` and `LevelMax`. We will discuss on these once we introduced the operational semantics.

The operational semantics of the timed state-machines given informally above is now precisely defined using MActions. First, we define the runtime model using the three classes `Event`, `EventBroker` and `State MachineInstance` (marked grey in figure 4). Thereby, an instance of meta class `State MachineInstance` (abbreviated as `SMI` in the following) represents an *instance of* a `StateMachine`. This is expressed using the explicit *instanceOf* relation of the M3Actions framework. During runtime, this *instanceOf* relation is available and can be navigated via an implicit property `metaObject`. As runtime data, the class carries a reference to the current `activeState` and the machine's internal `clock`. `SMI`s are connected to a 'platform infrastructure' represented by class `EventBroker` that distributes the `Events`.

The action semantics come in via MOperations, depicted as operation signatures in figure 4. The main execution loop is contained in `dispatchEvents`, shown in figure 5(a). Assuming all `SMI`s are connected as `clients` to a broker instance, the main idea is to continuously broadcast each event to all client machines which in turn may emit events back that are scheduled by adding them at the end of the queue (FIFO). As seen in figure 5(a), first a decision node checks the `eventQueue` for emptiness. If the queue is empty, the termination condition checks whether all state-machines have reached a final state[11]. In case not all machines have reached a final state, a default event is created by a create action and afterwards it is enqueued in `eventQueue` (multi-valued assign operator '`+=`'). The next action queries for the first element of this list and the flow continues

[10] otherwise we would have to specify additionally the operational semantics of an expression language which does not contribute to exemplifying LT-OCL.

[11] if we assume that there is only a single event-broker, the check `self.clients->forAll(activeState.final)` would have been sufficient.



(a) EventBroker#dispatchEvents (b) StateMachineInstance#execute

Fig. 5. Main behaviour of the state machine language

with an iterate action over all `clients`[12]. The iteration over all connected clients constitutes the main part of the behaviour and consists of two actions:

fire next transition is an invocation of `execute(event : Event)`, where the iteration variable `sm` is used as new `self` and the current `event` is passed as parameter. The `execute` operation performs a single step of the machine `sm` and is described further below (cp. figure 5(b))

sm.clock := sm.clock + 1 increments the `clock` of the current `sm` object using an assignment action

Once the iteration has finished, the processed event is removed from the queue using the multi-value `removeAt` operator. Note that this global event dispatch behaviour models the simplest form of a loss-free event channel.

The second behaviour is `execute` of class `SMI` as shown in figure 5(b). Briefly described, the behaviour consists of three phases:

(1) Evaluate the guards of the active state to find any transition that can fire for the current event. This is done by invoking the `evalGuard` operation on `self`[13]. The invoked behaviour is shown in figure 6. Otherwise, if no transition is enabled, the behaviour quits.

[12] Note that `main` indicates the control flow in case of multiple object flows.
[13] `'self:self'` action has `evalGuard` as invocation target.

(2) Once a transition is found (i.e. `firedTransition` holds a reference to that transition), it is checked for a clock reset statement. If one exists, the local `clock` is reset.
(3) In case an action clause is present at the transition, a new event is created and `value` is set to the annotated string of the fired transition. Finally, this new event is enqueued at the event broker.

The behaviour of step (1) has been reduced to the minimum that is necessary to react on events and time conditions. As can be seen in figure 6, only two decision nodes check for enabled transitions. Firstly, the `clock` value is evaluated whether it exceeds the limit given in `timeLimit`. Normally, a more sophisticated expression language would be necessary to evaluate other conditions than this fixed 'greater than' clock comparison. Secondly, a subsequent decision compares the value string of the event to check for a matching guard.

Fig. 6. Evaluation of guard conditions: StateMachineInstance#evalGuards

As consequence of step (3), not every execution leads to the creation of events, but each registered state-machine *may* contribute a new event. In other words, the event queue may grow maximal by the number of registered clients in each iteration. Thus, dispatching only one event per cycle has the effect that the delivery of events might take longer the more traffic is caused by communication.

The three operations introduced are all that is needed to execute the steam-boiler models shown in figure 3. However, the initial system setup (i.e. instantiation and connection of the state-machines to the broker) are not explained here.

The system as introduced so far did not specify where the measuring events come from. In our experiments, we have modelled the physical measuring unit as 'virtual' state-machine that is connected to the event broker and that simply asks for user input in every round of event distribution[14]. These manually created events were then enqueued to trigger state-transitions in the system shown in figure 3. From our experience with the modelling of open systems such exchange of information with the environment is quite common and usually modelled separately.

[14] input was considered to be the `value` of an event, where empty strings indicate no event.

4.1 Analysis of the Steam-Boiler

In this section, we perform some analysis of the steam-boiler by means of LT-OCL constraints to exemplify the usage of temporal constraints for the domain specific state-machine language introduced in the previous section.

As *safety property*, the steam-boiler specification states that in every cycle of the steam boiler execution, the water inside the boiler should be always in safe limits within five seconds. We can express this using an always/eventually combination over the state-machine instances (LT-OCL 1):

```
let cp : SMI = SMI.allInstances()
 ->select( metaObject.name='ControllerProgram' ) in
let pump : SMI = SMI.allInstances()
 ->select( metaObject.name='Pump' ) in
always(cp.activeState='Regulating' implies cp.clock <= 5 and
 eventually((cp.activeState='Normal' and pump.activeState='On') or
   (cp.activeState='EmergencyStop'and pump.activeState='Stop')))
```

Since we don't want to express invariants over all instances of class `StateMachine` or `SMI`, we have to select the corresponding instances by name (here: using `let` statements). Note that from the semantic definition of LT-OCL, the whole expression is *undefined* until both let expression binds values for *cp* and *pump* (cf. definition 2). Additionally, we can assure correct response behaviour for e.g. the critical `LevelMax` event. To guarantee that whenever the quantity is less than the minimum limit the pump will eventually be started, might be expressed as follows (LT-OCL 2):

```
always(Event.allInstances()->forAll( value='LevelMin' implies
    eventually(pump.activeState='Pumping')))
```

An interesting property is if we want to explicitly state the time *when* the pump will be opened relative to the occur access to the `clock` attributes of both units, we could try to express this by comparison of the two clocks of the controller and the pump. However, the pump unit does not reset its clock which makes it impossible to access the points in time when the transition happens. If we would add a clock reset statement to the transition from state `On` to `Pumping`, i.e. `OpenPump/clock := 0` instead of `OpenPump`, we could try to formulate the time difference as follows (LT-OCL 3):

```
always(cp.activeState='Regulating'and pump.activeState='Pumping'
       implies pump.clock <= cp.clock + 1)
```

Note that this constraint might not be true in case the event queue is filled with other events coming from additional state-machines. As pointed out earlier, the delivery time depends on the amount of events that are generated. Even worse, the modelling of the local clocks of each state-machine are incremented in the action semantics after propagation of an event to a machine. That means, even though there would be no clock resets present at any machine, still not all clocks would be always globally *synchronized* and the constraint (LT-OCL 4):

```
always(SMI.allInstances()->forAll( sm1 |
  SMI.allInstances()->forAll( sm2 | sm1.clock = sm2.clock )))
```

would never be true. This is due to the fact that the induced states are still bound to the 'micro-states' of the action semantics and do not reflect yet the *observable states* of the language. This problem can be solved by marking the two transitions named `SGT` of the behaviour shown in figure 5(a) as State Generating Transitions (cf. section 3.2). Thus, we obtain a consistent system state across all state machines whenever the loop's SGT fires. In the example, this leads to a consistent clock increment for all state machines and therefore constraint (LT-OCL 3) can be rewritten with the *next* operator (LT-OCL 5):

```
always(cp.activeState='Regulating'
  implies next(pump.activeState='Pumping'))
```

In other words, we can either introduce the clock reset for (LT-OCL 3) or directly express the constraint over system states.

5 Related Work

There have been a couple of proposals to apply temporal extensions to OCL in the past. Driven by the need to specify temporal constraints on the dynamics of UML models, the focus of previous work concentrated on extension of OCL for UML models, for example [14] and [15]. These proposals did not provide a formal semantics.

Most recently, the work of [16] defines Temporal OCL (TOCL) with the linear temporal operators *always*, *sometime*, *always-until*, *sometime-before* and *next*. The authors give a formal definition of the temporal semantics by an extended version of the object model developed by Richters et al. (cp. OCL standard, cf. Appendix A in [1]) and which is reused in our approach. The approach presented in this paper can be seen as taking these ideas further by binding the notion of system states to traces and operational semantics. Additionally, we consider an *open* system view through interpretation along execution traces with thoroughly elaborated notion of *undefined* temporal OCL formulas. However, LT-OCL defines only future expressions (no *'previous'*, *'alwaysPast'*).

Another main direction of work targets on model checking of temporal OCL constraints [3]. Thereby, OCL definitions are mapped onto Object-Based Temporal Logic (BOTL) as semantic domain [17]. BOTL is a formal language synthesized from static expressions (similar to algebraic signatures) and Computation Tree Logic (CTL). The semantics of BOTL in turn are given by Kripke structures (cf. section 2). The authors make no explicit statement on application for other languages than UML, but the work concentrates on model checking of formalized UML state machine models. Related work of OCL extensions of branching time logic is further reported in [4] [5] for real-time systems.

5.1 Discussion

The introduction of temporal operators into a predicate logic like OCL is a non-trivial task. Apart from theoretical considerations about decidability[15], existence of efficient evaluation algorithms is a precondition to handle the complexity of expressions in practice. Hence, one design rational for LT-OCL was to deliberately limit expressions to future-directed (monadic) statements. Moreover, the choice in defining the semantics over *object changes* was motivated in the development of an efficient implementation that allows on-the-fly monitoring of constraints by means of runtime verification [18]. We are currently studying how an evaluation algorithm can be implemented that effectively uses the change information to re-evaluate only parts of an LT-OCL expression in order to deal with the overall complexity.

In the steam-boiler example, we modelled parallelism of state-machines explicitly by a loop that distributes events sequentially. MActions, ASM and others support parallelism as native language construct. As direct consequence for LT-OCL, expressiveness is limited to either *local evaluation* for a trace of each thread or *global evaluation*, if a global ordering of change events is possible. Otherwise, one need to define something equivalent to CTL (or CTL*) to express properties in branching time with path quantifiers.

6 Conclusion

This paper defined Linear Temporal OCL as extension of Essential OCL with *next*, *until*, *always* and *eventually* operators. LT-OCL formulas have temporal semantics and are interpreted over behaviour of a language given by an M3Actions specification. We explained purpose and application along with the steam-boiler control example that was specified in a timed state-machine language. We hope that this work contributes to the dynamic analysis of executable models for domain specific languages that are not defined by 'stereotyping UML', but that have precise semantics given by an action semantics. However, we see the results presented here as one step forward to an dynamic assessment framework based on OCL. For further experiments and evaluation of the approach, LT-OCL is currently implemented as proof of concept.

References

1. OMG: OCL 2.0 Specification. Object Management Group (2006) formal/2006-05-01
2. Ziemann, P., Gogolla, M.: An extension of OCL with temporal logic. In: Critical Systems Development with UML, pp. 53–62 (2002)
3. Distefano, D., Katoen, J.-P., Rensink, A.: On a temporal logic for object-based systems. In: Formal Methods for Open Objectbased Distributed Systems, pp. 305–326. Kluwer Academic Publishers, Dordrecht (2000)

[15] see [10] for a discussion.

4. Flake, S., Mueller, W.: An OCL extension for real-time constraints. In: Advances in Object Modelling with the OCL. LNCS, pp. 150–171. Springer, Heidelberg (2001)
5. Cengarle, M.V., Knapp, A.: Towards OCL/RT. In: Eriksson, L.-H., Lindsay, P.A. (eds.) FME 2002. LNCS, vol. 2391, pp. 390–409. Springer, Heidelberg (2002)
6. Soden, M., Eichler, H.: Enterprise Modelling and Information Systems Architectures - Concepts and Applications. In: Proceedings of the 2nd International Workshop on Enterprise Modelling and Information Systems Architectures. LNI, vol. P-119. GI (2007)
7. 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)
8. Abrial, J.R.: Steam-boiler control specification problem. In: Formal Methods for Industrial Applications, Specifying and Programming the Steam Boiler Control (the book grow out of a Dagstuhl Seminar, June 1995), London, UK, pp. 500–509. Springer, Heidelberg (1996)
9. Goldblatt, R.: Mathematical modal logic: a view of its evolution. J. of Applied Logic 1(5-6), 309–392 (2003)
10. Abiteboul, S., Herr, L., van den Bussche, J.: Temporal versus first-order logic in query temporal databases. In: ACM Symposium on Principles of Database Systems, Montreal, Canada, pp. 49–57 (1996)
11. Humbold University Berlin: M3Actions - Operational Semantics for MOF Metamodels (2008), http://www.metamodels.de
12. Soden, M., Eichler, H.: Eclipse Proposal: Model Execution Framework (2009), http://www.eclipse.org/proposals/mxf/
13. Soden, M.: Operational semantics for MOF metamodels: Tutorial on M3Actions (2008)
14. Conrad, S., Turowski, K.: Temporal OCL meeting specification demands for business components. In: Unified Modeling Language: Systems Analysis, Design and Development Issues, pp. 151–165 (2001)
15. Ramakrishnan, S., Mcgregor, J.: Extending OCL to support temporal operators. In: Proceedings of the 21st International Conference on Software Engineering (ICSE 1999) Workshop on Testing Distributed Component-Based Systems, LA, May 16 - 22 (1999)
16. Ziemann, P., Gogolla, M.: An OCL extension for formulating temporal constraints. Technical report, Universität Bremen (2003)
17. Distefano, D., Katoen, J.-P., Rensink, A.: Towards model checking OCL. In: Proceedings, ECOOP Workshop on a Precise Semantics for UML (2000)
18. Havelund, K., Technology, K., Rosu, G.: Monitoring programs using rewriting. In: Proceedings, International Conference on Automated Software Engineering (ASE 2001), pp. 135–143. IEEE, Los Alamitos (2001)

An MDA-Based Approach for Behaviour Modelling of Context-Aware Mobile Applications*

Laura M. Daniele, Luís Ferreira Pires, and Marten van Sinderen

Centre for Telematics and Information Technology,
University of Twente, Enschede, The Netherlands
{l.m.daniele,l.ferreirapires,m.j.vansinderen}@ewi.utwente.nl

Abstract. Most reported MDA approaches give much attention to structural aspects in PSMs and in generated code, and less attention to the PIM level and the behaviour of the modelled applications. Consequently, application behaviour is generally not (well) defined at the PIM level. This paper presents an MDA-based approach that incorporates behaviour modelling at the PIM level in the development of a specific category of applications, i.e., context-aware mobile applications. The paper also illustrates a behaviour model transformation realized by using *Medini QVT*, which is a tool that implements the Query/ View/ Transformation (QVT) Relations specification defined by OMG for model-to-model transformations.

1 Introduction

The MDA community agrees on the need to consider behavioural aspects in model transformations meant to support application development. However, there is still no agreement on how behaviour aspects should be supported. Although considerable effort has been done on model transformations from PIMs to PSMs in several application domains, most reported MDA approaches [7,8,13] give much attention to structural aspects in PSMs and in generated code, and less attention to the PIM level and the behaviour of the modelled applications. Consequently, application behaviour is generally not (well) defined at the PIM level, and behavioural aspects have to be incorporated later in the development process, by adding hand-written code as annotations to PSMs or to implementation code skeletons. In this paper, we provide an MDA-based approach that focuses on behaviour modelling of a specific category of applications, namely, context-aware mobile applications. Context-aware mobile applications are intelligent applications capable to sense changes in the user's environment and consequently adjust their behaviour in order to provide relevant functionality to their user anywhere and at anytime.

Our MDA-based approach for the development of context-aware mobile applications decomposes the PIM level in three levels of platform-independence, where each consecutive PIM level consists of a refinement of the previous one, stressing the

* This work is part of the Freeband A-MUSE Project (http://a-muse.freeband.nl). Freeband is sponsored by the Dutch government under contract BSIK 03025.

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, pp. 206–220, 2009.

behavioural aspects. This paper presents our first results towards the automation of this approach, illustrated with a case study that realises a model transformation with the *Medini QVT* tool [17], which implements the Query/View/Transformation (QVT) Relations specification defined by OMG for model-to-model transformations.

The structure of the paper is the following: Section 2 presents an overview of the A-MUSE MDA-based methodology and focuses on the PIM level of this methodology, Section 3 illustrates with a case study the application of Medini QVT in the first transformation step of our approach, Section 4 discusses our second transformation step, Section 5 discusses some related work, and Section 6 presents our conclusions and identifies topics for future work.

2 MDA-Based Approach

According to MDA principles, the A-MUSE design methodology divides the design of distributed applications in different levels of models with different degrees of abstraction and platform-independence. Fig. 1 shows how the levels considered in this methodology can be applied to context-aware mobile applications [1,2].

The *service specification level* describes a context-aware mobile service[1] as a monolithic behaviour from an external perspective. At this level, we specify the functionality that our service offers to its user and we do not consider any structural detail of the service, i.e., we abstract from its internal components.

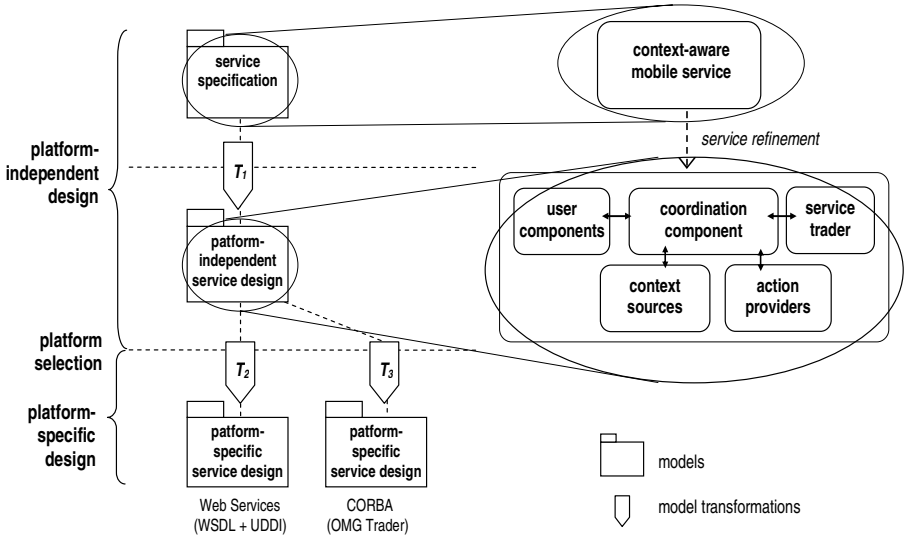


Fig. 1. A-MUSE design methodology for context-aware mobile applications

[1] The term *service* at this level denotes the observable behaviour of the whole application, as opposed to the use of the term service in service-oriented architectures to denote the functionality supported by a service provider reachable from some middleware.

The *platform-independent service design level* describes a context-aware mobile application from an internal perspective revealing a (given) architecture. This architecture, which is described in [4], consists of context sources, action providers, coordination component, service trader and user components. Context sources sense events in the user's environment and provide these events to the coordination component, which consequently triggers actions that are executed by action providers. Context sources and action providers are registered in the service trader in order to be dynamically available to the coordination component. Each user accesses the service through a user component, which provides the user interface and forwards requests to the coordination component. The core of this architecture consists of the coordination component, since it orchestrates all the internal interactions. These interactions are: (i) user requests from user components, (ii) context events from context sources, (iii) actions to be executed by action providers, and (iv) resources registered in the service trader.

The *platform-specific service design level* describes the realisation of a context-aware mobile application in terms of specific target technologies. Several alternative PSMs may implement a PIM as long as correctness and consistency are guaranteed. Therefore, it is in principle possible to use different middleware technologies to realise the platform-specific service design, such as, for example, web services or CORBA, as indicated in Fig. 1.

This paper focuses on the platform-independent design level of the methodology shown in Fig. 1, namely on the service specification and platform-independent service design model and the transformation T_1 between these models. Fig.1 shows that the platform-independent design phase in the A-MUSE design methodology is decomposed in the service specification and platform-independent service design steps. The platform-independent service design model should be a refinement of the service specification, which implies that correctness and consistency particularly of behavioural issues have to be addressed in the refinement transformation. However, when trying to realise this refinement transformation, we noticed that the gap between service specification and platform-independent service design is rather wide, so that correctness and consistency was hard to guarantee in a single refinement transformation T_1. Therefore, we introduced an intermediate step in which the service specification behaviour is refined. This intermediate step results in an intermediate design called *service design refined model* and our final PIM is renamed to *service design component model*. The refinement transformation T_1 has been consistently decomposed in two transformations T_1' and T_1". Fig. 2 shows the approach we have defined with all the PIM levels and transformations between these levels.

Fig. 2 depicts the starting point of our approach, i.e., the service specification, in which we define the functionality offered by the application to the user in terms of actions and causality relations between these actions. The actions and causality relations at the service specification level are too abstract to be directly realised by platform-specific technologies. Therefore, the second step of our approach consists of defining a service design refined model, in which we refine each of these actions and causality relations into more detailed actions and causality relations that can be directly supported by the realisation platform. In the next step we compare the refinements in the service design refined model in order to identify sequences of actions that recur in

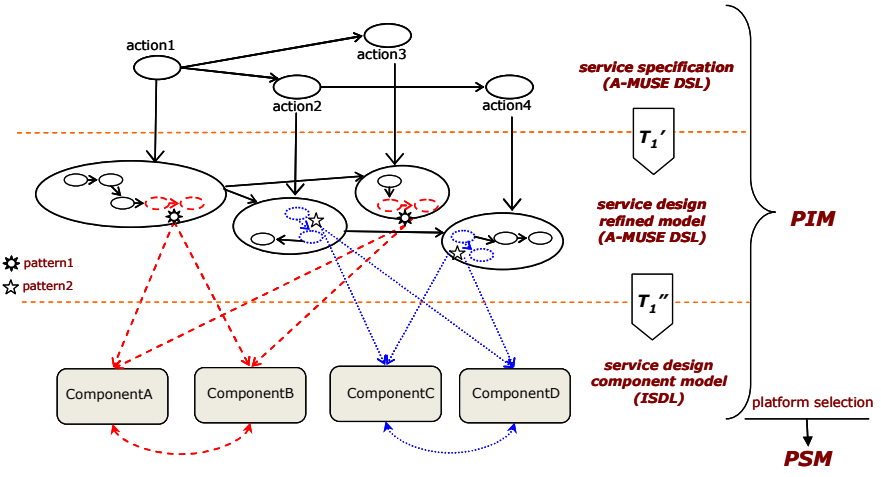


Fig. 2. PIM levels and transformations between these levels

several refinements. We call these sequences of actions *interaction patterns*, which we have defined in [3] as a 'recurring sequence of actions performed by two or more interacting components defined from the internal perspective of a service'.

Fig. 2 shows two interaction patterns and their assignment to components. The first interaction pattern, which we have marked with the "✲" symbol, recurs in the refinement of actions 1 and 3, and is assigned with dashed arrows to the corresponding components (components A and B in Fig. 2). The second interaction pattern, which we have marked with the "☆" symbol, recurs in the refinement of actions 2 and 4, and is assigned with dotted arrows to components C and D. The assignment of interaction patterns to components results in the service design component model, in which we define the behaviour of each individual component. The service design component model provides the input for the platform-specific service design level without imposing any specific implementation choices. For example, the coordinator component can be implemented as a BPEL process, the action providers as web services, and the service trader as a UDDI registry.

In contrast to our approach, most reported MDA development practices [7,8,13] do not consider behaviour refinements at the PIM level and directly start the development process by defining a platform-independent model of the architecture that implements the application (our service design component model). We believe that this is the major novelty of our approach.

3 Transformation from Specification to Design Refined Model

This section illustrates with a case study the transformation T_1' of Fig. 2 (from service specification to service design refined model). This case study is the Live Contacts application [14], which consists of a context-aware mobile application that runs on Pocket PC phones, Smartphones and desktop PCs. Live Contacts allows its users to

contact the right person, at the right time, at the right place, via the right communication channel. More information about Live Contacts can be found in [15].

We implemented transformation T_1' with the Medini QVT tool [17], which consists of a core engine that implements the QVT Relations standard defined by OMG [18], and a graphical debugger and editor to facilitate transformation development. Fig. 3 shows an overview of the Medini QVT transformation approach. Inputs to Medini QVT transformations are: (i) a source and a target metamodel defined in *Ecore*, which is the metamodel type used by the Eclipse Modeling Framework (EMF) [6], and (ii) a source model conforming to the source metamodel. The Medini QVT transformation produces as output a target model that conforms to the given target metamodel.

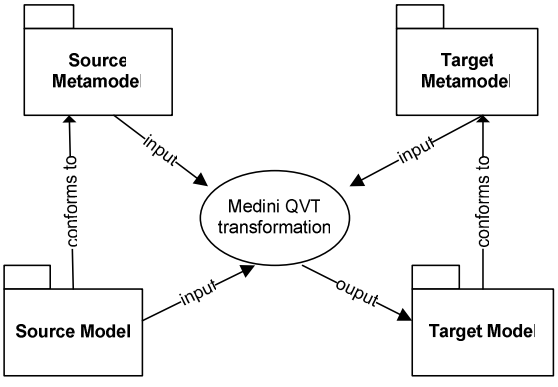


Fig. 3. Overview of the Medini QVT transformation approach

3.1 Source Model

The service specification consists of two parts: a UML information model that represents status information handled by the modelled context-aware mobile application, and an A-MUSE DSL model that specifies the behavioural aspects of the application. The A-MUSE Domain Specific Language (DSL) [11] is a language developed and applied in the A-MUSE project [9] that allows us to model behavioural aspects in terms of causality relations between interactions without constraining the internal implementation of the application. Our approach is not restricted to any specific modelling language and in this work the A-MUSE DSL is only a vehicle to reach our purposes.

Since we are developing context-aware mobile applications, our service specification actually includes also a UML context model [4], which represents the relevant concepts used by context-aware mobile service components that manipulate context, i.e., context sources and coordinator component of Fig. 1. Transformation T_1' uses this model as the reference for context information.

In our initial PIM we model the interactions between one user instance and the system. This simplification implies that when we realize the service at the platform-specific design level, we have to make sure that the resulting system is able to support different user instances that run simultaneously and consistently. Fig. 4 depicts this service specification perspective.

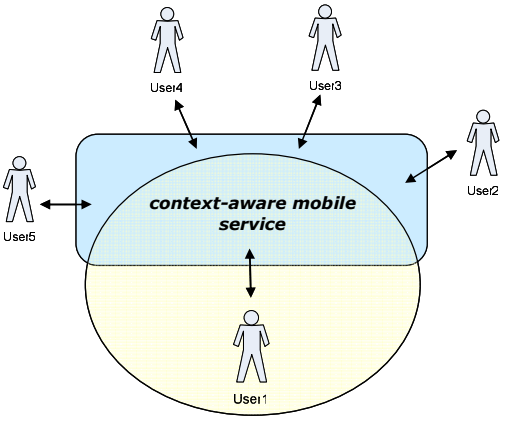


Fig. 4. Service specification perspective

Fig. 5 presents a service specification example for the Live Contacts application. We refer to the application user as an item *me* of type *FocusUser*. An item is a global behaviour variable that can be accessed in the behaviour that defines the variable. A user may request to remove a buddy from the buddy list (*removeReq*) by giving as input to the application the name of this buddy. If the buddy is not in the list (*!IsInList(removeReq.name, BuddyList): Boolean* condition), the user request is rejected (*removeRej*), otherwise (*IsInList(removeReq.name, BuddyList): Boolean* condition) the request is accepted (*removeAcc*) and the buddy is removed from the buddy list of the user (*me.getBuddyList().removeBuddy(me.getBuddyList().getBuddy (removeReq.name))* function).

The user may also contact a buddy using some specific communication means (*contactReq*). In this case, the user must give as input to the application the buddy name and the preferred means to reach this buddy. Depending on the contact means selected by the user, the application opens the appropriate communication channel (*SMS* or *e-mail*) and retrieves the mobile number or the e-mail address of the buddy (*getMobilePhoneNr()* or *getEmailAddress()* functions). We do not provide a complete specification of the SMS and Email services, since this is out of the scope of this paper. We only assume that these services need an input (*mobileNr* and *emailAddress*, respectively).

The user may also be notified by the application with an alert (*proximityAlert*) when a buddy, who is online in the application, is in the neighbourhood of the user (*proximityEvent*). The status information depicted in the behaviour of Fig. 5 is defined in the UML information model of the service specification.

3.2 Target Model

The service design refined model consists of a UML information model, and an A-MUSE DSL model that describes the structured behaviour of the modelled application, revealing the interactions among architecture components [4]. Fig. 6 shows these components in the architecture that has been applied in the A-MUSE project to realise

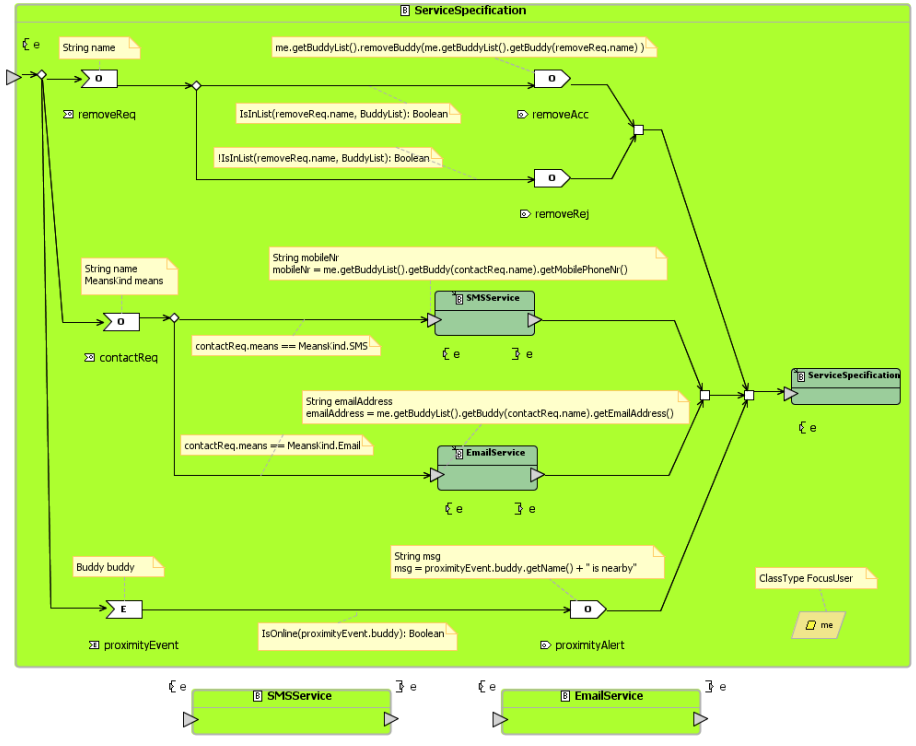


Fig. 5. Service specification: example (exported from the A-MUSE DSL editor)

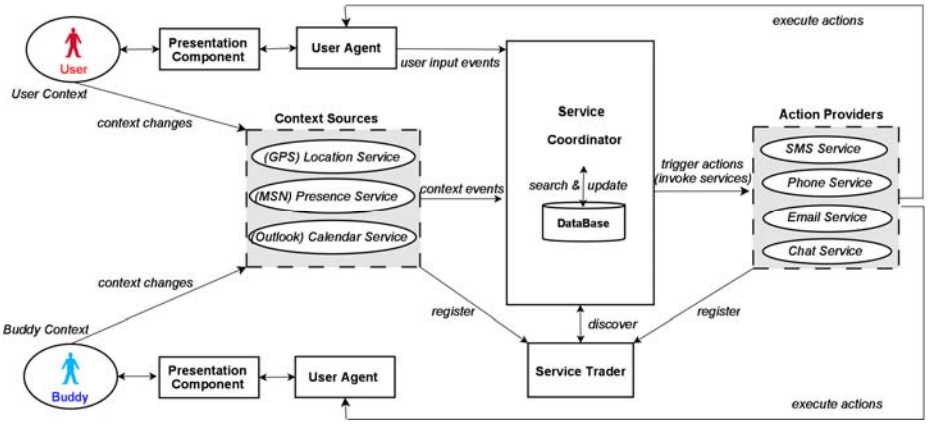


Fig. 6. A-MUSE architecture for context-aware mobile applications

the Live Contacts application. However, this architecture is general enough and can be reused for other context-aware mobile applications by simply redefining some application-specific components, such as context sources and action providers.

Fig. 6 shows the perspective of a single user with a single buddy. The presentation component takes care of the interactions with the end-user. The user agent (UA, one per user) acts on behalf of the user with the presentation component to obtain user input and present user output, and provides the service coordinator with user input events. The service coordinator (C) takes care of orchestrating the other components, searching and updating a database (DB), which contains information about users (e.g., name, password, preferred contact means and list of buddies). We assume that there is one service coordinator and one database. The service coordinator also interacts with context sources and action providers.

The context sources (CS) sense changes in the user context and provides the service coordinator with context events. Fig. 6 shows the (GPS) location service that provides information about a user's current location, the (MSN) presence service that provides indications whether users registered in the Live Contacts application are available online in the network, and the (Outlook) calendar service that provides calendar information. In this example we assume that there is one (GPS) location service, one (MSN) presence service and one (Outlook) calendar service for each user agent. These services are registered in the service trader.

The action providers (AP) are responsible for performing actions triggered by the service coordinator. Actions represent application reactions to user input events and context events. Fig. 6 shows an SMS service, phone service, e-mail service and chat service, which enable a user to communicate with buddies through, respectively, sending messages, making a phone call, sending e-mails or chatting. We assume that there is one of these services for each user agent. These services are registered in the service trader (ST), which registers all the available context sources and action providers in order to allow the coordinator to discover and invoke them.

Fig. 7 presents the service design refined model that reveals the architecture mentioned above and shows how we have refined the functionality described in the service specification, i.e., *remove Req*, *contactReq* and *proximityEvent*, in terms of sequences of actions.

Each action in Fig. 7 is marked with a label and represents an interaction between two components and the direction of this interaction. In order to avoid clogging the figure, we have not included the status information handled by components. This information is the same as depicted in Fig. 5, but assigned to the proper corresponding refined actions.

Fig. 7 shows that the user request to remove a buddy from his/her list arrives to the user agent (UA), which forwards this request to the coordinator (C). The coordinator checks the database (DB) to determine whether the buddy is included in the buddy list of the user (*findRemReq* and *findRemRsp*). If this is the case, the coordinator removes the buddy from the list (*removeBuddy*) and sends a positive response to the user agent (*removeAcc*), which presents the result to the user. If the buddy is not in the list, the coordinator sends a negative response to the user agent (*removeRej*), which presents the result to the user.

The user request to contact a buddy with a specific means arrives to the user agent (*contactReq*). The user agent forwards this request to the coordinator, which retrieves the buddy to be contacted from the database (*findContactReq* and *findContactRsp*).

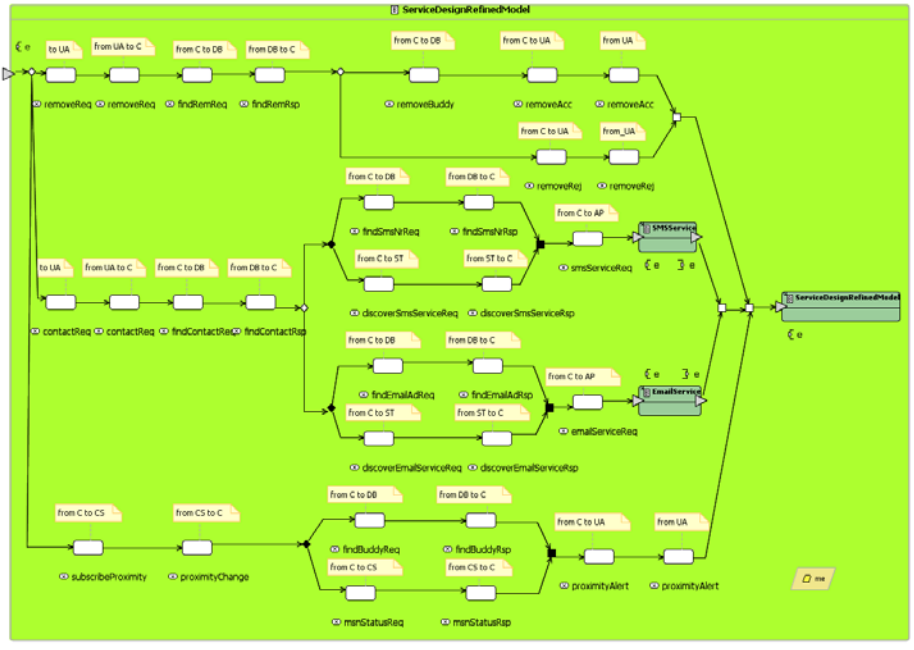


Fig. 7. Design refined model: example (exported from the A-MUSE DSL editor)

The coordinator evaluates the parameters of the contact request. Depending on the means selected by the user, the right communication channel is selected (SMS or e-mail). In both cases, the coordinator performs two activities concurrently, namely, retrieving from the database the number or address where to contact the buddy, and asking the service trader (ST) to discover the proper service to contact the buddy. Once both activities are concluded, the coordinator is able to invoke the proper action provider (AP) and provide it with the necessary input, which may be the mobile number or the email address of the buddy.

In order to notify the user of the occurrence of a proximity event, the coordinator has to subscribe in the context sources (CS) a context expression related to that event (*subscribeProximity*). When the value of this expression becomes true, the context source notifies the coordinator (*proximityChange*). While retrieving the buddy name from the database (*findBuddyReq* and *findBuddyRsp*), the coordinator requests to the context sources the current MSN status of the buddy (*msnStatusReq* and *msnStatusRsp*) in order to check if this value is *online*. If this is the case, the coordinator generates an alert to the user agent, which forwards the alert to the user (*proximityAlert*).

3.3 Transformation Relations

Most reported MDA model transformations are realised between models that conform to structurally similar but different metamodels, such as, for example, from UML class diagrams to Java code skeletons. In contrast, transformation T_1' is an architectural

transformation, which relates element structures of the source model to more complex element structures in the target model. In order to generate the target model of Fig. 7 from the source model of Fig. 5, we have created a transformation called *SStoSDRM* between the domains *SS* (Service Specification) and *SDRM* (Service Design Refined Model). This transformation consists of five basic relations in the QVT Relations language used by the Medini QVT tool. Fig. 8 shows these relations schematically.

The first relation (*BehaviourMapping* in Fig. 8) creates a one to one mapping of an SS element *Behaviour* with name *ServiceSpecification* onto an SDRM element *Behaviour* with name *ServiceDesignRefinedModel*. The second relation (*EntryItemInstanceMapping* in Fig. 8) creates a one to one mapping from SS to SDRM of: (i) an *EnablingRelation* element that relates an *EntryPoint* to an *OrSplit* (▷⟶◇), (ii) an *EnablingRelation* element that relates an *OrJoin* to the *EntryPoint* of a behaviour instance (□⟶▷), and (iii) an *item* element with name *me*. The third relation (*RemoveRequestMapping* in Fig. 8) maps the *removeReq* functionality of the SS onto the

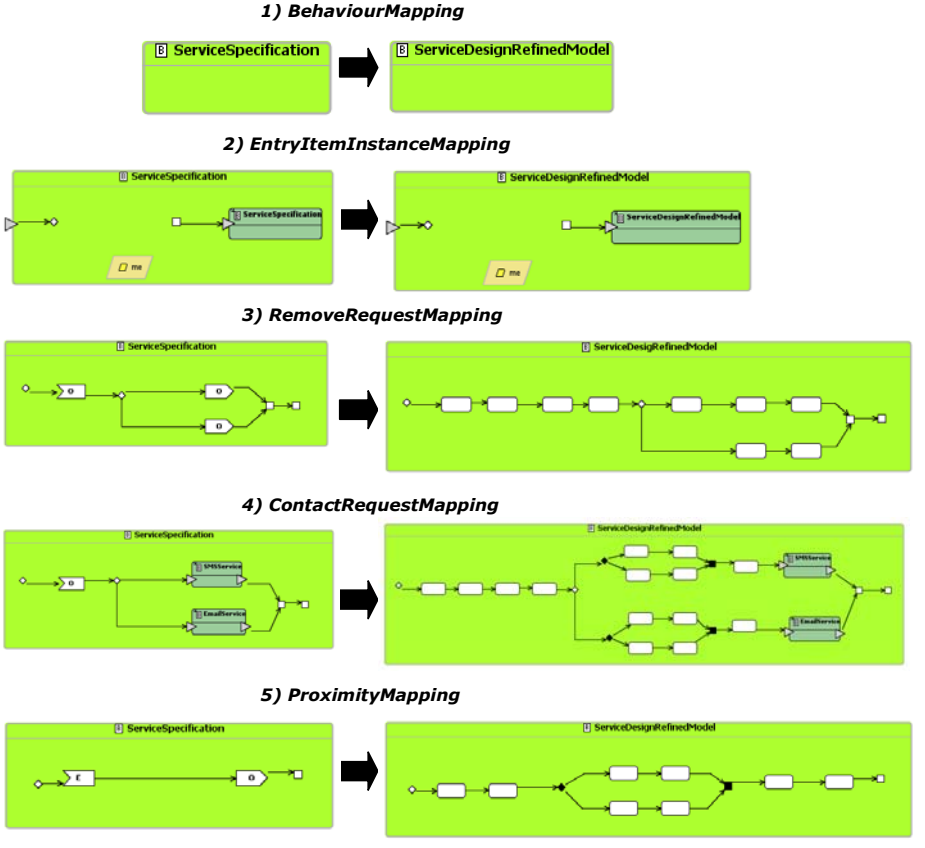


Fig. 8. Transformation relations

refinement of the same functionality in the SDRM. The fourth relation (*ContactRequestMapping* in Fig. 8) maps the *contactReq* functionality of the SS onto the refinement of the same functionality in the SDRM. The last relation (*ProximityMapping* in Fig. 8) maps the *proximity* functionality of the SS onto the refinement of the same functionality in the SDRM. In this way the architectural refinement alternatives are documented and can be applied appropriately to the source model.

4 Transformation from Design Refined Model to Component Model

This section discusses the source and target models of transformation T_1" of Fig. 2 (from service design refined model to service design component model) with the Live Contacts application.

4.1 Source Model

The source model of this transformation consists of the target model of transformation T_1'. In this model, we have been able to identify recurring sequences of interactions between components (interaction patterns). Particularly, we have identified two types of interaction patterns, namely *basic* and *composite* patterns. Basic interaction patterns occur between two interacting components. An example of basic patterns in Fig. 7 is *findRemReq* and *findRemRsp*, which we call *search* pattern, and involves interactions between the coordinator (C) and the database (DB). Composite interaction patterns occur between more than two components and consist of combinations of basic patterns. Examples of composite patterns in Fig. 7 are the refinement of the *removeReq* functionality, which we call *user request with acceptance or rejection* pattern, the refinement of the *contactBuddy* functionality, which we call *user request with external service pattern*, and the refinement of the *proximityEvent* functionality, which we call *context event with alert* pattern. From our experience with the Live Contacts application we have defined a library of basic and composite patterns [5] that cover the functionality of context-aware mobile applications that comply with the architecture of Fig. 6.

4.2 Target Model

Fig. 9 shows an example of a service design component model with limited functionality. This should be the target model of the transformation T_1". In this model, we have assigned the basic and composite interaction patterns identified in the service design refined model to concrete components that can realise these patterns.

Fig. 9 depicts the assignment of the *user request with acceptance or rejection* composite pattern to components. Dashed lines indicate the assignment of basic patterns to components. The three involved components are the user agent, the coordinator and the database. Fig. 9 depicts the behaviour of these components considering their interactions. Fig. 9 uses ISDL (Interaction System Design Language) [12], which allows the specification of behavioural aspects of interacting components.

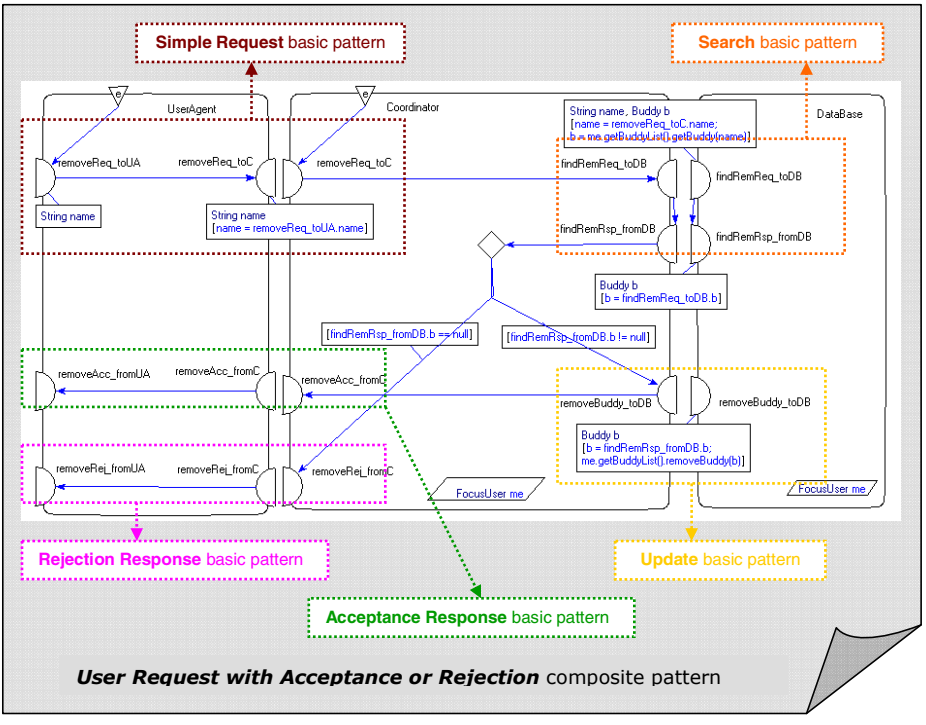


Fig. 9. Service design component model: example (exported from Grizzle [10])

5 Related Work

We briefly discuss here some related work that deals with MDA approaches in several application domains. In [16] the importance of behaviour modelling techniques for MDA is stressed. Given that MDA requires application behaviour to be explicitly represented at the PIM level and that there is no consensus on how to represent this behaviour, different categories of behaviour modelling methods are analysed. The state-machines paradigm is identified as the best basis for such representation. We agree that state machines are suitable to represent behavioural aspects of applications, but we think that they are more suitable for the specification of behaviours that are directly assigned to concrete components, and less suitable for the specification of highly abstract behaviours, as in the case of our service specification (initial PIM).

Considerable effort has been spent on model transformations from PIMs to PSMs in several application domains. For example, in [13], a formal MDA approach for the development of mobile health system is discussed. A model-driven approach for the development of access control policies for distributed systems is presented in [8]. In [13] an MDA approach to implement personal Information Retrieval (IR) processes is proposed. Although these approaches are applied to different application domains, they all report on case studies in which little attention is given to the behaviour of the modelled application at the PIM level, and directly generate PSMs based on the

architecture that implements the application. The PIMs are defined by using UML, and PSMs are represented in programming languages, such as Java. Therefore, the PIM-to-PSM transformation is performed between models that have structurally similar but different metamodels. In contrast, we focus on application's behaviour refinements at the PIM level and we realise transformations between models that have the same metamodel, but our transformations embody architectural decisions and preserve correctness and consistency.

6 Conclusions and Future Work

According to MDA principles, the A-MUSE design methodology divides the design of distributed applications in platform-independent and platform-specific design levels. In this paper, we focused on the platform-independent design level of this methodology, and we proposed an approach to model the application behaviour at this level through model refinements. While realising these refinements, we noticed that the gap between abstract specifications and concrete models of application components is rather wide, so that it was advisable to add an intermediate level of behaviour modelling to the A-MUSE methodology. Therefore, we defined an approach that decomposes the platform-independent design level in three models and two model transformations. This paper focuses on the first of these model transformations.

Towards the automation of the first model transformation, we investigated the usage of the Medini QVT tool, which allowed us to define transformation rules in the QVT Relation language defined by OMG. Although the results we have discussed in this paper are preliminary, we can already conclude that it is possible to realise automated behaviour model refinements in the context of MDA, and Medini QVT is an appropriate tool to support this goal. However, further work needs to be performed in order to achieve full automation of this transformation. The presented example transformation considers only part of the functionality offered by the Live Contacts application to its users, and it should be extended with transformation rules that consider the whole functionality. These extended transformation rules should also be tested and validated with new case studies in order to demonstrate that they can be reused with several context-aware mobile applications. Moreover, we still have to identify the most appropriate level of granularity to define our transformation rules. When realising transformation T_1', we have identified two types of interaction patterns, namely basic and composite patterns. Basic patterns involve interactions between two components and composite patterns involve interactions between more than two components. We defined transformation rules for composite interaction patterns and we noticed that these patterns are not very flexible, since they are complex and have a fixed structure. Therefore, our transformation rules became large and complex. The library of composite patterns is small, so the benefit of using these patterns is that the number of transformation rules we needed to create was also small. In contrast, the library of basic patterns is large and it would require a large number of transformation rules. However, basic patterns give more flexibility in the design, since they are small, simple, and can be dynamically combined in different configurations of complex behaviours. Therefore, we learned that when considering the granularity of interaction patterns, the trade-off between the number, size, and complexity of transformation

rules on one hand, and the flexibility of design choices on the other hand have to be considered. The most appropriate level of granularity level for interaction patterns is subject to further investigation.

The second model transformation has been realised manually by assigning interaction patterns to components that realise these patterns. This assignment was quite straightforward, since we defined the interaction patterns as annotated actions that explicitly specify which components participate in the pattern. However, we noticed that some synchronization and concurrency issues of interacting components still had to be considered. For example, the coordinator component that orchestrates all the interactions with other components in Fig. 7 has to schedule somehow the execution of the composite patterns that refine the *removeReq*, the *contactReq*, and the *proximity* abstract actions. The designer may decide to interleave these composite patterns, by executing all the patterns one at a time in a single thread of control. Alternatively, the designer may decide to execute these patterns in parallel threads of control. Independently of the option chosen, some formalism should be used to represent and analyse these choices. Moreover, the model in Fig. 7 represents only one user instance interacting with the application. In reality, the coordinator has to handle multiple user instances running at the same time. Therefore, further investigation is necessary on the formalisms that can be used to support these aspects. Process algebra and labelled (modal) transition systems seem to be suitable formalisms for these purposes. For example, an automated technique for synthesizing behavioural models from safety properties and scenario-based specifications by using Modal Transition Systems (MTSs) that can be directly derived from labelled transition systems is discussed in [19].

Although in this paper we made some design decisions to illustrate the case study, our approach is not restricted to these decisions. For example, we used A-MUSE DSL and ISDL to define our behaviour models, since these are general-purpose languages that allow the modelling of application behavioural aspects in terms of causality relations between interactions without constraining the internal implementation of the modelled application. However, we consider A-MUSE DSL and ISDL as vehicles to define behaviour models that can be automatically transformed. Other languages and techniques, such as UML activity diagrams or Message Sequence Charts (MSC), may be used to define our models, as long as they allow us to represent behavioural aspects of the modelled applications exhaustively. Moreover, we tailored our approach to the design of a specific category of applications, i.e., context-aware mobile applications. However, the same approach for behaviour modelling of applications through model refinements, transformation rules and interaction patterns can be applied to other categories of distributed applications by simply adjusting the reference architecture of Fig. 6. This adjusted reference architecture should reflect the architectural components and interactions between components that hold in the new target application domain. Consistently, the interaction patterns should be identified in accordance with this new architecture.

References

1. Almeida, J.P.A., Iacob, M.E., Jonkers, H., Quartel, D.: Model-Driven Development of Context-Aware Services. In: Eliassen, F., Montresor, A. (eds.) DAIS 2006. LNCS, vol. 4025, pp. 213–227. Springer, Heidelberg (2006)

2. Almeida, J.P.A.: Model-Driven Design of Distributed Applications. Ph.D. thesis, University of Twente, Enschede, The Netherlands (2006)
3. Daniele, L., Ferreira Pires. L., van Sinderen, M.: Interaction Patterns for Refining Behaviour Specifications of Context-Aware Mobile Services. In: Proceedings of the 4th International Workshop on Model-Driven Enterprise Information Systems (MDEIS 2008), Barcelona, Spain, June 2008, pp. 64–76. INSTICC Press (2008)
4. Daniele, L., Ferreira Pires. L., van Sinderen, M.: Context Handling in a SOA Infrastructure for Context-Aware Applications. In: Proceedings of the 2nd International Workshop on Architectures, Concepts and Technologies for Service Oriented Computing (ACT4SOC 2008), Porto, Portugal, July 2008, pp. 27–37. INSTICC Press (2008)
5. Daniele, L., Ferreira Pires. L., van Sinderen, M.: Live Contacts Case: from Service Specification to Service Design. A-MUSE project deliverable D2.27 (2008)
6. Eclipse Modeling Framework Project (EMF), http://www.eclipse.org/modeling/emf
7. Eissen, S.M., Stein, B.: An MDA Approach to Implement Personal IR Tools. In: 18th International Conference on Database and Expert Systems Applications (DEXA 2007), pp. 259–263. IEEE Computer Society Press, Los Alamitos (2007)
8. Fink, T., Koch, M., Pauls, K.: An MDA Approach to Access Control Specifications Using MOF and UML Profiles. In: The First International Workshop on Views on Designing Complex Architectures (VODCA 2004). Electronic Notes in Theoretical Computer Science, vol. 142, pp. 161–179 (2006)
9. Freeband A-MUSE Project, http://a-muse.freeband.nl
10. Grizzle Home, http://isdl.ctit.utwente.nl/tools/grizzle
11. Heerink, L., Quartel, D.: Domain Specific Language for Context-Aware Mobile Services. A-MUSE project deliverable D1.13 (2007)
12. ISDL Home, http://isdl.ctit.utwente.nl
13. Jones, V., Rensink, A., Ruys, T., Brinksma, E., van Halteren, A.: A Formal MDA Approach for Mobile Health Systems. In: Proceedings of the Second European Workshop on Model Driven Architecture (MDA) with an emphasis on Methodologies and Transformations (EWMDA 2004), Computing Laboratory, University of Kent, Canterbury, Kent CT2 7NF, UK, Canterbury, pp. 28–35 (2004)
14. Live Contacts Home, http://livecontacts.telin.nl
15. Ter Hofte, G.H., Otte, R.A.A., Kruse, H.C.J., Snijders, M.: Context-Aware Communication with Live Contacts. In: Conference Supplement of Computer Supported Cooperative Work (CSCW 2004), Chicago, USA (November 2004)
16. McNeile, A., Simons, N.: Methods of Behaviour Modelling: A Commentary on Behaviour Modelling Techniques for MDA. Metamaxim Ltd Home, http://www.metamaxim.com/download/documents/Methods.pdf
17. Medini QVT: IKV++ Technologies Home, http://www.ikv.de
18. Object Management Group: MDA-Guide, Version 1.0.1, omg/03-06-01 (2003)
19. Uchitel, S., Brunet, G., Chechick, M.: Behaviour Model Synthesis from Properties and Scenarios. In: Proceedings of the 29th International Conference on Software Engineering (ICSE 2007), Minneapolis, USA, May 2007, pp. 34–43. IEEE Computer Society Press, Los Alamitos (2007)

A Model Driven Approach to the Analysis of Timeliness Properties

Mohamed A. Ameedeen[1], Behzad Bordbar[1], and Rachid Anane[2]

University of Birmingham, Birmingham, UK
{M.A.Ameedeen,B.Bordbar}@cs.bham.ac.uk
[2]Coventry University, Coventry, UK
R.Anane@coventry.ac.uk

Abstract. The need for a design language that is rigorous but accessible and intuitive is often at odds with the formal and mathematical nature of languages used for analysis. UML and Petri Nets are a good example of this dichotomy. UML is a widely accepted modelling language capable of modelling the structural and behavioural aspects of a system. However UML lacks the mathematical foundation that is required for rigorous analysis. Petri Nets on the other hand have a strong mathematical base that is well suited for analysis of a system but lacks the appeal and ease-of-use of UML. Design in UML languages such as Sequence Diagrams and analysis in Petri Nets require on one hand some expertise in potentially two incompatible systems and their tools, and on the other a seamless transition from one system to the other. One way of addressing this impediment is to focus the software development mainly on the design language system and to facilitate the transition to the formal analysis by means of a combination of automation and tool support. The aim of this paper is to present a transformation system, which takes UML Sequence Diagrams augmented with time constraints and generates semantically equivalent Petri Nets that preserve the timing requirements. A case study on a small network is used in order to illustrate the proposed approach and in particular the design, the transformation and the analysis processes.

1 Introduction

One of the most pressing tasks facing software developers in general and software engineers in particular is the development of software tools that support an integrated approach to the design and the analysis of software systems. The design of a system may be considered as an essentially cognitive activity with a focus on clarity, while its analysis is usually firmly grounded on mathematics and relies often on formal representations and formal processing. The tension that results from this dichotomy has presented a serious challenge for the deployment of existing software tools in both areas. This difficulty is further compounded by the lack of interoperability between the tools associated with each phase. There is a clear need for the development of tools and frameworks that can reconcile the goals of the design and the analysis processes [1-3].

On the design side the Unified Modelling Language (UML) has been a focal point of activity in the software design community. Its rich constructs have conferred to

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, pp. 221–236, 2009.

UML a privileged role in designing software systems in a variety of domains such as network, business modelling and security [4]. One shortcoming of UML is however its inability to support the model analysis process.

The requirements for the formal analysis of software systems have been met by the introduction of a wide range of formal languages which are well suited for analysis, among them Alloy [5], Z [6] and Petri Nets (PN) [7]. In providing support for design and analysis, one common approach is to create the design in UML languages and transform it into a formal representation for analysis. For example, UML2Alloy makes use of Alloy for the analysis of a model which is captured in UML class diagrams and OCL [3]. The analysis performed in the Alloy framework involves solving logical constraints on the model. Although Alloy is useful for the analysis of the static aspects of a design but it is not particularly suitable for behavioural modelling [3]. This limitation is one of the reasons that led developers to rely on formal languages such as Petri Nets. Petri Nets are suitable for analysis of behavioural aspects of models such as deadlock detection, liveness and reachability. Their usefulness has also been enhanced by the availability of tools such as PIPE [8] and CPNTools [9].

The gap between the design and analysis in this respect can be bridged by using Model Driven Development (MDD) model transformation as outlined in [10]. The proposed model transformation addresses this issue by combining the strengths of the two languages: the specification of the behaviour of a system is formulated in UML Sequence Diagrams and the analysis is performed on Petri Nets. The transition from UML to Petri Nets is achieved via a transformation process, which takes a model of Sequence Diagrams and automatically generates the equivalent Petri Net model. The model can subsequently be analysed with Petri Net tools. Figure 1 gives a high-level description of this process. The transformation has already been covered in a previous work by the development of a model driven development (MDD) model transformation tool, SD2PN [10]. As an initial stage in the development of the framework it did not include timing capabilities.

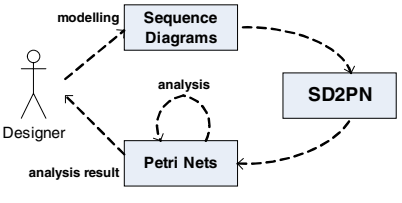


Fig. 1. Overview of the Model Transformation

The contribution of this paper is extending the model transformation in [10] with timing constraints, as part of a framework for software development. The main advantage of this extension is the application of the tool to the analysis of time sensitive systems, such as the modelling of Quality of Service (QoS) and its validation. The proposed framework is supported by a case study on a QoS specification and analysis in a Personal Area Network (PAN). UML Sequence Diagrams are used to model parts of the IEEE 802.11 protocol. The model is then transformed into Petri Nets and

analysed using relevant Petri Net tools for calculating the maximum waiting time for a station in the PAN.

The remainder of the paper is organized as follows. Section 2 provides some background information on MDD, UML and Petri Nets. Section 3 presents a summary of previous work [10] and its extension. Section 4 describes the proposed tool. Section 5 deals with a case study based on the PAN and its analysis in Petri Nets. Section 6 provides a discussion and outlines some further work. Section 7 concludes the paper.

2 Preliminaries

This section introduces some preliminary material regarding Unified Modelling Language, Petri Nets, Model Driven Development and their role in the transformation process.

2.1 Unified Modelling Language

The function of a *model* is to capture a view of the system. In software engineering models are abstractions of a physical system which has a specific purpose [11]. Unified Modelling Language (UML) is a family of languages, which is widely accepted as the *de facto* standard for software modelling. UML models can be used to specify the structure of the system, its behaviour and the constraints that the system must adhere to. This includes constraints related to the timing of the occurrence of events.

Models in UML are instances of *metamodels.* A metamodel includes system elements, their relationships and a set of rules to which every model must conform in order to be well defined. In this paper Sequence Diagrams are used as the main modelling language for describing the behaviour of a system.

Sequence Diagrams. Sequence Diagrams are used to define the interactions between objects and the flow of events within a system. They are based on Message Sequence Charts which are extensively used to capture scenarios for distributed telecommunication systems. Figure 2 is a small metamodel for Sequence Diagrams, adapted from [10], and will be used throughout this paper for explanation purposes. The metamodel of Figure 2 extends the metamodel used in our previous work [10]. However, while [10] presented a metamodel for general Sequence Diagrams, this paper presents an extension that enables the Sequence Diagrams to be augmented with timing constraints while still adhering to the UML 2.1 standards. The shaded boxes in the metamodel in Figure 2 depict these time related extensions.

Time Properties. The shaded elements in the metamodel of Figure 2 depict the previously mentioned extension that signifies the addition of time properties into Sequence Diagrams. These elements are adapted from "Common Behaviors", chapter 13 of the UML 2.1 Superstructure [11]. IntervalConstraint, TimeConstraint and DurationConstraints are all specifications of the class Constraint and they are used to define particular types of constraints. TimeConstraint and DurationConstraint refer to TimeInterval and DurationInterval respectively. An Interval is used to specify the range between two ValueSpecifications through a maximum and minimum value.

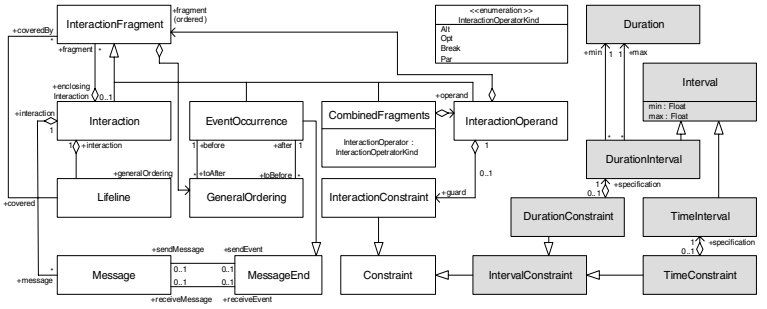


Fig. 2. Sequence Diagram Metamodel

These values are inferred as float instead of ValueSpecification as to keep the metamodel to a minimum. While Intervals have a maximum and minimum value, Duration "defines a value specification that specifies the temporal distance between two time instants" as described in page 437 of [11]. This is also evident in [13] where Douglass interprets that an Interval is not a length of time, but rather a start and end point of a time frame while Duration is a relative time measure that has a scalar value independent of the start time. Douglass further noted that time constraints are syntactically represented textually inside curly brackets, which is also evident in page 59 of [11]

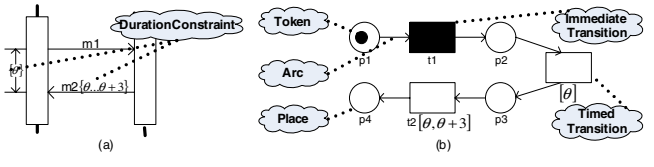


Fig. 3. Examples of a (a) Sequence Diagram and (b) its equivalent Petri Net

Störrle [14] interprets the concept of time in UML 2.0 Sequence Diagrams as divided to two types; the first of which is preserving the state of the system for a certain duration or a time interval while the second represents the duration for a single event to occur. Both these types can be represented by intervals between pairs of event occurrences. This is consistent with UML 2.1 since page 482 of [11] also describes Duration to be "always between occurrences".

Figure 3 (a) shows an example of a Sequence Diagram that features both types of time constraints. The interval between *m1* and *m2* denoted by θ shows that the state after *m1* is completed is preserved for the duration of θ while the occurrence of message *m2* takes between θ and $\theta+3$ to occur, where θ is a constant.

2.2 Petri Nets

Petri Nets are a mathematical and graphical modelling language, which can be applied to complex systems. Petri Nets can be used to model a diverse set of behaviour

including parallel, asynchronous, concurrent, hierachical and stochastic as well as dynamic behaviours [7]. Similarly to Sequence Diagrams, a Petri Net can model the flow of events in a system graphically. The formal and mathematical nature of Petri Nets can be used to overcome the limitations of Sequence Diagrams with respect to analysis.

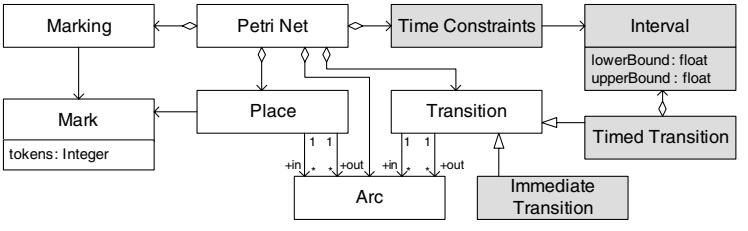


Fig. 4. Petri Net Metamodel

Figure 4 shows the metamodel of a Petri Net, which will be used throughout the paper. This metamodel is adapted from [10] and enhanced with timing properties (shaded elements). The model elements will be explained in terms of an example specified in Figure 3 (b).

The example depicts a Petri Net that models the behaviour captured in the Sequence Diagram of Figure 3 (a). This Petri Net consists of 4 *places* and 3 *transitions* that are all connected by *arcs.* An *arc* in a Petri Net serves as a connector between *places* and *transitions* and may not connect 2 *places* or 2 *transitions*.

A *transition* in the Petri Net has *input places* and *output places*, which are *places* that have *arcs* in and out of the *transition* respectively. A *transition* is *enabled* and ready to *fire* when all of its *input places* have at least a *token* each. *Tokens*, as presented in Figure 3 (b), are depicted as filled circles contained inside *places*. When a *transition fires*, a *token* will be removed from each of the *input places* and added into one of the *output places*. For further information on Petri nets see [7]. The shaded parts of Figure 4 are extension of the conventional Petri Net metamodel in [10] by including time properties, which are explained as follows.

Timed Petri Nets. Timed Petri Nets are extensions to the conventional Petri Nets by the inclusion of timing information such as the time associated to the firing of transitions. There are different flavours of Timed Petri Nets. In this paper, the Timed Petri Net with closed intervals as outlined in [15] are used. The timing information in the metamodel are inferred from the Petri Net tools where [8] shows the existence of two distinct types of transitions and [16] states that each time marking is modelled via closed intervals. These *intervals* are defined via specific upper and lower bounds attached to a *transition*. For a *transition* to *fire*, firstly it must be *enabled*. Secondly, from the moment it gets *enabled*, a clock starts; the *transition* can *fire* when the value of the clock is within the interval. An example of a timed transition is shown in Figure 3 (b) where the transition *t2* has a time constraint with the closed interval $[\theta, \theta+3]$. The transition *t2* can only fire under two conditions; it must be enabled and the clock

must be between θ and θ+3. For more information regarding Timed Petri Nets, readers are referred to [15].

The graphical representation of Timed Petri Net also differs slightly from the Petri Net used in [10]. In this paper, the *immediate transitions* or *transitions* without time constraints are depicted as black rectangles while the *timed transitions* are depicted as white rectangles; both of which are shown in Figure 3 (b) This is to provide a contrast between the two types of transition although semantically, an immediate transition is equivalent to a timed transition with an interval of [0, 0]. For *timed transitions*, the *interval* is shown in a bracket by the label of the *transitions*, with a comma separating the upper and lower bound. In a scenario such that the upper and lower bounds are the same i.e. [50, 50]; it is abbreviated as [50]. Each of the upper and lower bound must be of type *float* as inferred from the Sequence Diagrams.

The inclusion of time constraints in Petri Nets enhances their capability for modelling time-sensitive systems. Moreover, with the benefit of using existing Timed Petri Net tools such as CPNTools [9] and PIPE [8], time analysis could take place, thus making it an ideal destination model in an MDD model transformation.

Petri Net Analysis. The mathematical nature of Petri Nets creates a strong base for various types of analysis. Murata [7] outlines a number of analysis methods how they relate to the problems in designing an enterprise system. Among others, *Reachability* analysis is used to study the dynamic properties of a system i.e. how taking one action may effect the chances of an event happening in the future. A *Boundedness* analysis is used to check the effect of the system to the buffers and registers for storing intermediate data while a *Liveness* analysis checks the system for deadlocks. All these analysis and more can be performed on general Petri Nets and are supported by tools such as CPNTools [9], PIPE [8] and various other tools.

While the analysis capabilities of general Petri Nets focus on the structural and behavioural properties of a system, the addition of time properties to the Petri Nets allows for performance analysis as well. A Cycle-time analysis could be used to determine the duration for a complete sequence of action in the system while a tool such as CPNTools [9] can be used for computing the amount of time that separates two events, i.e. time between requesting access to a resource and getting the resource. Various Petri Net tools also provide a platform for other performance analysis such as average time, standard deviations, confidence intervals and throughput analysis as described in [8, 16].

2.3 Model Driven Development

One of the aims of Model Driven Development [17] is to promote the role of *modelling* in software development. Central to the MDD is the process of Model Transformation, which automatically generates a new model from an existing one. Figure 5 depicts an outline of MDD and the process of *Model Transformation.* A number of *Transformation Rules* are used to define how various elements of one metamodel (*source metamodel*) are mapped into the elements of another metamodel (*destination metamodel*). The process of Model Transformation is carried out automatically via the software tools which are commonly referred to as *Model Transformation Frameworks* [18-20].

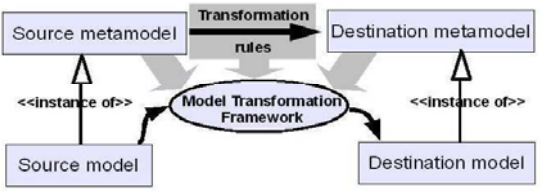


Fig. 5. Model Driven Development

A typical Model Transformation Framework requires three inputs: source metamodel, destination metamodel and transformation rules. For any instance of the source metamodel, the Model Transformation Framework executes the rules to create an instance of the destination metamodel. One way to express such rules is through Query/View/Transformation [21]. QVT is a standard for expressing MDD model transformations governed by the Object Management Group (OMG). Further reading on QVT could be found in [21].

3 Model Transformation

This section recalls the model transformation in [10] and discusses the extensions made to it to enable the analysis of timeliness properties using MDD. SD2PN [10] used a rule-based approach to map Sequence Diagrams into conventional Petri Nets. This model transformation had three stages; Decomposing Sequence Diagrams into fragments, transforming each fragment into Petri Net blocks and putting together the blocks of Petri Nets. A brief outline of the five transformation rules in SD2PN is given below.

Figure 6 shows the transformation rules used in SD2PN. Rule 1 describes the transformation of a *message* fragment into a block of Petri Net as shown in Figure 6. Rule 2 in refers to the CombinedFragment with the InteractionOperatorKind alternative while Rule 3 refers to option. Page 468 of [11] describes an option with with a sole operand to be semantically equivalent to an alternative where the second operand is empty, which will be the default for Rule 3. The guards for these CombinedFragments can be directly transformed from the sequence diagram fragment and incorporated as guards on the respective transitions. However, Timed Petri Net tools are not equipped to consider the guards as constraints and this limitation is a course for future research. Rules 4 and 5 refer to the *InteractionOperatorKind break* where the node *X* signifies the terminal node and *parallel* fragment, respectively.

The snippet of QVT given for Rule 3 is an example of how this model transformation could be carried out. In line 11, the type of the InteractionOperator is checked, and the enumeration for InteractionOperatorKind *option* is 1. The Petri Net is then built according to the diagram of Rule 3.

In this paper, the five rules of SD2PN are refined with time properties to make them compliant with the new metamodels. Referring to section 1 of Figure 6, Rule 1 is used to transform every message in a Sequence Diagram into a Petri Net block

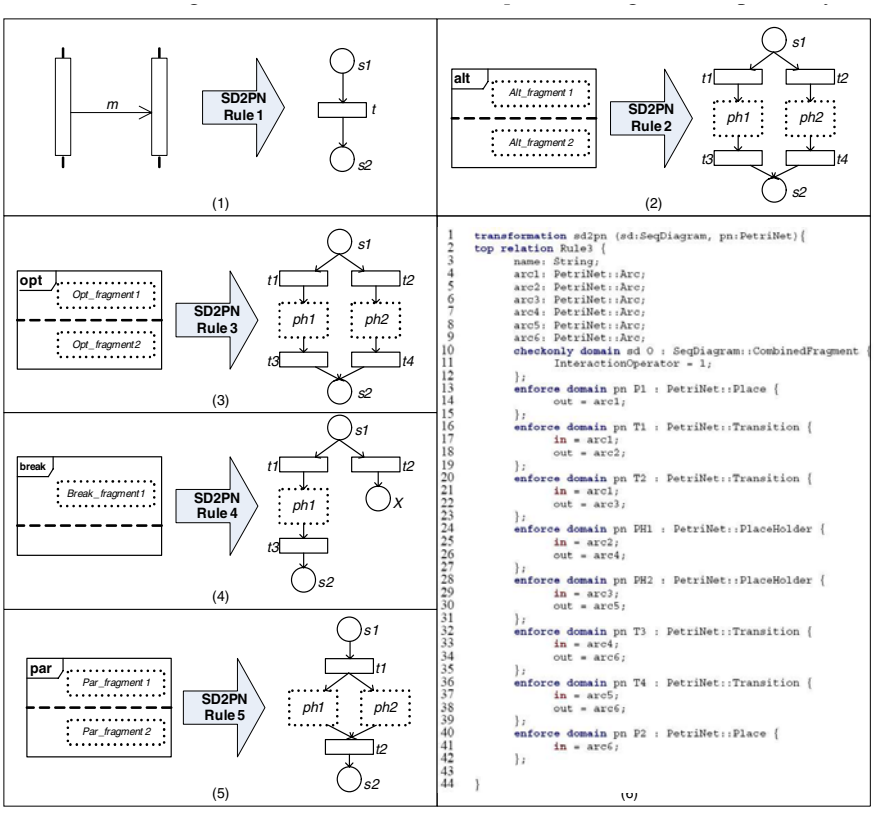


Fig. 6. Five Rules of SD2PN and example of QVT

consisting of two *places*, *s1* and *s2*, and a *transition*, *t*. By adding a time constraint to this rule, the *transition t* is given an Interval constraint with a maximum and minimum value acting as its upper and lower bound. There are three possible scenarios that could provide different outcomes to the rule. If a message has an interval with different maximum and minimum values associated to it i.e. {10...30}, the *transition t* in the resulting Petri Net block will be designated as a *Timed Transition* with a closed interval [10, 30]. Similarly, if a message has a duration associated to it i.e. {20}, the *transition t* in the resulting Petri Net block will be designated as a *Timed Transition* with a closed interval [20, 20] or abbreviated as [20]. However, if a message does not have any time properties attached to it, the *transition t* in the resulting Petri Net block will be designated as an *Immediate Transition*.

Rules 2 through 5, depicted by sections 2 through 5 in Figure 6 respectively, refer to the transformation of each InteractionOperators into a Petri Net block. However, since there are no intervals or durations that are attached to InteractionOperators, every *transition* in the resulting Petri Net block are designated as *Immediate Transitions*.

This paper also introduces a new rule for SD2PN, Rule 6 as illustrated in Figure 7, to map time properties in Sequence Diagrams that are not attached to a message into a

Petri Net block. This rule, although resulting in a Petri Net block similar to Rule 1, only has two possible scenarios. In cases where exists an interval in the Sequence Diagrams i.e. {10...30}, the *transition t* in the resulting Petri Net will have an interval of [10, 30] where else if there exist a duration in the Sequence Diagram i.e. {20}, the *transition t* will have an interval of [20].

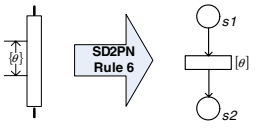


Fig. 7. Rule 6 of SD2PN

Once all the fragments are transformed into Petri Net blocks, they are amalgamated using two operations, which are referred to as *morph* and *substitute* defined in [10]. *Morph* is used for aggregating two Petri Net blocks to create larger Petri Nets. It can be seen that each Petri Net block has a single input and output *place*. Invoking *morph* with two Petri Net blocks merges the former's output *place* with the latter's input *place*. *Substitute* is used to replace a *placeholder* in the Petri Net blocks, such as *ph1*and *ph2* in sections 2 through 5 of Figure 6 with a different Petri Net block. This will be done repeatedly until there are no longer any *placeholders* left in the block.

It is shown in Theorem 1 of [10] that the model transformation generates only Free Choice Petri Nets that are predominantly used for effective and efficient analysis in enterprise system [2]. This result is preserved in this paper since the same Petri Net blocks are used. More information on SD2PN and the proof that it generates Free Choice Petri Net are available in [10].

4 Transformation Tool

Figure 8 depicts the architecture of the tool by showing the stages involved in the execution of a transformation [22]. The tool makes use of the XMI [23] representations of Sequence Diagrams which is provided by all mainstream UML tools such as [24, 25]. Using an XMI parser, the tool creates Java objects based on the Sequence Diagram metamodel. By utilizing SiTra [26], the tool transforms the Sequence Diagram objects into Petri Net objects based on the transformation rules.

Fig. 8. SD2PN Transformer

Finally, the functions *morph* and *substitute* introduced in [10] are used to aggregate the Petri Net objects, and thus create the Petri Net which corresponds to the original Sequence Diagram. The resulting Petri Net model can then be analysed by using a chosen Petri Net tool. The XML writer for the tool can be customized to correspond with a specific tool. This allows the designer to create a system completely in Sequence Diagram while still taking advantage of the analytical capabilities of Petri Net.

5 Case Study

This section presents a case study, which involves the specification and analysis of the Quality of Service (QoS) of a Personal Area Network (PAN) via SD2PN. The case study demonstrates the transformation of Sequence Diagrams into Timed Petri Nets and the use of the created Petri Nets to analyse QoS properties such as maximum delay.

5.1 Case Description

Figure 9 depicts a simplified PAN that has two stations and a Wireless Router that serves as an access point to the internet. In the router, the basic IEEE 802.11 Carrier Sense Multiple Access with Collision Avoidance (CSMA/CA) protocol is used [27]. Due to space constraints only the elements of the protocol that are relevant to QoS will be considered.

CSMA/CD assigns different *waiting time* to packets in order to manage the access of the stations to the medium. There are three different waiting times for various types of packets. The shortest waiting time for medium access is called *Short inter-frame spacing* (SIFS) which is used for short control messages or polling responses. The waiting time for time-bounded service such as a poll from the access point is considered *PCF inter-frame spacing* (PIFS) and the longest waiting time and lowest priority, *DCF inter-frame spacing* (DIFS) is used for asynchronous data services. There is a mechanism called *contention window* (CW), which is introduced in order to facilitate collision *avoidance*. The contention window makes use of an integer value that starts with $CW_{min} = 7$ and doubles every time a collision occurs. Every time a station tries to gain access to the medium, a random number is generated between 0 and CW and is added to the waiting time. This ensures that the stations do not send their packets at the same time. CW is doubled for every collision that occurs to accommodate a larger number of stations vying for the access of the medium. Readers are referred to [27] for more information.

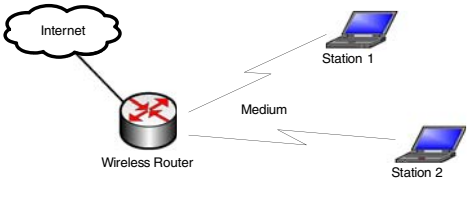


Fig. 9. Personal Area Network (PAN)

Several assumptions were made in this case in order to simplify matters and provide a better understanding of the tool. Firstly, the waiting time for all packets is constant and all packets are categorized as DIFS. Secondly, the CW is constant and does not increase, and since there are only two stations, the CW would be minimum, i.e. $CW_{min} = 7$. Thirdly, the packets are dropped after the unsuccessful tries from the station and each station sends only one packet. These assumptions do not invalidate the results of the analysis by any means; they just limit the scope of this case study.

5.2 Interaction Sequence Diagram

The Sequence Diagram in Figure 10 (a) gives an overview of how a station sends a packet to the medium in the IEEE 802.11 protocol. This Sequence Diagram also features the time properties in regards to the events that occur. The medium access control (MAC) layer of the station receives a packet from an application and registers it. It is then idle for the duration of the waiting time, which in the case of DIFS is 50μs. After idling, the MAC checks the status of the medium. If the medium is free, the station is able to send the packet across to the medium. However, if the medium is busy, the station will have to wait until the medium is free before idling again for 50μs. This is followed by a random time slot generated based on the CW, which in this case is between 1 and 7. Since a standard time slot is 20μs, this means that an additional waiting time of between 0 to 140μs referred to as the elapsed backoff time (bo_e) for a total waiting time of between 120μs and 240μs as shown in the Sequence Diagram. The MAC then checks the status of the medium again before either sending the packet across or waiting again. If the medium becomes busy while the station is still counting down the bo_e, then the counter stops and the remaining time is called residual backoff time (bo_r). As a result, the next waiting time will only be incremented by the value of bo_r and this increases the probability of a successful attempt from the stations' point of view. Note that the maximum waiting time in Figure 10 (a) is reduced with every attempt.

The diagram is a simplified overview of the events that take place. In reality, each of the events has multiple sub-events that occur in the background. For example, the details for the calculation of bo_e and bo_r are not shown in the Sequence Diagram and are all grouped under the event *waitForAccess*.

5.3 Model Transformation

A set of Petri Nets was generated by taking the Sequence Diagrams in Figure 10 (a) as the source model in SD2PN; they represent the process of sending packets in IEEE 802.11. Figure 10 (b) gives the overview of the result of the transformation process as a mapping from the Sequence Diagram in Figure 10 (a). The Petri Nets preserve the time constraints specified in the sequence diagrams and will allow for various forms of QoS analysis to be performed.

The Petri Net in Figure 10 (b) models the behaviour of one station trying to gain access to the medium to send a packet. In cases where more than one station are trying to access the medium, the Petri Net in Figure 10 (b) is duplicated for each station and is synthesized i.e. merged using a bottom-up synthesis technique [28]. Although the tool does not currently support *synthesis*, we are actively working towards its integration into the system.

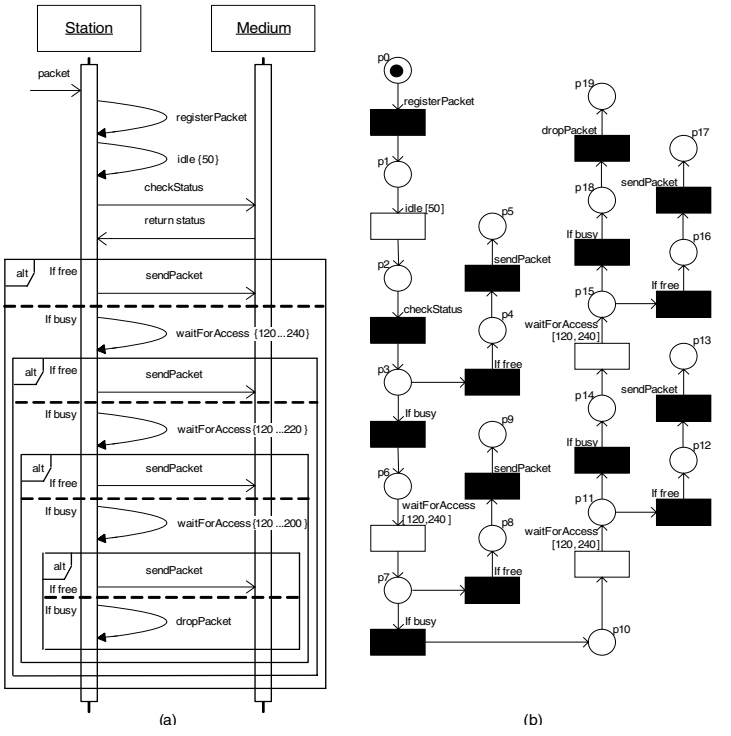


Fig. 10. Model of (a) Sequence Diagram for a station in PAN and (b) its equivalent Petri Net

5.4 Model Analysis

The Petri Net that results from the transformation lends itself to various types of analysis such as deadlock detection, liveness and safeness analysis [29]. Time sensitive analysis such as tangible states analysis and throughput analysis are also possible [8]. In this case study, throughput analysis will be used to analyse maximum delay as one aspect of QoS.

The maximum delay is calculated based on how long it takes for a station to gain access to the medium *(sendPacket).* For the case of a single station shown in Figure 10 (a), the maximum delay will be 50μs since there is no contention with other stations. However, in the case where there are two stations competing for access to the medium, the maximum waiting time for a station is 290μs as shown in Figure 11. This calculation is based on the flow of events in the Petri Net. Since there are two stations, the Petri Net in Figure 10 (b) is duplicated to model the second station. After registering the packet (firing of *registerPacket* transition), in Figure 10 (b), both stations will face a mandatory idle time of 50μs (firing of *idle* transition) before checking the status of the medium. Following that, only one station will be able to gain access to the medium while the other will have to wait between 120μs and 240μs (firing of *waitForAccess* transition), thus a maximum waiting time of 290μs (= 240μs + 50μs).

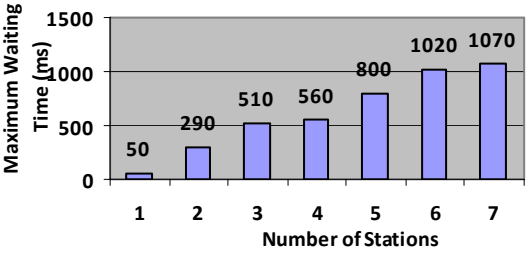


Fig. 11. Maximum Waiting Time analysis result

The graph in Figure 11 indicates the maximum delay that a station may face before gaining access to the medium to send a packet. The number of stations is limited to 7 to ensure there are no collisions; this is based on the previous assumption that the CW does not increase. Such information can be used to analyse the choice of protocols for interaction within the system.

6 Discussion

Many researchers have highlighted the trade-off between the ease of use of UML and its lack of precision. Recent work in this area has been marked by a concerted effort aimed at enhancing UML by incorporating formal methods techniques [30-34]. Formalisation offers many advantages including the ability to analyse a model via techniques such as model checking and theorem proving in order to ensure correct specification. The introduction of logical and timing constraints into a model, in particular, facilitates the investigation of non-functional aspects of the system such as QoS and security. It has been noted however that formalisation is often achieved at the expense of simplicity and that the main challenge is to strike a balance between precision and ease of use.

Formalising UML is an active area of research. For example, Evans *et al* [30] propose the use of Z as the underlying semantics for Class Diagrams to deal with the static aspects of models. Küster-Filipe [35] presents a semantics for Sequence Diagrams based on Labelled Event Structures which are used to prove the correctness of SD2PN [10]. The approach adopted in this paper, although it promotes the use of formal methods, differs from the latter in a significant way. It relies on model transformation and formal method analysis tools to facilitate *automated* analysis.

Model transformation has also received considerable attention. Kim [32] proposes transforming both Class Diagrams and State Machines into Object-Z using MDA technology. To the best of our knowledge, this transformation has not been implemented yet. A similar approach is adopted in [34] and [33] which transform Class Diagrams and OCL Constraints into the formal language B [36]. In particular, [33] proposes a UML profile for B called UML-B and the automation of the transformation with a tool called U2B. A major feature of of this approach is that it makes use of B provers to check the conformance of the operations' pre and post conditions to the invariants of the model. The main difficulty with provers, as underlined in [33], is that

even semi-automatic provers assume a substantial amount of knowledge from the user. In contrast the approach presented in this paper aims to limit the reliance on formal method expertise such that a designer may model a system conveniently in Sequence Diagram and still manage to use the analysis capabilities of Petri Nets.

The proposed framework transforms the UML Sequence Diagrams into Timed Petri Nets and takes advantage of their suitability for formal analysis. The transformation produces Free Choice Petri Nets, which support the investigation of various properties such as liveness, safeness and deadlocks detection [29]. It is possible to integrate existing Petri Net tools into the tool set, so that for a created UML Sequence Diagram, through a chain of tools, the user can *automatically* receive feedback on, among others, the liveness, safeness and deadlock freeness of the model. This combination of formalisms, tools and model transformations is bound to reduce the cognitive load on users since a thorough understanding of the underlying formal structure of the model is no longer required. Moreover, Free Choice Petri Nets are also proving to be particularly suitable for the analysis of large-scale systems [1, 2], an important feature that widens the scope of the application of the proposed framework to encompass similar systems. In addition to the structural and behavioural analysis, the time properties included in this paper will also enable performance analysis to be conducted on the system. Analysis like maximum throughput, density probability, interval, cycle time [8, 16] and many other time related analysis can be carried out.

It is possible to augmenting Sequence Diagrams with logical constraints as pre and post conditions for each execution of events. Such constraints can be expressed in languages such as OCL [12] and mapped by extending the model transformation presented in the paper. This would result in Coloured Petri Nets [7] which are an extension of Petri Nets. Coloured Petri Nets have been investigated extensively and various tools, such as CPNTools [9], have been developed for their analysis.

7 Conclusion

This paper has presented a framework of applying Model Driven Development for transforming time augmented Sequence Diagrams into Timed Petri Nets. This model transformation serves to bridge the gap between the design and analysis phases of a system, thus enabling a designer to conveniently design a system in UML Sequence Diagram while taking advantage of Petri Nets' strong mathemathical foundations to analyse the model. Furthermore, the addition of time properties into the model transformation allows for performance analysis such as execution time computation and throughput analysis on top of the established structural and behavioral analysis capabilities of Petri Nets. The presented approach has been evaluated successfully with the help of an example of a Personal Area Network.

Acknowledgement. The authors wish to thank Behrang Saroui for his part in the tool development.

References

1. van der Aalst, W.M.P.: The Application of Petri Nets for Workflow Management. The Journal of Circuits, Systems and Computers 8(1), 21–66 (1998)
2. Vanhatalo, J., Volzer, H., Leymann, F.: Faster and More Focussed Control-Flow Analysis for Business Process Models Through SESE Decomposition. In: Fifth International Conference on Service Oriented Computing, pp. 43–55. Springer, Vienna (2007)
3. Anastasakis, K., et al.: UML2Alloy: a Challenging Model Transformation. In: ACM/IEEE 10th international confernece on Model Driven Engineering Languages and Systems (2007)
4. Juerjens, J.: Secure Systems Development With UML. Springer, Heidelberg (2004)
5. Jackson, D.: Software Abstractions Logic, Language, and Analysis. MIT Press, Cambridge (2006)
6. Spivey, J.M.: The Z Notation: a reference manual. Prentice Hall, Englewood Cliffs (2001), http://spivey.oriel.ox.ac.uk/~mike/zrm/
7. Murata, T.: Petri Nets: Properties, Analysis and Applications. Proceedings of the IEEE 77(4), 541–580 (1989)
8. Bonet, P., et al.: PIPE v2.5: a Petri Net Tool for Performance Modeling, in XXXIii Conferencia Latinoaméricana de Informática (2007)
9. CPNTools, Computer Tool for Coloured Petri Nets, http://wiki.daimi.au.dk/cpntools/
10. Ameedeen, M.A., Bordbar, B.: A Model Driven Approach to Represent Sequence Diagrams as Free Choice Petri Nets. In: 12th International IEEE Enterprise Distributed Object Computing Conference (EDOC), München, Germany, pp. 213–221 (2008)
11. OMG, OMG Unified Modelling Language (UML) Superstructure 2.1 (2007), http://www.omg.org
12. OMG, UML 2.0 OCL 2nd revised submission (2003), http://www.omg.org
13. Douglass, B.P.: Doing Hard Time: Developing Real-time Systems with UML, Objects, Frameworks and Patterns. Object Technology Series. Addison Wesley, Reading (1999)
14. Störrle, H.: Trace Semantics of UML 2.0 Interactions, University of Munich (2004)
15. Wang, J.: Timed Petri Nets: Theory and Application. Springer, Heidelberg (1998)
16. Jensen, K., Kristensen, L.M., Wells, L.: Coloured Petri Nets and CPN Tools for modelling and validation of concurrent systems. International Journal on Software Tools for Technology Transfer (STTT) (2007)
17. Stahl, T., Volter, M.: Model Driven Software Development; technology engineering management. Wiley, Chichester (2006)
18. Jouault, F., Kurtev, I.: Transforming Models with ATL. In: Bruel, J.-M. (ed.) MoDELS 2005. LNCS, vol. 3844, pp. 128–138. Springer, Heidelberg (2006)
19. Muller, P.A., Fleurey, F., J´ez´equel, J.M.: Weaving Executability into Object-Oriented Meta-languages. In: Briand, L.C., Williams, C. (eds.) MoDELS 2005. LNCS, vol. 3713, pp. 264–278. Springer, Heidelberg (2005)
20. Akehurst, D.H., et al.: SiTra: Simple Transformations in Java. In: ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (formerly the UML series of conferences), Genova, Italy (2006)
21. OMG, MOF 2.0 Query/View/Transformation (QVT) Specification (2008), http://www.omg.org
22. Saroui, B.S.: Model Transformation from Sequence Diagrams to Petri Nets. University of Birmingham, Birmingham (2008)
23. XMI, XML Metadata Interchange (XMI), v2.1 (2005), http://www.omg.org

24. ArgoUML, ArgoUML web site (2005), http://sourceforge.net/projects/argouml
25. Poseidon. Poseidon for UML, from Gentleware (2006), http://www.gentleware.com/
26. Akehurst, D.H., et al.: SiTra: Simple Transformations in Java. In: ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems (2006)
27. Schiller, J.H.: Mobile Communications. Pearson Education, London (2003)
28. Agerwala, T., Choed-Amphai, Y.-C.: A synthesis rule for concurrent systems. In: ACM IEEE Design Automation Conference (1978)
29. Desel, J., Esparza, J.: Free Choice Petri Nets. Cambridge University Press, Cambridge (1995)
30. Evans, A.F., Robert, Grant, E.: Towards Formal Reasoning with UML Models. In: Proceedings of the OOPSLA 1999 Workshop on Behavioral Semantics (1999)
31. Kim, D., et al.: A UML-Based Metamodeling Language to Specify Design Patterns (2003)
32. Kim, S.-K.: A Metamodel-based Approach to Integrate Object-Oriented Graphical and Formal Specification Techniques. University of Queensland, Brisbane (2002)
33. Snook, C., Butler, M.: UML-B: Formal modelling and design aided by UML. In: ACM Transactions on Software Engineering and Methodology (2006)
34. Marcano, R., Lévy, N.: Transformation Rules of OCL Constraints into B Formal Expressions. In: 5th International Conference on the Unified Modeling Language, Dresden, Germany (2002)
35. Küster-Filipe, J.: Modelling concurrent interactions. Theoretical Computer Science 351(2), 203–220 (2006)
36. Abrial, J.-R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)

A Hybrid Graphical and Textual Notation and Editor for UML Actions

Anis Charfi, Artur Schmidt, and Axel Spriestersbach

SAP Research CEC Darmstadt
Darmstadt, Germany

Abstract. The aim of UML Actions is to allow detailed and platform independent modeling of object-oriented behavior. However, behavior modeling with UML actions is still not adopted due to three main reasons. First, UML defines only the abstract syntax of the actions and no concrete syntax. Second, users have to work directly with the very complex meta-model of UML actions. Third, all existing tools do not provide appropriate support for creating and working with action-based behavior models. In this paper, we propose a hybrid language combining the strengths of textual and graphical notations in one concrete syntax for UML actions and present a supporting editor. We also report on a user study that was conducted to evaluate the notation and the editor.

1 Introduction

In the theory of Model-Driven Architecture (MDA) [1], it should be possible to specify a software system including its structure and behavior at the platform-independent modelling level (PIM), then refine the PIM to one or more platform-specific models (PSMs), and after that generate code for different target platforms and languages. However, in practice, only structure modeling is well supported at the PIM level whereas behavior modeling has only limited support. This is partly due to the lack of appropriate means in the Unified Modeling Language (UML) until the version 1.5 for detailed behavior modeling (i.e., for modeling the content of a method body at the PIM level). This situation gave rise to MDA-light [2], which is the more prominent instance of MDA in practice. In this approach, the PIM contains only the structural part of a system. The PIM (or a transformed version of it) is used to generate code skeletons, in which the behavior has to be inserted manually in a specific target language and in a platform-specific way. Such an approach breaks the portability, interoperability, and reuse objectives of MDA.

To support detailed behavior modeling, the OMG introduced the action semantics specification since UML 1.5. Actions are primitives for modeling behavior independently of programming languages and platforms. Actions are contained in UML Behavior, e.g. in a UML Activity that defines the behavior of an operation. There are predefined actions for various purposes, e.g., to create object instances, to call an operation, etc.

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, pp. 237–252, 2009.

UML Action Semantics has the potential to fill the gap between MDA-light and real MDA as it provides means for behavior modeling at the PIM level. Nevertheless, there are still four problems that need to be tackled before behavior modeling with UML actions becomes widely adopted. First, UML 2 does not specify a concrete syntax for actions: it only states that actions should be represented by rectangles with rounded corners. This problem has been recognized by the OMG, which issued a call for proposals requesting a concrete syntax for UML actions [3]. Second, in lack of a concrete syntax the users have to understand and work directly with the big and complex meta-model (or meta-muddle as called in [4]) of UML. For instance, one needs to understand and know the different types of input and output pins each action has and which ones must be set depending on the action type. Third, the tool support for UML Actions is very limited and only a few tools support the meta-model.

In this paper, we present a hybrid language combining the strengths of textual and graphical notations and an editor that tackle the three problems discussed in the last paragraph. This work was performed in the context of the EU project VIDE [5], which focuses on methodologies and tools for behavioral modeling especially at the CIM and PIM layers. Our contribution is three-fold. First, we propose a visual notation for the most used actions. This notation hides several complexities of the meta-model. Second, we present an editor supporting the proposed notation and providing features that ease working with action models such as auto-layouting and auto-completion. Third, we evaluated the notation and the editor through a user study that compares learning and using the notation and its editor with to learning and using a new textual programming language.

The remainder of this paper is structured as follows: Section 2 gives some background. Section 3 motivates the need for a notation for UML Actions and better tool support. Section 4 presents the proposed notation and Section 5 presents the editor. Section 6 discusses the user study and its results. Section 7 studies related work and Section 8 concludes the paper.

2 Background

This section gives a brief introduction to MDA, UML Action Semantics, and the PIM level language of VIDE.

2.1 Model-Driven Architecture

The MDA [1] development process is centered around models and model transformations. It defines three main types of models that are refined gradually until generating the application code. A Computation-Independent Model (CIM) is a high-level domain model that defines the requirements and usage of the software system without giving any details on how the system is implemented. The CIM is then refined in to a Platform-Independent Model (PIM), which defines the functionality of the software system in an abstract way without any platform-specific details. The PIM is transformed, possibly using a Platform Description

Model (PDM), into a Platform Specific Model (PSM). A PSM is a more concrete model that defines the system functionality for a specific platform. MDA allows to model a software system once (at the PIM level) and then generate the implementation to several target programming languages and target platforms. The last step in the MDA process is the code generation out of the PSM model.

2.2 UML Action Semantics

Since version 2.0, UML has a more consolidated specification for Action Semantics. UML actions are atomic primitives for behavior modeling that can be used within activity diagrams. Actions are contained in UML Behavior, e.g. in a UML Activity that defines the behavior of an operation. With actions it is possible to specify the implementation of a method or constructor body at the PIM level.

An action has input pins and output pins, which allow the passing of data between various actions by using object flow connections. The order of executing the actions can be defined either using a Petri Net like flow concept (i.e., control flow connections) or using nodes that structure the contained actions (such as conditional node, sequence node, etc). UML 2 defines several classes of actions for different purposes, e.g., for object manipulation (i.e., to create or destroy object instances), for calling operations, for manipulating object properties or variables (i.e., to read and set properties and variables), for manipulating associations (i.e., to create and destroy links between objects), etc. UML does not specify a concrete syntax for actions. It only states that they can be represented through rectangles with rounded corners.

2.3 The VIDE PIM Language

In the context of the VIDE project, we defined a PIM level language that focuses on behavioral models for business and data-intense applications. This language uses UML actions together with OCL expressions and queries [6] as specified in the Eclipse MDT project [7]. The VIDE PIM language covers most concepts of UML 2 structural modeling, such as packages, classes, operations, properties, and associations. Further, this language includes the following UML 2 action types: invocation actions, object creation/destruction actions, link actions, structural feature actions, and variable actions. Some action types are not in the scope of VIDE namely actions for reading system state as all side-effect free queries are expressed in OCL, actions for signal processing, and actions for changing the classification of objects.

3 Problem Statement

In this section, we introduce a simple example that will be used throughout the paper for illustration. Then, we discuss three problems that hinder the adoption of UML actions for behavior modeling in practice.

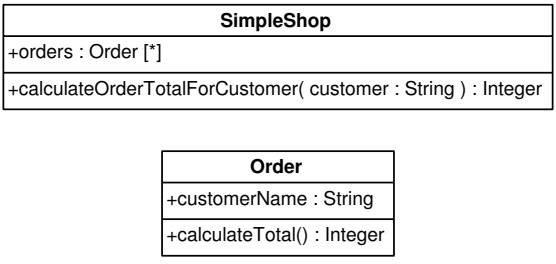


Fig. 1. The classes of the simple shop scenario

3.1 Motivating Example

Figure 1 depicts the classes of a simple shop scenario: the shop is represented by the class *SimpleShop*. This class has a collection of orders of the type *Order*. Each order belongs to a customer and provides a method to calculate its total. The method *calculateOrderTotalForCustomer* will be used to illustrate behavior modeling: it finds all orders belonging to the customer specified as parameter and sums up the totals of his orders, which are returned by the method *calculateTotal*. A reference implementation of the method *calculateOrderTotalForCustomer* in Java is shown in Listing 1.

```
int calculateOrderTotalForCustomer(String customer) {
    int total = 0;
    for (Order order : orders) {
        if (order.customerName.equals(customer)) {
            total = total + order.calculateTotal();
    }}
    return total;
}
```

Listing 1. Java implementation of *calculateOrderTotalForCustomer* method.

Although it is a small scenario, the implementation of the method *calculateOrderTotalForCustomer* contains iteration, a conditional statement, operation invocation and other basic constructs that are usually used in programming.

3.2 Limitations of UML Action Semantics

With UML Actions it is now possible to model whole applications at the PIM level. However, there are three problems that hinder the adoption and usage of UML actions in practice.

Lack of Concrete Syntax for Actions. UML does not define a concrete syntax for actions. The tool vendors also did not propose any more advanced concrete syntax. E.g., in Topcased [8], a leading open source MDA tool, actions are represented by rounded rectangles with a label that is set by default to the type of the action. In MagicDraw [9], a representative for commercial MDA tools, actions are represented in the same way with an additional stereotype indicating

the type of the action. Having semantically different actions represented almost in the same way hinders the understandability of action models especially when these models get bigger. In addition, not all relevant information is immediately visible in the notations of state of the art editors, e.g., in an *AddVariableValueAction* in Topcased the variable is not visible in the action representation. Such information can only be seen/edited in a separate properties view.

The OMG has recognized that without a concrete syntax users will not adopt UML actions. Therefore, it has issued a call for proposals requesting a textual concrete syntax for the UML action language [3]. However, a textual notation might not be appropriate for certain users such as modelers or domain experts who have domain knowledge and an understanding of UML and its concepts, but little experience in using textual programming languages. For those users a graphical notation might be more appropriate.

Complexity and Size of the Meta-model. As UML does not define a concrete syntax for UML actions, the users have to work at the abstract syntax level, i.e., they face directly all complexities of the UML meta-model and especially the parts about actions and activity diagrams. The users must know all actions and understand their properties. They must also know what input and output pins each action has and what pins and properties must be set.

Insufficient Tool Support. In the context of the VIDE project, we have examined many UML tools with respect to their visual notations for UML actions and the user assistance during the creation of action models [10]. To demonstrate the support available in state of the art editors Figure 2 shows the action models of the method *calculateOrderTotalForCustomer* from the running example created with MagicDraw UML version 16 (right) and with Topcased version 2.2 (left).

Looking at the two diagrams and comparing them to the Java version in Listing 1, one sees that it is more difficult to find out what the action models do. It takes more time to inspect the whole diagram as one has to follow the control and data flows explicitly. To make the diagram more readable, the Topcased user has to update the labels manually as they are set by default to the action type. The users are not supported at all when they need to set properties or enter expressions. Further, all pins (the little boxes adjacent to the actions) have to be created manually. This requires good knowledge of the meta-model. The control flow and data flow edges have to be created and connected manually as well. In addition, the tools are not able to automatically layout whole diagrams in such a way that they are readable. As a result, the user has to care about this burden and to update the layout on changes, which can be very time-consuming.

A fourth problem that hinders the adoption of UML Action Semantics is the lack of mappings to programming languages and also the lack of supporting model compilers. This problem has been tackled in the context of the VIDE project through the definition of mappings to languages such as Java [11] or ODRA [12] and the implementation of supporting code generators.

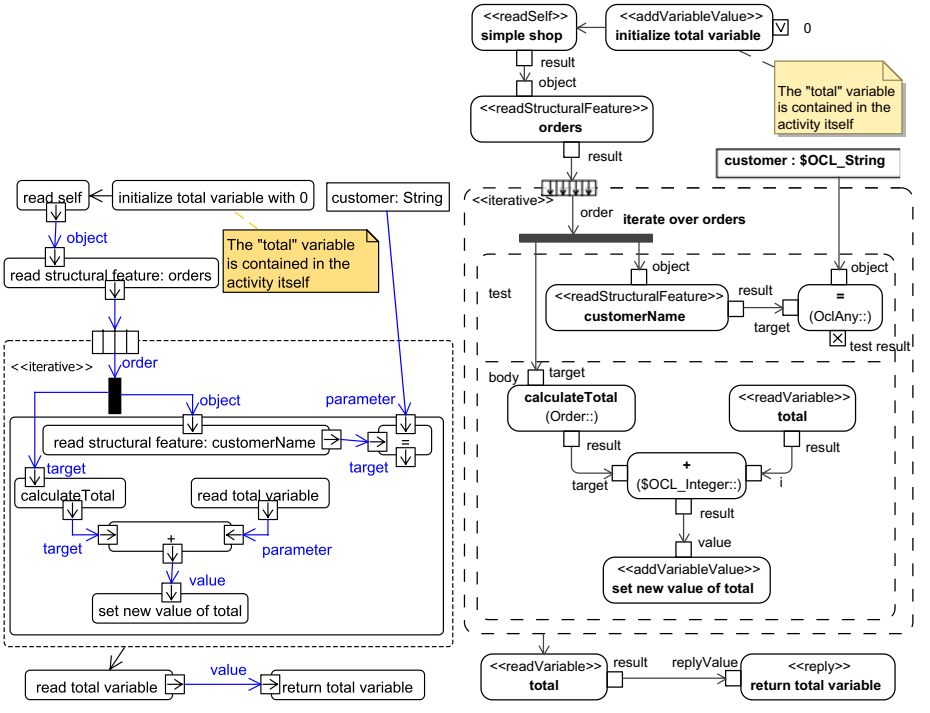


Fig. 2. The method *calculateOrderTotalForCustomer* implemented using an activity diagram in TOPCASED (left) and in MagicDraw (right)

4 A Hybrid Notation for UML Actions

In the proposed notation actions are realized as *templates* that contain placeholders for expressions that correspond to frequently needed action properties and/or pins. Thus, the frequently used action properties/pins are visible in the notation and can be edited directly. As a result, the users do not have to know all details of the UML meta-model that are relevant for a certain action and the meta-model complexity is hidden. Since UML defines more than 40 different actions, we defined a notation only for the actions included in the VIDE language (cf. Sect. 2), i.e., actions for object creation, method invocation, access and manipulation of object properties, as well as definition and manipulation of variables. Further, our notation supports three structured control flow nodes for specifying control flow. Data flow is not shown in the notation as it is expressed indirectly through variables. The graphical representation of certain actions is partly inspired by different UML diagrams.

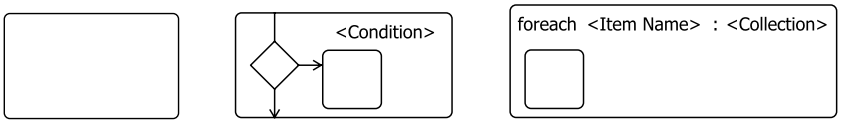
Moreover, the proposed hybrid notation is inspired in several regards by textual notations. First, the notation constraints the placement of action to be in a sequential order like a textual notation. Second, control flow is expressed

implicitly via structured nodes and data flow is expressed indirectly via variables. These design decisions allow us to exploit the textual readership skills of the users and also enable automatic layout. Scoping is indicated in our notation by spatial containment.

In the following, we present the visual representation for selected actions. A more comprehensive presentation can be found in [13], which also maps the visual representation to the abstract syntax. Due to space limitations, we present here the correspondence of visual syntax and abstract syntax only for the call operation action.

4.1 Control Flow

Structural nodes are elements of the UML meta-model that define control flow and which can contain further actions. Fig. 3 shows the graphical notation of three such elements: *sequence node*, *conditional node*, and *expansion region*.



(a) Sequence node (b) Conditional node (c) Expansion region

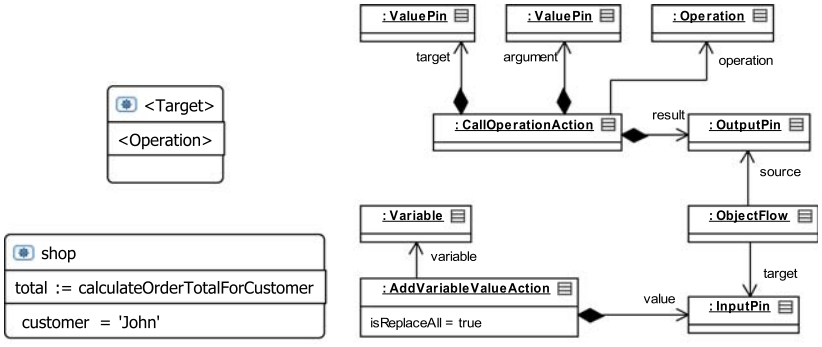
Fig. 3. Visual notation for control flow nodes

A *sequence node* executes the nested elements sequentially according to their order in the *SequenceNode*. It also defines the scope for variables as variables are only accessible in the *SequenceNode*, in which they are defined and in its child nodes. A *conditional node* realizes the conditional execution of a sequence node (the "body") based on a condition, which is a boolean expression that can be viewed/edited. An *expansion region* allows to iterate over collections. In the proposed notation, the user must specify an expression that identifies the collection and the name of a temporary variable that will hold the value of each collection item during the iteration. The nested *sequence node* including all actions it contains will be executed for each item.

4.2 Basic Actions

The following actions provide basic functionality such as calling an operation, creating an object, and returning a value.

The *Call Operation Action:* this action invokes an operation on an object or a class (if the operation is static). As shown in Fig. 4(a), the target (an object or class) and then the operation must be specified by the user in the upper compartments (upper figure). If the operation returns a result a variable will be created automatically. If the operation expects arguments parameter compartments will be added. The lower part of Fig. 4(a) shows a call operation action that calls the

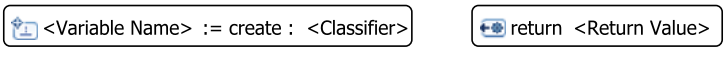


(a) Notation (b) Abstract syntax

Fig. 4. Notation and abstract syntax for a *CallOperationAction*

operation *calculateOrderTotalForCustomer* from the running example. As this operation returns a value a variable will be automatically created to which the result is assigned. The name of this variable is set by the user. For each argument of the operation there is an entry text field in the lowest compartment of the notation where the argument has to be specified (in the example there is only one parameter).

Figure 4(b) shows the abstract syntax corresponding to the visual representation of Figure 4(a). The chain of instances starting with *result* and ending with *variable* assigns the returned result to the variable and is only present if the *operation* referred from the *CallOperationAction* returns a result. For both arguments of the *CallOperationAction* a *ValuePin* (*target* and *argument*) that hold an expression is created.



(a) (b)

Fig. 5. Notation for *CreateObjectAction* and *ReplyAction*

The notation of *Create Object Action* (Fig. 5(a)) requires the user to specify the class that will be instantiated and a variable that will created to hold the new object. The notation of the *Reply Action*, which returns the result of an operation, is shown in Fig. 5(b). Only the expression for the return value must be specified.

4.3 Variable Actions

A variable can be declared using the *variable* element. Variables can be multi-valued, i.e., contain more than one value and the values can be ordered and/or

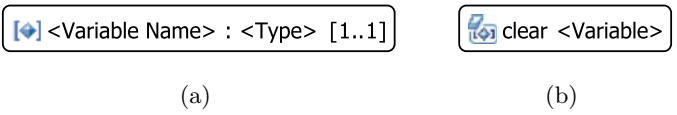


Fig. 6. Notation for variable declaration and clearVariableAction

unique. Figure 6(a) shows the notation for a variable declaration. The notation allows the users to set the variable name and its type and multiplicity.

The notation of *Add Variable Value Action* is shown in Fig. 7(a). This action allows a value to be inserted into or assigned to a variable. The user has to specify the variable name and a value expression. If the specified variable is multivalued, the value will be added to the variable, otherwise the existing value will be removed before adding the new value, which results in a simple assignment. If the specified variable is ordered (and multivalued), an additional text field appears in the notation where the user specifies the insert position.

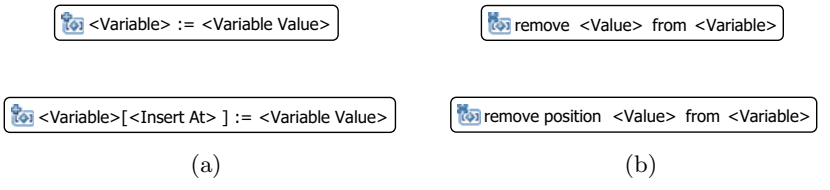


Fig. 7. Notation of AddVariableValueAction and RemoveVariableValueAction

The *Remove Variable Value Action* removes a value from a variable. It is the counterpart to the *AddVariableValueAction*. As Fig. 7(b) shows, a variable must be specified in the notation. If that variable is ordered and not unique, a position expression must be specified, i.e., the index at which a value in the variable will be removed (as indicated by the lower notation in the figure). In addition the value to be removed must be specified.

4.4 Structural Feature Actions

These actions manipulate the values of structural features of an object, such as class properties and association ends (which are also properties, but they are not contained in classes). These actions are similar to variable actions.

The *Add Structural Feature Value Action* adds a value to a structural feature of an object. Exactly like an add variable value action, this action inserts or assigns a value to the structural feature. The notation for both usages is shown in Figure 8(a): the upper notation is used for simple assignments and if no insertion position must be specified, i.e., for multivalued but unordered structural features. The lower notation is used when the insertion position must be specified, i.e., in case of multivalued and ordered structural features.

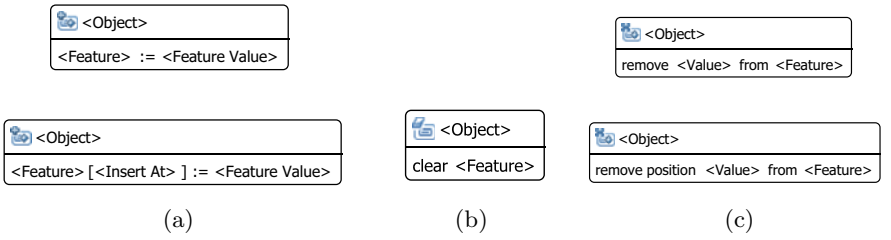


Fig. 8. Notation of structural feature actions

Fig. 8(b) shows the notation of the *Clear Structural Feature Action*, which removes all values from a structural feature of an object. Only the object and one of its features must be specified.

Fig. 8(c) shows the notation of *Remove Structural Feature Value Action*, which is the counterpart to the *AddStructuralFeatureValueAction*. This action removes a value from a structural feature. The object and one of its structural features have to be specified in this notation. A position may have to be specified as in the case of remove variable value action.

5 A Visual Editor for UML Actions

We implemented a visual editor for UML actions based on the notation presented in Section 4. A screen shot of the editor showing the action model of the method *calculateOrderTotalForCustomer* is presented in Fig. 9. A video showing the tool can be found at [14].

After creating a UML class diagram of the application, the user can start our editor from the UML tree editor of Topcased [8] by right-clicking the activity that implements a certain method and start modeling the behavior of that method. The user can create actions by selecting them in the palette and clicking on the element in the editor area where the new actions should be inserted.

The editor provides a range of features that make it user-friendly and easy to use. Thus, it enables users to focus more on their actual task, i.e., on behavior modeling. The four most important features of the editor are:

- *Automatic layout:* The editor does the layout of actions automatically, taking this burden away from the user. It is always clear how to read an action model: from the top to the bottom.
- *Auto-completion:* The editor provides proposals when the user enters expressions into a text field, e.g., the name of a variable. Only relevant proposals are displayed based on the given context (e.g., the current scope of the edited action) and the already entered text. This feature frees the user from having to remember all available objects and from having to switch to other models such as the structural model.
- *Drag & drop:* Following the direct manipulation interaction style, this feature allows moving actions between different *SequenceNode*s and reordering

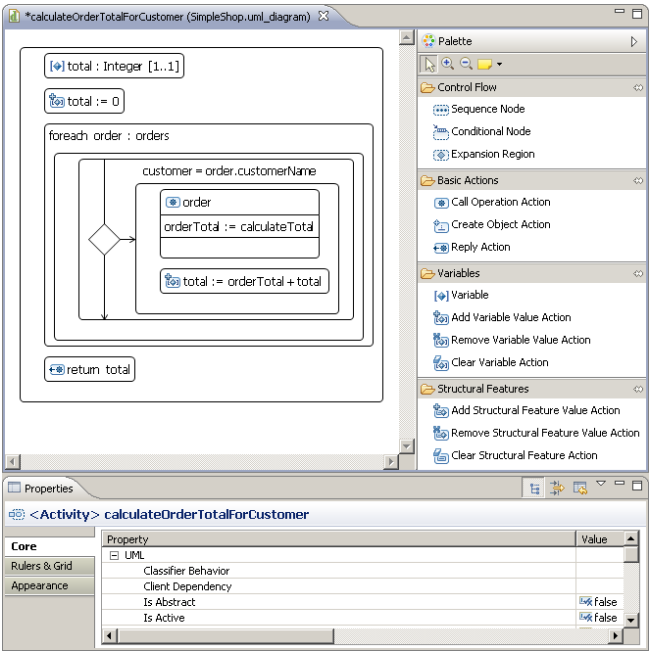


Fig. 9. The visual editor for UML actions

actions within a *SequenceNode*. The moving and reordering functionality is essential for the creation of behavior models with the visual editor.

– *Undo/redo:* The editor provides the possibility to undo or redo changes to the model. Knowing that the user can restore previous states of the "code", he is not afraid of making changes to the behavior and to experiment with it. This is important since the editor should provide means to incrementally develop the behavior instead of having to "put in the solution".

The visual editor was implemented as an Eclipse plugin that is generated by Eclipse Graphical Modeling Framework (GMF version 2.1.0) [15], which allows fast creation of visual editors and works with the Ecore meta-models of Eclipse Modeling Framework (EMF) [16]. An EMF-based implementation of UML 2.1 and OCL 2.0 is available from the Eclipse MDT project [7]. Further, basing the editor on Eclipse and EMF allows its integration with the other Eclipse/EMF based VIDE tools such as the textual editor of action models [6] and the code generators [11,5] from UML actions to Java, ODRA, and ABAP. This integration allows modeling behavior at the PIM level with the visual and textual editors and then generating complete and compile-ready applications.

6 User Study

To see whether the proposed hybrid notation really brings the expected advantages over a textual notation, the hybrid notation (and the editor) was compared

with Smalltalk and Squeak as IDE in a user study. Smalltalk was chosen because it supports object-oriented programming (just as the VIDE language) and because it has a very similar set of language features (at least for the tasks in this study). Furthermore, it is relatively unproblematic to find participants who do not know this language. The study was designed with the following research questions in mind: How fast can the hybrid notation be learned? What mistakes are prevented or facilitated by the visual notation? and in which situations is the hybrid notation better than a pure textual notation.

Of particular interest was how fast the notation can be learned. Participants were given simple tasks from the business applications domain and they were observed while they were making their first steps with the new language, trying to solve the tasks. As an indication of how fast each language was learned the time taken to solve these tasks was measured. The underlying hypothesis was that the participants will learn the hybrid notation faster and thus need less time to solve the tasks compared to the textual notation.

6.1 Experimental Design

In total 16 participants took part in the study, ranging from less experienced to advanced programmers. The participants were all familiar with UML as a modelling language for static structures, but they did not know UML actions, neither did they know Smalltalk. About 2/3 of the participants have never used a visual programming language before.

A *within-groups* design was chosen, i.e., every participant solved the same tasks using both languages. This way less participants are required (for statistical relevance) and the variation of personal programming skills is less of a problem. To account for the transfer of the solution into the other language, every second participant started with Smalltalk. Additionally, the tasks were designed in such a way that they were not challenging in an algorithmic sense.

Each session lasted ca. 60 to 90 minutes and there were three tasks in total. A framework with all necessary classes and method stubs was provided in both, Smalltalk and UML. It was an extended version of the scenario in the motivating example (Figure 1). The first task required to implement the *calculateOrderTotalForCustomer* method from scratch. The other two tasks were smaller and required to identify (and syntactically understand) certain constructs in the given method bodies and to remove these or to add new elements to them. All participants were given an unlimited amount of time to complete each task. The time measurement was started as soon as the participant started working on the task and stopped as soon as the participant was certain that the program was done. There was only one condition for the returned solution: for Smalltalk, the program had to compile and for VIDE, all fields of all actions had to be filled out (text input is compiled immediately). As a consequence of this condition, there were no syntax errors in the programs. To collect information about the kinds of syntax errors the participants made, they were observed and video-tapped while solving the tasks.

6.2 Results and Discussion

As shown in Figure 10, all tasks were solved faster with the hybrid notation and the respective editor. The statistical significance of this observation is supported by a paired, two-tailed Student's t-test: the null hypothesis (i.e., the performance of both notations does not significantly differ) could be rejected at the 0.1% level for task 1 and 2 and at the 1% level for task 3. Hence, the differences can be classified as statistically very highly significant and highly significant, respectively.

We observed a clear trend during the study, which is backed by the statistical significance: participants had fewer difficulties with the proposed hybrid notation than with the syntax of Smalltalk. In fact, the visual nature of the hybrid notation helped to reduce syntax hassle, resulting in significantly lower times needed to generate the solutions. According to observations during the solutions, the proposed notation mainly prevents several types of errors, such as wrong usage of blocks (missing or superfluous brackets) and missing termination statements. Another factor for the significantly lower times is presumably that actions are represented by "templates" in the hybrid notation, making it easier to learn the notation as the exact syntax for actions does not have to be remembered by the sers. The saved solutions of the participants were also investigated with regard to semantical correctness. The great majority of errors in the solutions were glitches. Most participants made them consistently between the two languages. Thus, no significant differences between the quality of the solutions generated with the hybrid notation and Smalltalk were found.

All in all, the data presented here allows to draw the conclusion that the hybrid notation was easier to learn for the participants than a new textual programming language. Another evaluation comparing the hybrid notation and the respective editor with the textual notation of UML actions and a respective editor [6] as well as a comparison with Java can be found in [10]. One possible criticism to the presented evaluation may be that the chosen example is rather

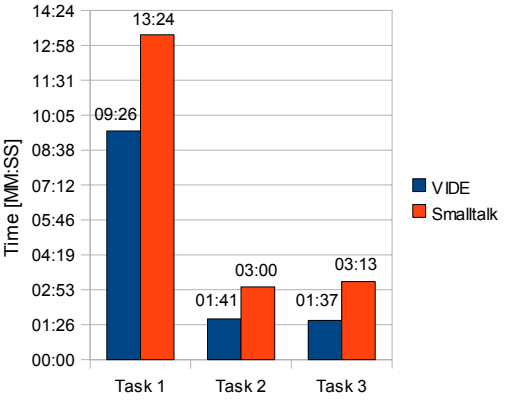


Fig. 10. The mean times for each task and language

small and our results may not scale to larger examples. However, a complex behaviour model can be always simplified by extracting some behavior into a new method and just calling it from the original behavior model. Further, the hybrid notation addresses the scalability challenges by allowing the graphical parts to be collapsed to their textual representation which is usually more compact.

7 Related Work

Behavior modeling on the PIM level is not new, there have been a number of approaches before that address it. Often they are realized as proprietary extensions of UML, such as Action Specification Language [17] (which is a textual language for behavior modeling) and Executable UML [18] (which uses state machines for behavior modeling). The problem is that these languages are not inter-operable, i.e., cannot be interchanged between different tools and also do not integrate well with structural models such as class diagrams. With UML 2 actions these problems are solved. However, a major limitation of the current UML standard is the lack of concrete syntaxes for UML actions (see also section 3.2). Most existing action languages for UML are textual and many of them are aligned to an existing programming language such as Java and ABL [19].

Ambler [20] observes that developers prefer textual notations for modeling. This is supported by Fowler [21] and several other practitioners that state that the diagrammatic way of code construction is incomparably slower than textual notations. On the other hand, Hailpern [22] states that UML has a potential to cope with the challenges of software modeling as there is a growing tool landscape and people trained to use them. This position is also taken by [23], who states that the common UML repository supports users to use *"notations they feel comfortable with and still interchange models with others who use different presentations for the same concepts."*. As the hybrid notation proposed in this paper combines the benefits of textual and visual notations and as it is based on standard UML actions its chances for adoption are quite good. Further, the provision of model compilers for code generation from UML actions [11,5] would favour even more that adoption.

As examples of languages that define both graphical and textual notations for action languages we mention Starr's Concise Relational Action Language (SCRALL) [24] and the Specification and Description Language (SDL) [25].

SDL was first developed using a Graphical Representation (SDL/GR) and was then extended with a textual Phrase Representation (SDL/PR). Both representations have the same underlying abstract syntax. While SDL preceded the development of UML actions it was certainly not designed as a UML action language. Recently, a mapping from SDL to UML actions was defined [25].

In contrast to SDL, Scrall [24] is fully compatible with UML 2.0 action semantics. The language is a hybrid text-graphical action language with a primary focus on the graphical language. However for simplicity the language supports textual expressions in SMALL or OAL e.g., for comparison, selection, computation and navigation, etc. that are integrated directly into a Scrall diagram.

The author promotes a pragmatic approach to *"Use text where compact linear expressions are most descriptive. Use graphics where the focus should be on the flow of data and control."* [24]. This combination of textual and graphical syntax lowers diagram complexity and space used and therefore addresses the thread of *gargantuan* models. This is also true for the hybrid approach we proposed for the VIDE language. In contrast to VIDE, SCRALL discards procedural artifacts, discouraging sequencing, the use of temporary variables and iteration. Instead, it focuses on explicit data and control flow to express parallelism more naturally. Consequently, SCRALL's graphical notation is a typical "boxes and lines" diagram notation for which it is difficult to provide a reasonable automatic layout – in contrast to our notation. It remains the burden of the user. Furthermore, typical diagrammatic notations without any restrictions tend to be more cluttered and less compact than text or our notation.

8 Conclusion

In this paper, we presented a hybrid notation and an editor for UML actions. With these contributions we tackle three problems hindering the adoption of UML actions for behavior modeling in practice namely the lack of a concrete syntax, the complexity of the meta-model, and the insufficient tool support. Our notation shields the user from several meta-model complexities and shows directly all action properties and attributes that must be set. The editor provides several features that ease creating/manipulating action models such as auto-layouting, auto-completion, and direct manipulation. The notation and the editor were evaluated using a user study that confirmed that learning the hybrid notation and its tool is faster than learning a new textual programming language. A comparison of the proposed notation with "pure" visual notations, e.g. those described in Section 7 remains to be done as future work.

Acknowledgements. This work is supported by the European Commission FP 6 Project VIDE - IST-033606-STP. We thank Andreas Roth for contributing to an early version of the proposed notation.

References

1. OMG: MDA home page (2009), http://www.omg.org/mda
2. Gruhn, V., Pieper, D., Röttgers, C.: MDA – Effektives Software-Engineering mit UML 2 und Eclipse. Springer, Berlin (2005)
3. OMG: Concrete syntax for a UML action language request for proposal (2008), http://www.omg.org/docs/ad/08-09-09.pdf
4. France, R.B., Ghosh, S., Dinh-Trong, T., Solberg, A.: Model-Driven Development Using UML 2. 0: Promises and Pitfalls. Computer 39, 59–66 (2006)
5. VIDE: Visualize All Model-Driven Developement: EU FP6 Project (2009), http://www.vide-ist.eu

6. Falda, G., Habela, P., Kaczmarski, K., Stencel, K., Subieta, K.: Platform-independent Programming of Data-intensive Applications using UML. In: Meyer, B., Nawrocki, J.R., Walter, B. (eds.) CEE-SET 2007. LNCS, vol. 5082, pp. 103–115. Springer, Heidelberg (2008)
7. Eclipse: Eclipse model development tools (2008), http://www.eclipse.org/modeling/mdt
8. Topcased (2008), http://www.topcased.org
9. No Magic: MagicDraw UML (2008), http://magicdraw.com
10. VIDE: Deliverable d.11.3: Workshop, competition, training course and verification (2009), http://www.vide-ist.eu/deliverables.html
11. Charfi, A., Müller, H., Roth, A., Spriestersbach, A.: From UML Actions to Java. In: Proc. of IDM 2009 (2009)
12. Falda, G., Habela, P., Kaczmarski, K., Stencel, K., Subieta, K.: Executable Platform Independent Models for Data Intensive Applications. In: Bubak, M., van Albada, G.D., Dongarra, J., Sloot, P.M.A. (eds.) ICCS 2008, Part III. LNCS, vol. 5103, pp. 301–310. Springer, Heidelberg (2008)
13. Schmidt, A.: A visual editor for behaviour modelling with UML actions. Master's thesis, Technische Universität Darmstadt (2009)
14. VIDE: Visual editor video (2009), http://www.vide-ist.eu/reflib/elearning/vide/b/b0006.html
15. Eclipse: Eclipse Graphical Modeling Framework (2008), http://www.eclipse.org/modeling/gmf
16. Eclipse: Eclipse Modeling Framework Project (2008), http://www.eclipse.org/modeling/emf
17. Carter, K.: UML ASL Reference Guide, ASL Language Level 2.5, Manual Revision D (2003), http://www.omg.org/docs/ad/03-03-12.pdf
18. Mellor, S.J., Balcer, M.: Executable UML: A Foundation for Model-Driven Architectures. Addison-Wesley Longman Publishing, Amsterdam (2002)
19. Heitz, C., Thiemann, P., Wölfle, T.: Integration of an Action Language Via UML Action Semantics. In: Draheim, D., Weber, G. (eds.) TEAA 2006. LNCS, vol. 4473, pp. 172–186. Springer, Heidelberg (2007)
20. Ambler, S.W.: A Roadmap for Agile MDA (2007), http://www.vide-ist.eu/deliverables.html
21. Fowler, M.: UML as Programming Language (2003), http://www.martinfowler.com/bliki/UmlAsProgrammingLanguage.html
22. Hailpern, B., Tarr, P.: Model-driven Development: the Good, the Bad, and the Ugly. IBM Syst. J. 45(3), 451–461 (2006)
23. Conrad, B.: UML without Pictures. IEEE Softw. 20(5), 33–35 (2003)
24. Starr, L.: SCRALL (Starr's Concise Relational Action Language) (2003), http://www.modelint.com/downloads/mint.scrall.tn.1.pdf
25. Bjorkander, M., Ober, I., Weigert, T.: SDL Mapping for the UML Action Semantics (2000), http://www.omg.org/docs/ad/00-08-01.pdf

Mapping Requirement Models to Mathematical Models in Control System Development

Dominik Schmitz[1], Ming Zhang[1], Thomas Rose[1], Matthias Jarke[1], Andreas Polzer[2], Jacob Palczynski[2], Stefan Kowalewski[2], and Michael Reke[3]

[1] Fraunhofer FIT, Schloss Birlinghoven, 53754 Sankt Augustin, Germany
{dominik.schmitz,ming.zhang,thomas.rose,matthias.jarke}@fit.fraunhofer.de
[2] RWTH Aachen University, Informatik 11, Ahornstr. 55, 52056 Aachen, Germany
{polzer,palczynski,kowalewski}@embedded.rwth-aachen.de
[3] VEMAC GmbH & Co. KG, Krantzstr. 7, 52070 Aachen, Germany
reke@vemac.de

Abstract. When developing control systems software, mathematically based modelling tools such as Matlab/Simulink are used for design, simulation, and implementation. Thus, a continuous model-based approach does not need to map requirements to, for example, UML class diagrams but to this mathematical representation. In this paper, we build on previous work that has applied the requirements formalism i^* to the development of control systems software and present a mapping from i^* models to Matlab/Simulink models. During a first manual transformation step, design alternatives are resolved. The second, automated step generates a Matlab/Simulink skeleton model from the i^* model. Finally, an interactive step allows incorporating existing hardware and platform components into the skeleton. As a running example, we consider the development of a parking assistant.

1 Introduction

The development of software for embedded devices is a challenge nowadays, especially within a car [3]. Nonetheless in regard to control functionality, model-based approaches are already quite common. But unlike usual software models, control system engineers put mathematical models of the controlled system and the controller at the centre of the modelling. Furthermore, models are used only during the later development phases. At the requirements level, simple text-based documents are still predominating [5].

The BMBF funded project ZAMOMO aims at a better integration of model-based software and model-based control engineering. As a result of this effort, we successfully applied the agent- and goal-oriented requirements engineering modelling framework i^* [12] to this field [10]. This enables for the first time an integrated and continuous, model-based approach. Besides various advantages on requirements level (see [11] for an elaboration), another key argument should be an easier step from the requirements phase to later development phases. To close this remaining open issue, we present within this paper how to derive an initial

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, pp. 253–264, 2009.

mathematical model in the format of the most commonly used Matlab/Simulink environment (see http://www.mathworks.com) from an i^* model. We provide suitable tool support and present our approach with a real world example concerning the development of a parking assistant for a small test vehicle at one partner's site.

The paper is organized as follows. Section 2 gives a brief introduction to control systems development. Afterwards Sect. 3 introduces our i^* based approach towards requirements modelling for control systems. Section 4 then presents the conceptual mapping from i^* to Matlab/Simulink, whereas the details of the implementation are given in Sect. 5. A discussion of related work (Sect. 6) as well as a summary and an outlook (Sect. 7) conclude the paper.

2 Control Systems Development

The task of a *controller* is to continuously compare and adapt the current value(s) of some system to some possibly changing desired value(s) [8]. For example, the driver of a car specifies a desired velocity implicitly and indirectly via the position of the accelerator. But only the engine controller within the electronic control unit computes the appropriate amount of fuel as well as the best point in time for injection and ignition. Control system functionality is nowadays implemented in software on electronic control units (ECUs) [3]. This is also the prerequisite for higher level support in form of driver assistance systems such as adaptive cruise control or the electronic stability program.

In industrial practice, a controller is developed in five steps [8]. Control engineers have to understand the controlled system, for example, an engine for which a controller is to be developed. Thus, the first step in the engineering process is to build up a *block diagram*, a simple graph-based representation of the interactions between the main components that captures the behaviour of the controlled system. This model helps identifying actuating and controlled process variables, i. e. the controlled system's properties in the second step. In the third phase, the controller's functionality is designed. The fourth step concerns simulating the whole control cycle in crucial situations. If it behaves as expected and demanded, the controller is finally implemented and put into operation in the last step. The process is usually supported by rapid control prototyping environments (RCP) [1]. Such a tool chain supports the engineer who can model and simulate the whole control cycle and validate the controller design at early development stages. The Matlab/Simulink environment by The Mathworks is most commonly used in this context. Simulink is a graphical programming language on top of the numerical math engine Matlab. It allows for the signal oriented (thus, block diagram based) visual capturing of complex systems and provides a large set of libraries with pre-defined mathematical blocks. The user either selects blocks from these libraries or provides his own implementation, customizes the blocks via parameters, and connects them by signals via their input and output ports. The internal representation of each block and hence the whole model as differential equations eventually enables simulations. We will see an example of such a mathematical model later on (see Sect. 4.2).

3 Requirements Engineering for Control Systems with *i**

*i** [12] is an agent- and goal-oriented, semi-formal modelling framework targeting the requirements engineering domain. In [10], the authors successfully explored the possibility to capture control system requirements with *i**. We will briefly reconsider this approach here and thereby give a short introduction to the *i** framework.

Strategic Dependency (SD) Diagram. In *i**, two levels of modelling are distinguished. On the higher level, the modeller can capture the various stakeholders (*agents*) and their dependencies within the so called *strategic dependency (SD) diagram*. Within controller development, the "controlled system" as well as the "controller" are the two main stakeholders. Furthermore, their combination, the "control cycle" needs to be considered a stakeholder of its own since some properties, e. g. stability, can only be assigned to this combination. In Fig. 1 the requirements regarding a parking assistant are captured. Besides the "parking assistant", the "car" as well as the "environment" are considered important stakeholders. The "parking assistant" *is part of* the "controller" whereas the other two stakeholders are part of the "controlled system". We have kept the model simple, but other stakeholders such as the driver could be added in a similar fashion.

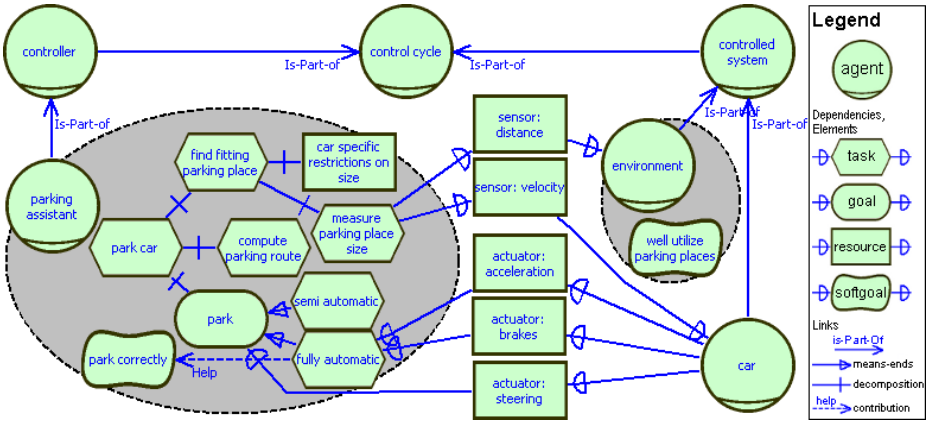


Fig. 1. Combined SD and SR Diagram for a Parking Assistant

Dependencies are used to capture the relationships between the modelled stakeholders. In a *task dependency* the depender specifies the details of the requested service while in a *goal dependency* it is left to the dependee how to provide a requested state. Similarly, with a *resource dependency* the dependee only has to provide the resource. A *softgoal dependency* is similar to a goal dependency but there are no clear-cut criteria when this goal is fulfilled. Instead a dependee can only make contributions. Thus, softgoals are suited to capture non-functional requirements (quality goals). In control systems engineering, all these

types of dependencies apply as well. Especially, *actuators* and *sensors* are simply captured by corresponding resource dependencies. Figure 1 shows that the parking assistant depends on the environment for "distance" information and on the car for "velocity". In turn, the car receives input from the parking assistant in regard to "steering" and potentially also "acceleration" and "brakes".

Strategic Rationale (SR) Diagram. The individual goals and processes of stakeholders and systems as well as their relation to external dependencies are captured in the more detailed *strategic rationale (SR) diagrams*. Regarding the modelling, the types of links (task, goal, resource, and softgoal dependencies) simply become modelling elements on this level. Additionally, the SR diagram provides new types of links to detail out a complex task (*decomposition*), to model alternatives to achieve goals (*means-ends*), or to specify qualitative contributions towards softgoals (*help, make, break, etc. contribution*). Figure 1 also shows SR details of some stakeholders involved. For example, the main task "park car" can be subdivided into three tasks "finding a fitting parking place", "computing the parking route", and eventually the actual "parking". The latter is modelled as a goal to consider two alternatives: a "fully automatic" approach (that thus also controls acceleration and brakes) or a "semi automatic" approach that only affects the "steering" while the driver has to accelerate. Overall, parking assistants are intended to support the softgoal "well utilize available parking places" in that most people wrongly decide that they do not fit into a parking place.

In contrast to a simple block diagram [8], the *i** model as introduced above is thus able to capture not only the functional interdependencies of controller and controlled system but also of software and non-functional issues as well as various additional stakeholders within a combined view. Thereby, *i** allows to explore reusability and scalability of controllers in different contexts systematically. Further enhancements have been introduced by accommodating the fact that control systems engineers are usually experts in some particular sub domain, such as, for example, engine controllers. We expect this knowledge to be captured in form of a domain model to support the creation of requirements models [11].

4 Mapping *i** to Matlab/Simulink

We have divided the transformation from *i** to mathematical models, more precisely Matlab/Simulink models, into three steps. The first step requires manual intervention by a human developer to resolve all ambiguities (i. e. goal alternatives) in the requirements model by taking design decisions. Secondly, the core mapping takes place by automatically generating a Matlab/Simulink structure for the remaining solution. The third step is particular to the domain of control systems development. The developer is supported by taking the intended hardware and platform details much earlier into account as in normal software development, i. e. already when designing and not only when implementing. In the following we elaborate each step, whereas implementation details are given in Sect. 5.

4.1 Step 1: Resolving Ambiguities

The intention of an *i** requirements model is to capture the problem characteristics and investigate various kinds of solutions. Consequently, the model is necessarily ambiguous. For example, in our running example (see Fig. 1) for "parking" both alternatives, a "fully automated" as well as a "semi-automated parking assistant" are considered. Before proceeding with the further development, the developer has to take his design decisions, i. e. he has to decide for each goal which alternative to select. Of course, the modelled softgoals and corresponding contribution links help during this decision process as well as existing tool support such as label propagation or others (see [6]). But nonetheless in the end it is a decision of the developer and that is why we consider it a manual transformation step. We expect the developer to document his decisions in the model by according markings. Regarding our example, there is only one goal "park" (inside the "parking assistant"). Assuming the choice (and hence marking) of a "fully automated parking assistant", we can easily derive a model that contains only the relevant alternatives, i. e. the task "semi automatic" is simply discarded based on the marking. The resulting model serves as the starting point for the next development step.

4.2 Step 2: Core Mapping

Table 1 shows how the concepts of *i** are mapped to the concepts of Matlab/Simulink. Mostly, a block, in Matlab called a *subsystem*, is the target for model transformation. This is especially true for all *agent* and *task* elements, whereas *softgoal* elements and dependencies are mapped to textual hints to remind the developer during further development of important non-functional aspects. *Goals* have more varying transformation targets. As dependencies they

Table 1. Mapping i* Concepts to Matlab/Simulink Concepts

i* **Model**		**Matlab/Simulink model**
stakeholders	agent	subsystem
dependencies	task dependency	subsystem in dependee, signals
	goal dependency	text
	softgoal dependency	text
	resource dependency	
	- sensor or actuator	subsystem and signals
	- normal	signal only
rationale elements	task	subsystem
	goal – no refinement	model verification
	– refined	subsystem
	softgoal	text
	resource	constant
links	decomposition, means-ends, is-part-of	nesting of subsystems
	contribution	– (omit)

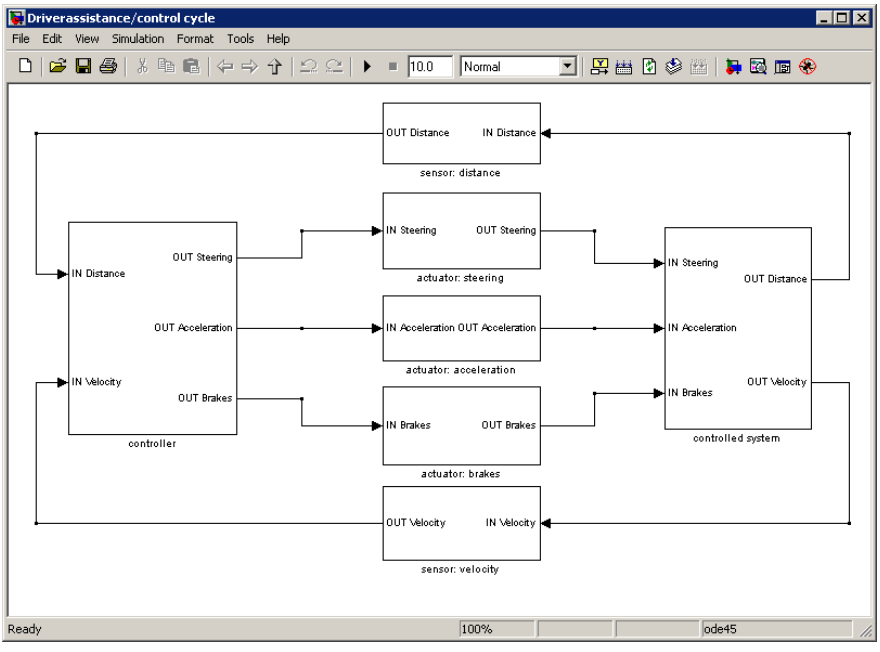


Fig. 2. "Control Cycle" Subsystem of Resulting Matlab/Simulink Model

are also mapped to *text*. As elements of an SR diagram they are mapped to a subsystem if they are refined, i. e. an incoming means-ends link exists. If they are not refined, they are assumed to describe hard goals that need to be achieved and therefore are mapped to the *model verification* element. *Resource dependencies* are very important since they are used in our approach to represent sensors and actuators. In this special usage, they are also transformed in a special way since then an additional *subsystem* is generated representing the sensor/actuator in Matlab/Simulink. Otherwise only a *signal connection* is created from the dependency.[1] On SR level, resources are mapped to blocks representing constants, since this is their most likely application. *Decomposition*, *means-ends*, and *is-part-of* links are mapped to parent-child relationships of *subsystems*, i. e. their nesting. And finally, *contribution* links can be omitted since the design decisions have already been taken.

Figure 2 shows the content of the "control cycle" subsystem as a result of the mapping. Next to the subsystems for "controlled system" and "controller", you also see the subsystems that result from sensor and actuator relationships (e. g. "distance", "steering", etc.) whereas the subactors "parking assistant", "environment", and "car" as well as any SR detail from Fig. 1 are not visible in this view since they are internal parts of subsystems.

[1] Notice: Due to its semantics, the directions of the dependency and of the resulting signal are opposed. If the controller depends on a sensor for some information, then we expect an incoming link to the controller on Matlab/Simulink level.

4.3 Step 3: Respecting Hardware and Platform Issues

The mapping described above quite generically maps all functional aspects to subsystems. But the chief attraction of using Matlab/Simulink is the vast set of pre-defined blocks that perform some particular computation, for example, a pre-defined, so called PID controller with proportional, integrative, and differencing components that only need to be parameterized. Thus, the developer will replace or detail out the generic subsystems during the further development by more specific Matlab/Simulink components. Accordingly, the generated model is only a first initial model providing a suitable structuring. Since in control system development specifics of the underlying hardware and platform need to be considered much earlier than in usual software development, pieces of information in this regard are possibly already part of the requirements model. Thus, this specific knowledge should also be considered during the transformation.

In the following, we will focus the specifics of the VeRa rapid control prototyping platform that has been developed by one of the project partners to provide an open but nonetheless close-to-production control system development environment (see Fig. 3 (a)). The VeRa system comes equipped with a tool chain that provides specific modelling elements to the mathematical modelling environment via a library, allows to use the Realtime Workshop (provided by The Mathworks) to generate code for the VeRa platform, and finally provides means to bring the generated code on the real hardware (flash).

Figure 3 (b) shows the content of a VeRa library related to the domain of driver assistance systems. It provides sensors/actuators helpful for realizing, for example, a parking assistant, adaptive cruise control, or the electronic stability program (ESP). As an example, regarding "distance" an ultrasonic or an infrared sensor are available. We have extended this library by meta tags that categorize each element into the basic categories "sensor", "actuator", "controller", and "controlled system". The modeller can specify which library(ies) to consider during transformation. By interpreting the meta information, we can easily offer potential matches for the mapping of elements. Currently, only the categories are matched, thus for each sensor in our requirements model the developer is offered all sensors from the libraries. In the future, we expect that correspondences between the domain model (see [11]) that is used to build up the requirements model and the assisting VeRa library will exist that can be exploited. Altogether, a much more tailored and specific model results from the model transformation.

(a) 

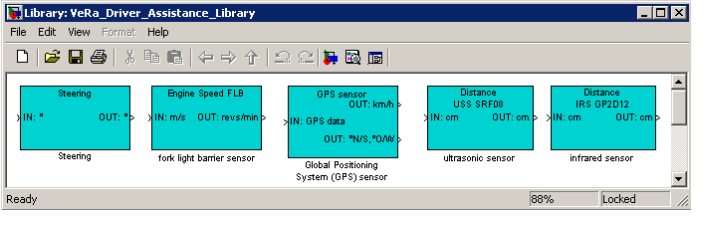
(b) 

Fig. 3. (a) VeRa RCP System (b) Driver Assistance Domain-Specific Matlab-Library

5 Implementation

Most importantly, the knowledge representation language Telos [9], that is underlying the i^* formalism, and its implementation ConceptBase [7] are used for realizing the core transformation step. Eventually, SimEx, a specific tool to interact with Matlab/Simulink kindly provided by IT Power Consultants (see http://www.itpower.de), finalizes the transformation procedure. Due to reasons of space, we focus the first two transformation steps here.

5.1 Telos and ConceptBase

The knowledge representation language Telos [9] has been introduced to capture knowledge about information systems and to support their development. Originating from former requirements modelling languages, the focus for the development was on achieving a high-level of adaptability. Thus, meta-modelling means are at the core. To achieve this, Telos offers a simple generic data model consisting of nodes (objects), links (attributes, relationships), and logical assertions that is extensible to specific application needs. The user can employ assertions as deductive rules to derive additional information automatically, or as integrity constraints to control the entry of information by users. Finally, we need at least one abstraction mechanism – classification – which enables us to talk about classes and their instances. A complete definition can be found in [7].

ConceptBase is a deductive object base management system for meta databases implementing the Telos data model (for detailed information please visit http://www-i5.informatik.rwth-aachen.de/CBdoc/). The system has been used in various projects ranging from development support for data-intensive applications, requirements engineering to co-authoring of technical documents. ConceptBase's implementation of O-Telos provides a couple of features beyond the core Telos. For one, there is a powerful query language. For another, modules have been introduced to structure the name space. Finally, ConceptBase supports a limited version of active rules to react to internal and external events.

5.2 Queries Support Step 1: Resolving Ambiguities

With the help of Telos we can easily provide checks on the readiness of a model for transformation by using the *query class* concept. A query is represented as a class where the instances form the answers to the query. The query itself is stated in form of a constraint of the class such that all objects of the object base fulfilling this constraint become (virtual) instances. Validation rules can be formulated by queries that explicitly search for violations. For example, Figure 4 shows a query that checks whether each goal has at most one (chosen) alternative by simply asking for a goal that has more than one. If the result is empty, the model is ready for transformation in this regard. Consequently, a set of such queries can be used to fully check the model.

```
QueryClass checkGoals isA IStarGoalElement with
  retrieved_attribute name : String
  constraint rule : $exists ml1,ml2/IStarMeansEndsLink
                    (ml1 <> ml2) and (ml1 to this) and (ml2 to this)$
end
```

Fig. 4. Query that Checks for Single Alternative at Goals

5.3 Metamodelling, Queries, and Answer Formats Support Step 2

It has turned out that the transformation is simplified when we define an intermediate Matlab/Simulink-like meta model in Telos. Since this is exactly the intended purpose behind Telos, the definition is straight forward. The available elements are mainly *blocks* but specialized into *subsystems*, *ports* (in and out), and some more. Regarding links, a single type *line* is sufficient. An attribute to *subsystems* finally captures the typical nesting of subsystems in Matlab/Simulink models. The complete meta model is given in [13]. The introduction of the Matlab/Simulink-like meta model allows for a refinement of the core transformation step (Step 2). The *i** model is first transformed into the new, also Telos-based modelling language (stored again in ConceptBase but in a separate module containing the Matlab/Simulink meta model instead of the *i** meta model). From here, we export the models content to an XML file that can be read by the SimEx tool to finally generate the information in the original Matlab/Simulink file format (.mdl).

Both sub transformation steps are realized by a set of queries in combination with suitable answer formats (see below). The queries for the first transformation result straight away from the mapping in Table 1 and are applied in a suitable order to the requirements model. We start with the agent element, then resources, tasks etc. In some cases special measures have to be taken. For example, we use the active rule feature to create a suitable ascending numbering of ports within a subsystem. Deductive rules are helpful to simplify queries by hiding complexity. As obvious from the transformation (see Table 1) the nesting of subsystems on Matlab/Simulink side results from various kinds of modelling (is-part-of, decomposition, parent-child relationship) in *i**. A derived attribute *parent subsystem* eases access to the particular parent subsystem. Similarly, we have introduced deductive rules to automatically derive whether a resource dependency is a sensor, an actuator, or none of this. Due to reasons of space, we have to omit these technical details. For more information, see [13].

User-defined answer formats allow us to access the result of a query and reformat it to our needs. While the original intent is only the second kind of application step (e. g. generating an XML representation of the Matlab/Simulink like model), we also use this feature for the first transformation, i. e. to derive the Telos frames for the Matlab/Simulink like language. Figure 5 shows as an example the two representations of the actuator "steering". In *i** (left side), this is captured by a resource element (`i*Elem_20`) and according links (`i*Link_19`, `i*Link_20`). In Matlab/Simulink (right side), we get a subsystem (`MSElem_20`) with children that represent ports (`MSElem_2041`, `MSElem_204`, ...). Furthermore,

the element is registered as a child of the "control cycle" (MSElem_56). Finally, several lines between the ports that are generated on various subsystem levels are created (MSLine_246, ...). Notice that varying numbers of frames are generated reflecting the different ways the two meta models represent information. In much the same way, the newly created Telos model in the Matlab/Simulink style can again be queried with user-defined answer formats to derive an XML representation that is readable by the SimEx tool.

*i**

```
Token i*Elem_20
    in IStarResourceElement with
  name
    Label0 : "actuator: steering"
  links
    Label1 : i*Link_19;
    Label2 : i*Link_20
end

Token i*Link_19
    in IStarDependencyLink with
  from  from : i*Elem_4  % "car"
  to    to   : i*Elem_20
end

Token i*Link_20
    in IStarDependencyLink with
  from from : i*Elem_20
  to   to   : i*Elem_9   % "park"
end
```

Matlab/Simulink

```
Token MSElem_20 in Subsystem with
  name
    Label0 : "actuator: steering"
  children
    Label1 : MSElem_2041;
  ... % omissions, e.g. port counters
end

Token MSElem_56 with % "control cycle"
  children  Label20 : MSElem_20
end

Token MSElem_2041 in Outport with
  parent  Label0: MSElem_20
end

Token MSElem_204 in Inport with
  parent  Label0 : MSElem_4 % "car"
end

Token MSLine_246 in Line with
  from  Label0 : MSElem_2041
  to    Label1 : MSElem_204
end
... % various details omitted
```

Fig. 5. Original *i** Frame of a Resource Dependency vs. Simulink Like Representation

6 Related Work

Model transformation is an important part of MDA, the Model Driven Architecture promoted by the OMG (http://www.omg.org/mda). In their nomenclature, we have presented here a model-to-model transformation. The design decisions can be considered as "additional information" respected during the transformation step. Furthermore, taking hardware specifics into account as we have presented here, results in a so called "platform-specific model". But we have not applied any of the existing model transformation approaches, such as, for example, the ATLAS transformation language (see http://www.eclipse.org/m2m/atl/). This is mainly due to the fact that these approaches favour the use of UML,

MOF, or EMF, respectively, as the basic modelling language. Thus, this would have resulted into doubling conversion efforts. Furthermore, we took advantage of ConceptBase's powerful features such as queries, deductive and active rules relying on a well founded knowledge representation approach. Thus, the main "intelligence" of our transformation is captured in the powerful rules and queries.

We also tried to use the XML representation of *i**, called iStarML [4], in combination with XSLT. But as others have already indicated [2], for complex models and relationships as they occur in our models, XML-based approaches are not suitable. And finally, the TAOM4E tool environment (http://sra.itc.it/tools/taom4e/) supports a fully model-driven approach towards software development based on *i**. But since we address control systems development, a mapping towards mathematical modelling languages is missing there.

7 Conclusions

Within this work, we have presented a model transformation that addresses the peculiarities of control systems development. It allows to map requirements captured in an *i** model to an initial mathematical model that is commonly used in control systems development (Matlab/Simulink). The transformation includes manual (taking design decisions) as well as automatic (core mapping) and semi-automatic (refine mapping towards hardware/platform) steps. At the core, we use the knowledge representation language Telos and its implementation ConceptBase to perform the essential transformation step.

Two case studies were investigated during the development of this approach, the parking assistant that has been presented here and an engine controller that is much more complex. Discussions with the project partners have already shown some advantages of our approach. As with any continuous model-based approach, the key advantage is consistency. For example, the Matlab/Simulink models that we derived from our requirements models sometimes deviated in details from the original Matlab/Simulink models that have been developed by our partners. In some cases, they simply did not treat all sensors or actuators in the same way. Our approach allows avoiding such pitfalls thereby easing maintenance in the long run. Furthermore, due to the automatic generation of models, a better structuring was enforced. By reflecting the fine-grained decompositions in *i** this prepares for future changes by a better modularization of the Matlab/Simulink model. On the other hand, the current implementation must still be considered a prototype. Overall transformation time is in the range of minutes (depending on the computer's power and main memory). This needs to be improved for every day real-life application.

Thus, regarding future work several improvements can be made. For one, we can improve the hardware/platform specific mapping by including a better similarity search (text and/or structure based) and also the domain model information on requirements level (see [11]). Furthermore, an abstraction layer has been developed in the project that aims at reducing the hardware dependency of the Matlab/Simulink model. We then only need to map to a "distance" sensor block, while the choice between ultrasonic and infrared is expected in

an XML-based configuration layer that hides the hardware-specific characteristics. And finally, our approach to model transformation via ConceptBase and its user-defined answer formats needs a closer investigation. Although it has mainly been driven here by pragmatic issues (neither model was given in EMF), the approach founded on a knowledge representation language seems to work well and potentially could be of advantage also in other fields, especially, since the core "intelligence" of the mapping is defined declaratively as rules and queries.

Acknowledgment. This work was supported by the BMBF in the project ZAMOMO (01 IS E04).

References

1. Abel, D., Bollig, A.: Rapid Control Prototyping. Springer, Heidelberg (2006)
2. Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., Weiss, E.: Graphical definition of in-place transformations in the Eclipse Modeling Framework. In: Nierstrasz, O., Whittle, J., Harel, D., Reggio, G. (eds.) MoDELS 2006. LNCS, vol. 4199, pp. 425–439. Springer, Heidelberg (2006)
3. Broy, M.: Challenges in automotive software engineering. In: Int. Conf. on Software Engineering, Shanghai, China, pp. 33–42. ACM, New York (2006)
4. Cares, C., Franch, X., Perini, A., Susi, A.: iStarML: An XML-based model interchange format for i*. In: Proc. of the 3rd Int. i* Workshop, Recife, Brazil, February 11-12. CEUR Workshop Proceedings, vol. 322, pp. 13–16 (2008)
5. Graaf, B., Lormans, M., Toetenel, H.: Embedded software engineering: The state of the practice. IEEE Software 20(6), 61–69 (2003)
6. Horkoff, J., Yu, E.S.K.: Qualitative, interactive, backward analysis of i* models. In: Proc. of the 3rd Int. i* Workshop, Recife, Brazil, February 11-12. CEUR Workshop Proceedings, vol. 322, pp. 43–46 (2008), CEUR-WS.org
7. Jarke, M., Gallersdörfer, R., Jeusfeld, M.A., Staudt, M.: ConceptBase - a deductive object base for meta data management. Journal of Intelligent Information Systems 4(2), 167–192 (1995)
8. Lunze, J.: Automatisierungstechnik. Oldenbourg (2003)
9. Mylopoulos, J., Borgida, A., Jarke, M., Koubarakis, M.: Telos - representing knowledge about information systems. ACM Trans. on Information Systems 8(4), 325–362 (1990)
10. Schmitz, D., Drews, P., Hesseler, F., Jarke, M., Kowalewski, S., Palczynski, J., Polzer, A., Reke, M., Rose, T.: Modellbasierte Anforderungserfassung für softwarebasierte Regelungen. In: Software Engineering, LNI, Munich, Germany (2008)
11. Schmitz, D., Nissen, H.W., Jarke, M., Rose, T., Drews, P., Hesseler, F.J., Reke, M.: Requirements engineering for control systems development in small and medium-sized enterprises. In: 16th Int. Requirements Engineering Conf., Barcelona, Spain. IEEE, Los Alamitos (2008)
12. Yu, E.: Modelling Strategic Relationships for Process Reengineering. Ph.D thesis, University of Toronto (1995)
13. Zhang, M.: Conception and implementation of the transformation of i* requirements models to mathematical design models in Matlab (in German). Master's thesis, RWTH Aachen University (to appear)

On Study Results: Round Trip Engineering of Space Systems

Andrey Sadovykh[1], Lionel Vigier[1], Eduardo Gomez[2], Andreas Hoffmann[3], Juergen Grossmann[3], and Oleg Estekhin[4]

[1] SOFTEAM, Paris, France
{andrey.sadovykh,lionel.vigier}@softeam.fr
[2] ESA ESOC, Darmstadt, Germany
Eduardo.gomez@esa.int
[3] Fraunhofer FOKUS, Berlin, Germany
{juergen.grossmann,andreas.hoffmann}@fokus.fraunhofer.de
[4] GTI6, Massy, France
oleg.estekhin@gti6.com

Abstract. Software developed for the space domain often has to deal with extremely long mission times (sometimes in the order of 15 to 20 years). During the lifetime of a mission programming platforms evolve and sometimes disappear forcing migrations or updates. Migration can also be triggered by the appearance of new platforms that can improve scalability, performance. European Space Agency (ESA) is interested in modernization approaches that simplify platform migration and that preserve the business values of systems. The Architecture Driven Modernization (ADM) promoted by the Object Management Group (OMG) proposes to recover the models which represent the business value and proceed with the platform migration in a forward MDA process. This article provides results of a study dedicated to assess the state-of-the-art tools and methods for model driven platform migration, including model-based testing and metrication.

Keywords: MDA, ADM, M2M, round trip engineering, PSM2PIM, U2TP, TTCN-3, model mining, model-based testing, model metrication.

1 Introduction

Some year ago, the Object Management Group (OMG) introduced the Model-Driven Architecture (MDA) initiative as an approach to system-specification and interoperability based on the use of formal models. In MDA, platform-independent models (PIMs) are initially expressed in a platform-agnostic modelling language, such as UML. The PIM is subsequently translated to a platform-specific model (PSM) by mapping the PIM to the target platform (e.g. CORBA and Java) using formal rules. Then, the code is generated from that PSM.

At the core of the MDA concept are a number of important OMG standards: The Unified Modelling Language (UML), Meta Object Facility (MOF), XML Metadata Interchange (XMI), and the Common Warehouse Metamodel (CWM). These standards

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, pp. 265–276, 2009.

define the core infrastructure of the MDA, and have greatly contributed to the current state-of-the-art of systems modelling.

As an OMG process, the MDA represents a major evolutionary step in the interoperability standards. For a very long time, interoperability was based largely on CORBA standards and services. Heterogeneous software systems interoperate at the level of standard component interfaces. The MDA process, on the other hand, places formal system models at the core of the interoperability problem. The most significant benefit about this approach is independence of the system specification from the underlying implementation technology or platform. The system definition exists independently of any implementation model and has formal mappings to many possible platform infrastructures (e.g., Java, C++, CORBA, Web Services).

Technical and economical advantages being promised by MDA are of a special concern for the space domain where mission life-cycle of large system often reaches 15-20 years (e.g., ISS, ATV, Columbus and many other programmes). Typically, COTS hardware and software platforms evolve tremendously throughout this lifecycle. Therefore, implementing effective and efficient modifications of legacy application software and porting it to new platforms would deliver significant benefits to space programmes in terms of cost reduction, increase of performance and flexibility.

The round trip engineering is usually referred to as a process of model generation from source code (reverse engineering) followed by generation of source code from models. In this way, existing source code is converted into a model, subjected to software engineering methods and then converted back into code. Usually, this process is fulfilled on a platform specific level, while in the current project the round trip engineering was considered on a larger scale: a model is abstracted to a platform independent level and then a new platform specific model is derived. The goal of this larger process is to extract a PIM containing the most persistent part of the legacy system and then to proceed to the deployment to a new platform through all the MDA phases. A special emphasis is put to the need of model-based verification and validation using generated tests.

The major goal of the study was (1) to develop a formal methodology for the above-mentioned process, (2) to survey the current state of the art in applicable of-the-shelf technologies and tools and (3) to demonstrate a feasibility of the approach.

2 Case Study

In order to demonstrate the applicability of the previously-mentioned methodology to ground systems, ESA provided an application or a “target system”, which is representative and simple enough to be able to concentrate on the methodology. This target system is a distributed file archive (FARC) enabled with a version control system.

After measuring using quality and maintainability metrics developed for this purpose, the platform independent model (PIM) had to be extracted, allowing capitalising of the existing know-how. In addition, PIM on conceptual level had to be used for architecture re-structuring. Then, the code had to be generated for Java platform in forward MDA process. All over the development process an early validation was required to ensure the model consistency, while on the last stage the newly re-factored system had to be validated using generated tests. Finally, the metrics analysis process

had to be repeated, in order to evaluate quality and maintainability metrics before and after modernisation process.

The following important concerns were identified for the case study. A list of COTS frequently used within the MDA / ADM engineering and relevant for space systems had to be identified and assessed. For some phases where there was a lack of tool support (PIM extraction, dedicated transformations), special actions were taken to review on-going research activities, prototype and evaluate them.

Furthermore, specific emphasis was put on early, model-based verification and validation. In particular, the expertise in TTCN-3 and U2TP was largely required.

The methodology and demonstrator tools chain had to rely as far as possible on standards (OMG's UML2, MOF, U2TP, XMI, KDM, and SPEM) and mature products to avoid proprietary solutions and ensure the capability to evolve with time.

3 Methodology

According to guidelines described above, an MDA / ADM process for the given case study was developed (c.f. Figure 1).

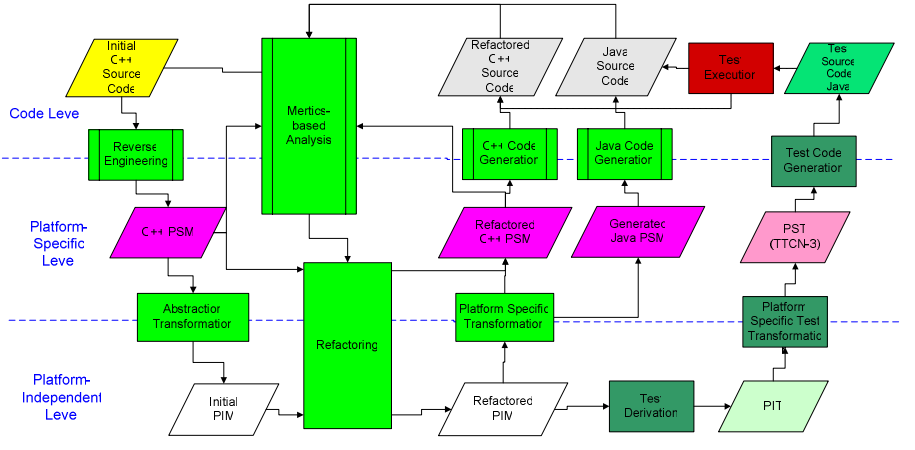


Fig. 1. RTE Space Methodology Concept – Flow Chart

Initial source code is subjected to a metrics based analysis in order to calculate reusability and maintainability metrics and to define goals for further refactoring. Then the code is reversed and a Platform Specific Model (PSM) is derived. This PSM is also subjected to the metrics based analysis.

The PSM is subjected to an abstraction transformation that aims at eliminating platform dependencies and extracting core business logic. Thus, a Platform Independent Model (PIM) is derived.

Taking into account the refactoring objectives resulted from the analysis process, the PIM is restructured in order to fulfil those goals. From this point, the verification

and validation process is started. The resulting PIM is used for the creation of a Platform Independent Test Model (PIT) which is subsequently mapped to a Platform-Specific Test Model (PST) as starting point for test code generation.

After the verification, a new PSM was generated for a Java platform. This PSM is subjected to a new round of verification. In addition the PSM is used to generate tests for further validation – Platform Specific Test Model (PST) is created. From this point, the analysis metrics is recalculated for early evaluation of the successfulness of the re-factoring.

In the next step, the code for a given platform is generated. It is validated using the generated tests. Finally, the code is again analysed, in order to determine the added value of the whole process.

In this way, the methodology covers all the abstraction levels: Code, Platform Specific and Platform Independent levels.

With regards to this methodology the project followed the phases listed below:

- PSM and PIM Extraction;
- PSM Derivation for the new target platform;
- Code Generation for the new target platform;
- Test Generation and Execution.

The metrics analysis mentioned before was fulfilled during the PIM Extraction and after the Code Generation.

4 Tools

According to the requirements and methodology definition the candidate tools were analysed and justification of the choice was provided. For the PIM Extraction and PSM Derivation the UML2 CASE Tool had to be chosen. The criteria for tools assessment provided below, we believe, may be characteristic for the platform migration problems.

First of all the UML2 Tool had to support both original and target platforms and in some cases even simultaneously. In our case the round-trip reverse/generation feature was necessary for C++ and Java. Due to the lack of the solutions for PIM Extraction, the tool had to support UML2 Profile development, transformations and user interface customisation. In addition, for the Code Generation/Migration phase, we needed an integrated configuration system for model and code management in a multi-user environment. Finally, EMF UML2 XMI export support was required for integration with testing workbench.

According to these parameters the choice of Objecteering UML2 CASE Tool[1] was justified. This tool was used for PIM Extraction, PSM Derivation and Code Generation. Specific functionalities were developed to help in this study: PIM extraction transformation, Find&Replace for Platform Replacement, Documentation generation and etc. These methods are described in the following section. In addition, the requirements for the FARC system were integrated with the model including traceability establishing and documentation generation.

[1] http://www.objecteering.com/

For the model-based testing the analysis identified the need for support of TTCN-3 and UML2 Test Profile (U2TP), the choice of the TT workbench[2] was justified during tool assessment. For the given study, special adapters for CORBA and XML communication were developed as well as specific transformation for UML2 sequence charts for U2TP.

For the metrication, Objecteering CASE Tool was used for model metrication, while CCCC[3] and Eclipse Java Metrics[4] were used for C++ and Java code metrication respectively. For reports integration and calculation of derived metrics we had to develop a specific tool.

5 Results

In this section we are presenting methods we applied for the case study fulfillment. In addition, we are briefly evaluating the results.

5.1 PIM Recovery and PSM Derivation

The particular interest for the methodology was recovering from the legacy artifacts the functional architecture also referred to as Platform Independent Model (PIM) in MDA terminology. Indeed, the Architecture Driven Modernization (ADM) approach proposes to base the modernization activities on the architectural models rather than code artifacts. Currently the reverse engineering methods and tools (SOFTEAM Objecteering, IBM RSA/RSM, BORLAND Together) allow to obtain a UML model, which we call Platform-Specific Model (PSM) since it is very close to the code. In this context the PSM-to-PIM method has to provide an abstraction transformation from the model obtained from the code into a model representing the functional aspect of the architecture.

PSM to PIM is often quoted as something possible, however hard to achieve because it is human knowledge related [1], [2], [3]. Depending on the context and understanding of what the platform is or what the business logic is, a component can be either categorized as part of a PIM or not.

For this project we used Objecteering CASE Tool and developed an instrumented approach [4] for PSM-to-PIM transformation involving human experts. Extracting a PIM from a PSM basically meant deciding for each model element whether it is part of the PIM or not. According to the expert decision the PSM elements were marked as Platform or Functional. The Functional part was then separated from the PSM model in order to build the PIM model. In order to simplify the decision process we introduced an algorithm based on dependency analysis.

First, packages representing the C++ COTS software libraries and all included elements were straightly marked as Platform. Secondly, all elements packaged within the main application folder were marked as Functional. The visualization mechanism developed highlighted classes that had to be manually revisited since contained elements (attributed, operations, parameters) dependent on the Platform classes. For theses classes experts decided case by case which part of the class is Platform and

[2] http://www.testingtech.com/
[3] http://sourceforge.net/projects/cccc
[4] http://metrics.sourceforge.net/

which is Functional. Finally the extraction process allowed obtaining Platform and Functional models. The Functional model constituted the PIM. The Platform model was used to find replacement COTS components for the Java platform.

During the Java PSM derivation, in addition to forward MDA PIM-to-PSM transformation process, we introduced several mechanisms for reusing the information from the original legacy system. Based on the Platform dependencies information from the previous phase, a class substitution mechanism was applied. The classes from newly integrated Java COTS replaced the references to their counterparts in the original C++ COTS where it was possible. In particular, this approach worked well for the XML parsing library Xerces which has a very similar API for C++ and Java. In addition, the stubs and skeletons generated for CORBA, where also straightly replaced. However, in many cases (API for threads, file system and etc.) the correspondences were not identified due to huge differences in the platforms. Some candidate solutions were outlined by introducing artificial dependency links to most convenient replacements in order to inform code developers. In addition, for all operations the original code for operations was put into comments for an easy access by developers.

After creation of the Java PSM, the Java code skeletons were generated. The skeletons represented the structure of the application. In addition for each operation a commented code was inserted showing the original business logic. From this stage the code migration phase started.

During the modelling phase we encountered that the results of the reverse engineering from C++ are often unsatisfactory since in the C++ development paradigm the usage of namespaces is not mandatory. The model obtained from source code may represent a flat structure with all classes in a single list. The classes may be duplicated and some ghost classes may appear since the reverse engines can not undoubtedly identify classes. As the result we encourage usage of namespaces for development in C++ for future development.

The experience gathered during the PSM marking using the dependency analysis makes us believe that more automation may be developed for this activity in the future in order to automate the functional PIM extraction.

5.2 Code Migration

The main migration activity was to check the generated method and class stubs for consistency and fill the method bodies with the corresponding logic.

The conversion from the C++ code to the Java code could be roughly divided in three categories:

- Changes required due to the difference in the syntax of programming languages;
- Changes required due to the difference in the idioms, internal working and best practices adopted in each language;
- Changes required due to the difference in the system dependent libraries and COTS (commercial off-the-shelf components) available in each language.

The C++ and Java have a lot of common syntax. Arithmetic expressions, cycles and conditional operators can be copied with minimal or no modifications at all and a lot of more complex C++ code can be converted to Java by simply removing or replacing some keywords and removing the C++ pointer dereference operators.

On the other hand there are syntax constructions that look similar but have difference semantics in C++ and Java, and there syntax constructions that are not present in Java at all.

An example of the former case C++ templates and Java generics share similar syntax but have very different semantics. In particular, it is not possible in Java to create an instance of the type used as a type parameter. An example of the latter case is the ability to pass a primitive value by reference as a method parameter in the C++. Such parameters are called in-out parameters and changes made to their value inside the method body will be visible to the method caller.

The most prominent difference that is beyond the simple syntax is the C++ and Java memory models. The common practice in C++ is either to use smart pointers that automatically call object destructor or to call the object destructor directly. Both practices ensure that the object is destroyed at some predictable time. Java has automatic garbage collector that provides no guaranties about object destruction time. Instead of using Java finalizers all non-trivial C++ destructors should be replaced with some special "dispose" method. The life-cycle of each class instance should be carefully examined and the dispose method should be called in the appropriate place.

While both C++ and Java support exceptions and their usage is almost mandatory in Java, a lot of C++ code still uses the error reporting idiom inherited from the C programming language, where the method returns some special value to indicate a failure. It is very difficult to write a code that checks all return values and handles or propagates failures correctly. When possible, all error handling should be converted to use exceptions instead of error codes as exceptions provide error propagation feature automatically and allow more clean way of handling and recovering from failures.

Independently of the language pair used in the migration process all system- and COTS-dependent code will have to be examined and in most cases replaced with completely new code. Data collections such as lists, sets and maps, multi-threading and file system access are usually provided by the standard language libraries. Even if it is not possible to copy and modify the original code due to the big differences between libraries provided by different languages their functionality is in general well understood and the counterpart from the other language is obvious.

Finding a replacement for a non-standard library can be much more difficult. While C++ and Java usually have either or both open-source and commercial libraries for each task that can be imagined, these libraries usually have very different API. Fortunately, in the case of this project the only external dependencies were the XML processing facility and CORBA. The XML processing was performed using the Apache Xerces which supports both C++ and Java. While Xerces for C++ and Xerces for Java API's are different, at least they are provided by the same vendor and have a common feeling to them. The CORBA has standardized language mappings for C++ and Java which are very similar, and in fact most of the CORBA-related code was directly copied from C++ to Java with language syntax corrections only.

Table 1 presents the code migration statistics. The key result is that more than 70% of cases were rather simple to migrate with the predefined rules. For these cases, one might consider usage of automatic translation from C++ to Java. For the rest about 30% cases manual code re-writing was possible only and it is hard to imagine that an automated translation may output a meaningful code. For future research these results should be carefully studied to be generalised for particular class of problems.

Table 1. Code migration statistics

	Simple Conversion	Complex Conversion	Total Rewrite	Manually Added
Classes	232	-	17	21
Fields	557	-	129	219
Methods	1959	116	48	394

5.3 Model-Based Testing

The basic idea of deriving test models from system models is to reuse the information about the system under (re)development also for developing the test models as the counterpart to the system.

The tests developed in this project for the FARC application aim to safeguard the modernization process. The modernized FARC application should behave in the same way the original application did. Hence, the tests are dedicated to the new system to verify that the refactoring (and possible extensions) have not broken the functionality of the original FARC application.

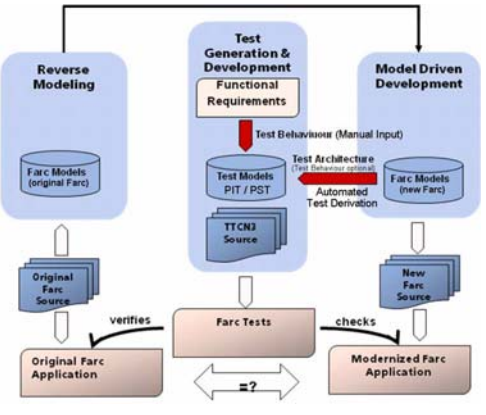


Fig. 2. General approach for model-based testing

For the configuration of test suites as well as for the identification of test interfaces we can rely on the refactored and modernized models (see Figure 2, right side). Test configurations and interfaces are derived from the FARC models using transformation technologies.

The distributed file archive (FARC) uses a server (FARC server) which is accessed by distributed clients. Each client is split in a user interface (FARCclientgui) and a model (FARCmodel). The gui is limited to represent the information and pass the input to the model, which implements the business logic. During this project the model was the target of modernization. The tests check the interaction between the FARCclientgui and the FARCmodel (see Figure 3). The test interface is the CORBA interface of the FARCmodel. All tests address the compound functionality of the FARCmodel and the FARCServer by checking the correctness of messages and the message interchange between the Tester (aka FARCclientgui) and the FARCmodel.

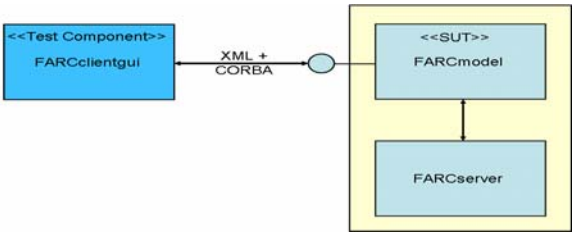


Fig. 3. Test Component Integration

As depicted in Figure 3, the FARCclientgui is replaced by a test component. The test component is connected with the CORBA interface of the FARCmodel which is considered to be the SUT. The communication is based on XML encoding on top of CORBA communication and encoding. Even though the FARCserver is not directly connected to the test component, it is implicitly tested due to the hidden communication between the FARCmodel and the FARCserver when the FARCmodel is contacted by the test component.

Since the available behavioural system models were not complete, the derived test models have been extended manually by additional test behaviour which has been derived directly from functional requirements specifications. The complete test derivation process started with the creation and annotation of behavioural and structural system models. We identified the requirements to be tested, annotated the components with respect to their role in the test process and identified and formalized setup (test preamble) and tear down (test postamble) behaviour.

The PIM resulting from the code and model abstraction process is the starting point for the test development (see Figure 1). From the PIM an initial PIT (platform-independent test model) is derived by adding preambles and postambles to the system behaviour as well as negative behaviour. From this initial PIT, a more complete PIT is automatically derived by adding timers, local and global verdicts, and alternatives in the test behaviour. In the following automated platform-specific transformation, the PIT is transformed into a PST (platform-specific test model), i.e. by adapting the communication between the test system and the SUT to the underlying platform technology the SUT is based on. In the case of the FARC application, the additional XML encoding is added to the test model. The PST is then transformed to TTCN-3 code, which is compiled to executable Java code, subsequently. In order to verify the test results, the original FARC application served as a reference and was used as a test oracle. This can be understood as a kind of regression testing. In case there is a difference between both test results, the original FARC application is considered to provide the right result, i.e. the modernized application failed.

During the project we discovered that the available models are only partially usable for test derivation. Especially we missed the availability of behavioural specifications. Without behavioural specification, only with the structural specification on hand, it is not possible to directly derive the interaction between the tester and the system under test. Structural specification can only be used to derive test data and interfaces and is not sufficient for a complete test specification. Moreover even a complete system model (structure and behaviour) does not cover the complete requirements for testing. System models are designed with the perspective of a system engineer. They specify

the system as it is (or as it should be) but in most cases they do not especially point out the usage of the system. A tester has to consider both: The specification of the system's functionality and structure to be able to determine the expected behaviour, and the usage of a system to identify the test scenarios and to derive the stimulation sequences. Thus, for future approaches it would be essential to integrate both perspectives in a common approach in order to benefit from reusing system models for test development and to get finally to a complete test specification with respect to the system requirements.

5.4 Metrication

Metrication is a process of measuring some properties of a software component. Usually the initial metrication results in a quantitative assessment of software, such as it size and complexity, and than a qualitative interpretation is given.

There is a number of well-known software metrics, such as a number of lines of code, cyclomatic complexity, cohesion and coupling, and others, but usually their definition contains a common sense description, and as such there is a leeway when performing the actual metrication. Our study found that even when considering the same programming language different tools provide different results for the same code and the same metric. Considering this fact and the difference in syntax and expressiveness of programming languages it is very difficult to perform a meaningful comparison of metrication results between different programming languages.

The other difficulty when dealing with metrics is the subjective factor of all qualitative interpretations. The notion of quality depends on the nature of the software in question and the tasks of its users and developers. There is a lot of quality models that attempt to describe software not from the point of its size and complexity (such as number of lines, number of classes or methods, number of links between classes) but from the point of its user properties (such as usability, reliability or maintainability). As with the low-level metrics these high-level software properties are easily defined using the common sense but the exact formulae or metrication methods are hard to define.

Considering this project's tasks we decided to use metrics to measure the changes made to the application during each phase of the migration process, and to measure the changes between to final result and the original application. The metrication tools were selected to provide the biggest common set of metrics between the different states of the application (the original C++ application, the UML model and the final Java application). It should be mentioned that the choice of metrication tools is very limited, and that the quality and features of such tools are lacking even for commercial variants. The resulting common set consisted of the following metrics: the number of lines of code, the depth of inheritance tree, the number of children of a class, the number of methods of a class, the cyclomatic complexity, the coupling between classes and the information flow measure. We defined a derived average complexity metric as a linear function from the values of these low-level metrics. The coefficients were chosen is such a way as to ensure that the average complexity of the original C++ software is equal to one to simplify the comparison with the following phases. The relative weights of each metric were chosen to be equal, but in general it is desirable to define relative weights on the basis of the project or application tasks. Table 2 overviews the metrication results.

Table 2. Metrication Summary

Derived Metric	C++ Code	C++ PSM	PIM	Java PSM	Java Code
Volume	299	331	291	268	274
Average Complexity	1,00	0,13	0,14	0,14	1,05
Total Complexity	299	43,43	39,35	36,77	289

The volume metric for C++ PSM does not represent the correct value since the problem of the C++ reverse mentioned in the section 5.1. In addition the PIM metric also include some platform skeleton classes left in the model for traceability. The real number should be less then in Java PSM. The code metrics for C++ and Java, as well as Java PSM are the cleanest and thus the most representative.

Using the described metrication methodology the migrated Java application got a score of 1,05. The result is quite expected since the size and complexity of the code that provides the same functionality in C++ and in Java is roughly the same.

The main conclusion made from the metrication activity is that the countable metrics by themselves (both low-level metrics that can be obtained automatically and all kinds of derived high-level metrics) can not provide a verdict on the system quality and maintainability. They should be used together with other code quality assessment technologies, such as automatic tests and static analysis.

6 Conclusions

The methodology established for the case study was successfully applied to the FARC system. The Functional PIM model was extracted using specially developed tooling. The same process allowed for identifying the platform dependencies. This information was used in the next phase for identifying correspondent substitute libraries in Java. During the PSM derivation, the new Java platform dependencies were integrated with the PIM model by means of specially developed tooling features. The obtained Java PSM model was used for generation of the Java code skeletons.

The generated Java skeletons were filled manually with the required functionalities applying the established migration rules. Despite usage of the MDA, the development effort was rather significant. However, the process resulted not only in the code but also in a complete UML2 model containing the documentation and requirements link. This links were automatically maintained in the model during the development process by means of the model/code synchronisation feature.

Metrication methodology established at the beginning of the project was successfully applied throughout all the stages: from the analysis of the original C++ application through the finally re-engineered JAVA application. However, the metrication process requires quite a considerable “human intervention” and careful analysis. It is not a straightforward procedure that could be easily automated. The reengineering approach used in the RTE Space project resulted in the Java application that is very similar to the original C++ application in terms of metrics. The resulting Java code demonstrated less volume than the original C++ application, but slightly increased average complexity.

Finally, the test campaign validated the correctness of the migration from the network protocol – behavioural point of view.

The major goals of the study were achieved: (1) a formal methodology for round-trip engineering of space systems was developed; (2) the current state of the art in applicable of-the-shelf technologies and tools was analysed; and (3) the feasibility of the approach was demonstrated.

References

1. Wadsack, J.P., Jahnke, J.H.: Towards Model-Driven Middleware Maintenance. In: OOPSLA 2002, Seattle, Washington, USA (2002)
2. Reus, T., Geers, H., Deursen, A.: Harvesting Software Systems for MDA-Based Reengineering. In: Rensink, A., Warmer, J. (eds.) ECMDA-FA 2006. LNCS, vol. 4066, pp. 213–225. Springer, Heidelberg (2006)
3. Boronat, A., Carsi, J.A., Ramos, I.: Automatic reengineering in MDA using rewriting logic as transformation engine. In: CSMR 2005, Manchester, UK (2005)
4. Vigier, L., Sadovykh, A.: PSM-to-PIM, a Pragmatic Way. In: ECMDA 2008 – Workshop on Model-driven Modernization of Software Systems, Berlin (2008)
5. Busch, M., Chaparadza, R., Dai, Z.R., Hoffmann, A., Lacmene, L., Ngwangwen, T., Ndem, G.C., Ogawa, H., Serbanescu, D., Schieferdecker, I., Zander-Nowicka, J.: Model Transformers for Test Generation from System Models. In: Conquest 2006. Hanser Verlag, Berlin (2006)
6. Zander, J., Dai, Z.R., Schieferdecker, I.:From U2TP Models to Executable Tests with TTCN-3 – An approach to Model Driven Testing. In: IFIP 17th Intern. Conf. on Testing Communicating Systems - TestCom 2005, ISBN: 3-540-26054-4, (2005)
7. Baker, P., Dai, Z.R., Grabowski, J., Haugen, O., Schieferdecker, I., Williams, C.: Model-Driven Testing: Using the UML Testing Profile. Springer, Heidelberg (2007) ISBN 3540725628, 9783540725626
8. Engel, K.D., Rennoch, A., Schieferdecker, I.: Architecture-driven Test Development. In: 1st Workshop on Model-based Testing in Practice (MoTiP 2008), Berlin, Germany (2008)
9. Chaparadza, R., Busch, M., Dai, Z.R., Hoffmann, A., Lacmene, L., Ngwangwen, T., Ndem, G.C., Serbanescu, D., Schieferdecker, I., Zander-Nowicka, J.: Transformations: UML2 System Models to U2TP models, U2TP models to TTCN-3 models and, TTCN-3 Code Generation and Execution. In: ECMDA 2006 Workshop on Model-Based Testing, Bilbao, Spain (2006)
10. Henderson-Sellers, B.: Object-Oriented Metrics: Measures of Complexity. Prentice-Hall, Inc., Englewood Cliffs (1996)

MoPCoM/MARTE Process Applied to a Cognitive Radio System Design and Analysis

Ali Koudri[1,2], Joël Champeau[2], Denis Aulagnier[1], and Philippe Soulard[3]

[1] Thales Aerospace Division
firstname.lastname@fr.thalesgroup.com
[2] Ensieta
joel.champeau@ensieta.fr
[3] Sodius
psoulard@sodius.fr

Abstract. Today, developments of real time embedded systems have to face new challenges. On the one hand, economic laws, such as Time-to-market, require a reliable development process allowing quick design space exploration. On the other hand, fast increasing technology, as stated by the Moore's law, requires techniques to handle the resulting productivity gap. Model Driven Development has been widely used in response to those issues. Benefits of such approach are numerous and have been demonstrated through several experiments. We present in this paper the Model Driven Development MoPCoM methodology, dedicated to SoC / SoPC design and analysis, and based on the use of the MARTE profile. This approach refines the MDA Y-Chart in order to ease design space exploration and IP integration. We illustrate our approach on Cognitive Radio System development implemented on an FPGA. This work is part of the MoPCoM research project (http://www.mopcom.fr) gathering academic and industrial organizations.

1 Introduction

Since few years, the market of System-on-Chip (SoC) has grown rapidly. It is expected to worth $56 billion in 2012, which represents almost 24% annual growth rate. As the technology evolves rapidly, according to the Moore's law, entire systems, made of processors, memories or sensors, can now be integrated on SoC or SoPC (System-on-Programmable-Chip) like FPGA (Field Programmable Gate Array).

Indeed, only reliable methodologies, based on well-adapted formalisms and tools, can handle the growing design complexity of such systems. The challenges posed by design of SoC/SoPC consist mainly in reducing TTM (time-to-market), costs and the productivity gap due to the rapid evolution of the technology [1]. To achieve those goals, SoC/SoPC design methodologies have to tackle co-design issues such as design space exploration, reuse of IPs (Intellectual Property) and high level synthesis.

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, pp. 277–288, 2009.

Model Driven Development (MDD) adds valuable contributions to SoC/SoPC Design: Analysis enhancement, Communication and Traceability improvement, Technology breakpoints reduction, etc.

Recently, the OMG (Object Management Group) which has supplied the Unified Modeling Language (UML[2]) and the Model Driven Architecture (MDA[3]) paradigm, has standardized the UML profile for Modeling and Analysis of Real Time Embedded Systems (MARTE profile[4]). This profile extends UML for Real Time Embedded Systems (RTES) design and analysis.

In this paper, we present a new methodology dedicated to SoC/SoPC design called MoPCoM. It is based on the use of MARTE profile and is tooled in the Rhapsody®modeling tool. In the next section, we present related works on SoC/SoPC design. We then provide a general overview of the methodology illustrated on a cognitive radio design implemented on an FPGA.

2 Related Works

Transforming customer requirements into implementations making good tradeoffs between performances and costs requires relevant analysis activities based on appropriate languages and tools. Since few years, the trend in Electronic System Level (ESL) is to foster fast design space exploration providing executable or verifiable specifications to customers at several abstraction levels in order to mitigate risks and avoid useless expenses and waste of time.

Several approaches have been applied in order to tackle issues posed by economic laws and rapid evolution of technology. Among them:

- High Level Synthesis [5] aims at transforming high level behavior specification into optimized architecture,
- IP reuse aims at building systems assembling IPs which requires standard interfaces or wrappers [6],
- Platform Based Design [7] consists in configuring a generic platform containing configurable components like micro-processors or FPGA to suit a specific kind of application.

Related works on MDA based RTES methodologies and associated tools are numerous. In this section, we give a brief overview of the main ones. In [8], authors highlight which native concepts of UML to use in order to design RTES in the ACCORD/UML methodology. In [9], authors present a methodology based on the use of UML and Platform Based Design called *Metropolis*. They highlight the necessary orthogonalization of several aspects: computation and communication, function and architecture, behavior and performance. They provide a set of stereotypes and a dedicated framework supporting function / platform mapping, refinements and code generation. In [10], authors present an object oriented methodology based on the use of UML: the *HASoC methodology* (Hardware and Software Object on Chip). They first provide an abstract model of the system that is executable (uncommitted model) and proceed to the partition into hardware and software parts taking into account implementation constraints (committed model). In [11], authors combine the visual features

of UML with the simulation and debugging features of the SystemC language in a methodology called UPES (Unified Process for Embedded Systems). This "specify-simulate-debug-refine" methodology is based on the use of the UML for SystemC profile and allows users to capture behavioral and structural aspects of the system at several levels of abstraction. The Gaspard methodology[12] is dedicated to Multi-Processors Systems-on-Chip (MPSoC) design. It is based on the use of a dedicated profile called "Gaspard Profile" allowing regular repetitive structure platform modeling. The Gaspard environment provides TLM and RTL code generation and bridges to several analysis tools. At last, the Harmony/ESW (Embedded System Workflow)[13] is a Rational Unified Process (RUP) based methodology dedicated to RTES design. It describes the rationale that guide RTES development from requirements capture to implementation using SysML[14] and SPT[15] profiles.

In the next sections, we present the MoPCoM process and an overview of the application that has been implemented. It is based on the use of SysML, Software Defined Radio (SDR) and MARTE profiles.

3 Process and Application Overview

The MoPCoM methodology is a refinement of the MDA Y-chart dedicated to design space exploration and Platform Based Design. It takes as input functional, non-functional and allocation requirements expressed in SysML. Figure 1 gives an overview of the process, highlighting 3 modeling levels:

- The *Abstract Modeling Level* (AML) is intended to provide the description of the expected level of concurrency and pipeline through the mapping of functional blocks onto a virtual execution platform,
- The *Execution Modeling Level* (EML) is intended to provide a generic platform defined in terms of execution, communication or storage nodes in order to proceed to coarse grain analysis,
- The *Detailed Modeling Level* (DML) is intended to provide a detailed description of the platform in order to proceed to fine grained analysis. It allows RTL code generation for hardware (VHDL) and software (C) parts including glue code (drivers).

For each level, we identify which subsets of MARTE are mandatory and we capture modeling rules through OCL constraints applied to the process model. Moreover, support for automatic code and documentation generation as well as metrics audits are implemented in the Rhapsody ®modeling tool using the MDWorkbench transformation tool (see section 8).

We have developed as a demonstrator for the MoPCoM methodology a Cognitive Radio System (CRS) that adapts its behavior to its environment [16,17] following the characteristics of available Radio Access Technologies (RAT), such as bandwidth or localization, in the electro-magnetic environment. Depending on the load, an area is selected and the CRS dynamically configures itself in order to communicate using corresponding protocol. We focus in this paper on the

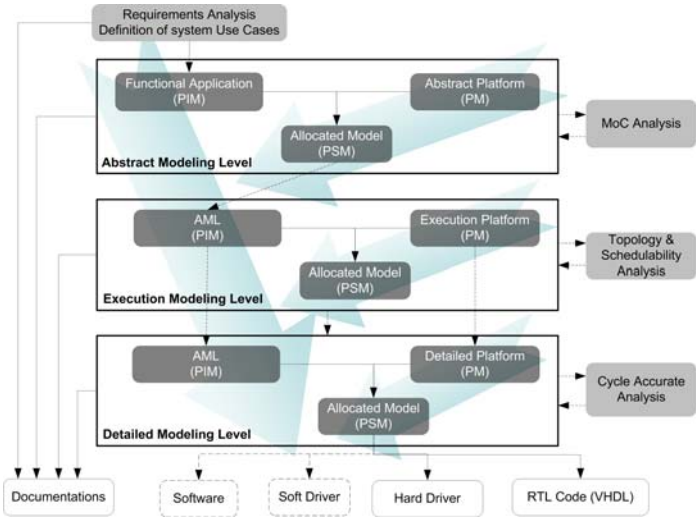


Fig. 1. MoPCoM Process Overview

"Locate RAT Source" Use Case which consists in detecting radio communication signals and characterize their main parameters. This system is implemented on a Xilinx ML-506 FPGA, including analog and digital devices.

4 Requirements Capture and Functional Analysis

Meeting functional as well as non-functional requirements in time and budget is the goal of every development. Requirements formalization is mandatory to achieve those goals and avoid any ambiguity. As requirements formalization is still an open issue, we define in this paper formalized requirements as requirements that are "executable or verifiable".

Using Use Cases is a good approach in order to formalize, analyze and document functional requirements, leaving aside implementation issues. Use cases are analytical tool capturing the causality links between events and behaviors. Moreover, Use Case scenarios carry over directly into the testing process through the interfaces identification.

In figure 2, Use Cases ① are described through sequence diagrams ② capturing nominal, alternate or exception usage scenarios. MARTE annotations ③ are used to capture real-time constraints written in VSL (Value Specification Language) provided by the profile and dedicated to expression of Non-Functional Properties. All instant or duration observations are related to an ideal clock that measures the progression of the real time, which is provided in the MARTE model library. Use case executability [13] requires the definition of its behavior ④ using activity diagram or statechart and regarding the causality model described in the scenarios.

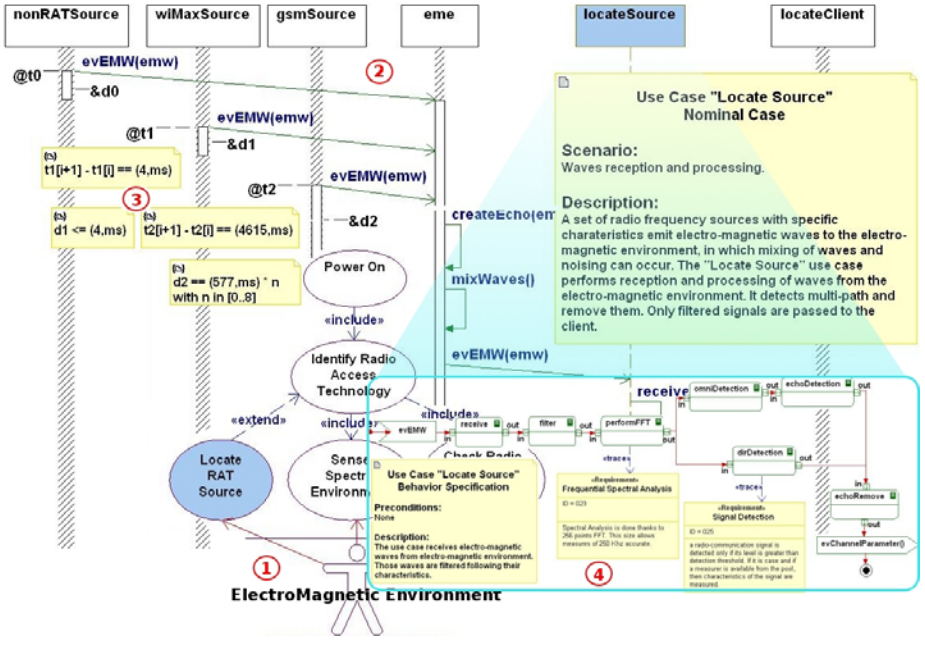


Fig. 2. Locate Nominal Scenario with Real-time constraints and Traceability

The purpose of the functional analysis is to identify business classes, their relationships and their responsibility as well as possible patterns that can be applied in order to address specific issues such as dynamic reconfiguration expressed at functional level. This can be achieved through black box / white box analysis and iterative breakdowns followed by scenarios and behavior refinements.

Efforts must focus on the functional design in purpose of reuse, flexibility or configurability. Functional blocks are defined against their provided and required interfaces in order to foster loose coupling between block definitions. Some of the well-known patterns successfully applied in the software domain[18] can be reused in the system domain as well as in the hardware domain.

At this point, we describe Signal Processing in C/C++ as input to the Mentor Graphics CatapultC®and GAUT(`http://web.univ-ubs.fr/lester/www-gaut/`) tools in purpose of High Level Synthesis. For each block stereotyped «HLS», standing for *High Level Synthesis*, C code is generated.

5 Abstract Modeling Level

While the functional analysis deals with the algorithmic aspects of the system, this level focuses on concurrency, time and communications. Indeed, we need appropriate concepts, related to Models of Computation (MoC), in order to proceed to relevant analysis like performance analysis or deadlock detection.

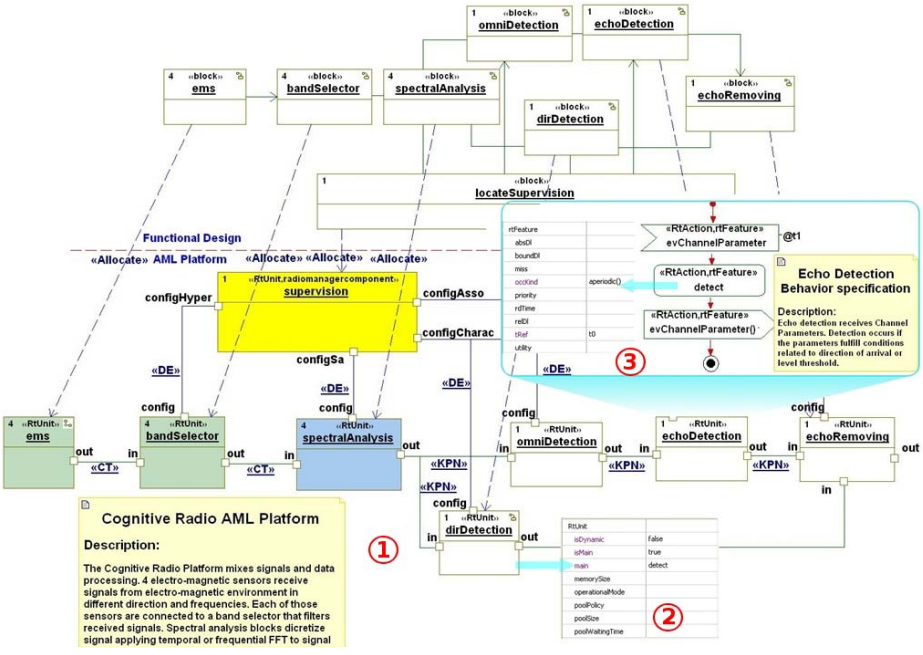


Fig. 3. Functional to AML Platform Allocation

A MoC defines the semantics that rule concurrent behaviors and communications. It has several characteristics related to time (causality, clocked or physical time), function (algorithm or Ordinary Differential Equation ODE), data (abstract data type, bit true or bit vector) and communication (Inter-Process Communication, shared variables, FIFO, wires), depending on the considered abstraction level.

MoCs provide different capabilities in terms of analysis and one of the goals of system designers is to select appropriate MoCs and proceed to required analysis. Tools supporting mixing several MoCs like Ptolemy [19] are useful when system under study combines heterogeneous parts (analog, digital, GALS, etc).

Concurrency in UML can be expressed at several levels. At the behavioral level, one can use for example AND states (state machines) or fork nodes (activity). At the structural level, the meta-attribute "isActive" of the UML Class bears the notion of concurrent entity. Concurrency patterns [20] can also be used in order to address this issue. Still, missing information related to real-time features are required to proceed to appropriate analysis.

The MARTE profile provides all those missing features through the notion of *RTUnit* in the High Level Application Modeling (HLAM) sub-profile. «RTUnit» stereotype can be used to classify a set of concurrent objects having real-time characteristics. Actually, an *RTUnit* is a SysML Block with additional real-time features. An RTUnit provides / requires a set of real-time services (*RTService*)

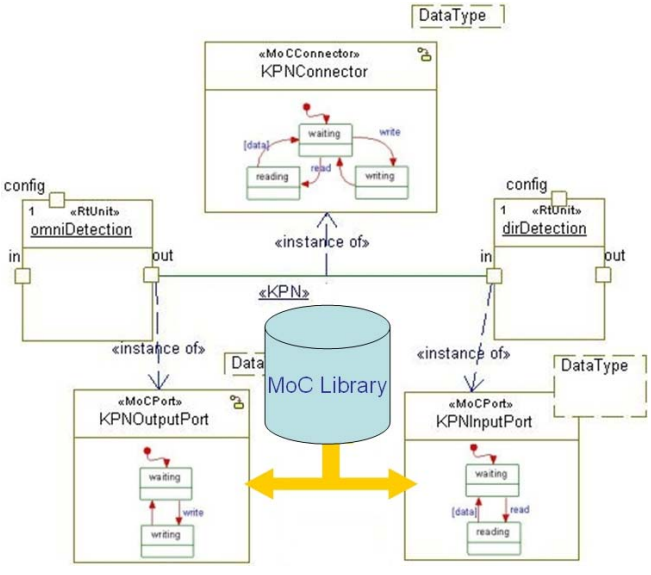


Fig. 4. KPN MoC Support

and owns at least one real-time behavior (*RTBehavior*) with (un)bounded queue. Real-time behaviors can be decomposed into real-time actions (*RTAction*).

Communications are point-to-point and occur through *RTeConnectors* between «RTUnit»instances in the context of their owning classifier. Ports and *RTeConnectors* implement the high level primitives that support communication mechanisms.

For instance, figure 3 shows an allocation of the functional design ① to AML platform ②, which mixes three models of computation: CT (Continuous Time), KPN (Kahn Process Network) and DE (Discrete Event). Functional behaviors are turned into real-time behaviors ③, for which we precisely specify real-time features. For example, the KPN MoC is supported thanks to a components model library into which we define "KPN Port" and "KPN Connector" as behaviored classifiers (figure 4) that are instanciated each time the «KPN»stereotype is used on ports or connectors. Transformations rules adapt then the functional behavior to take into account the knowledge of those communication mechanisms.

Allocation constraints are applied in order to precisely specify the MoC, especially constraints related to required level of concurrency from which special blocks dedicated to data multiplexing / demultiplexing can be inferred.

6 Execution Modeling Level

This level aims at analyzing execution of the AML allocated model onto an abstract execution platform. Performed analysis abstract costs related to Hardware (CPU, FPGA) or Hardware Abstraction Layers (HAL – Operating Systems, Middlewares) through statistical evaluations [21].

Platform and allocation are defined regarding trade-offs between implementation, performances and costs. In the context of design space exploration, all those aspects must be checked by a top-down refinement analysis. EML architectural designs emphasize the number of processing elements, organization of data or communication media and their characteristics. Interfaces between connected elements and communication protocols are considered from a high level perspective: data are bit true and transactions are inaccurately timed.

Main nodes of interest are computing resources gathering programmable components as well as hardware accelerators, storage resources gathering all kinds of data storage, communication media gathering all kinds of communication link providing transaction services, scheduling resources scheduling tasks and managing access to resources. All those resources are characterized by their relative execution speed and services they offer to the application.

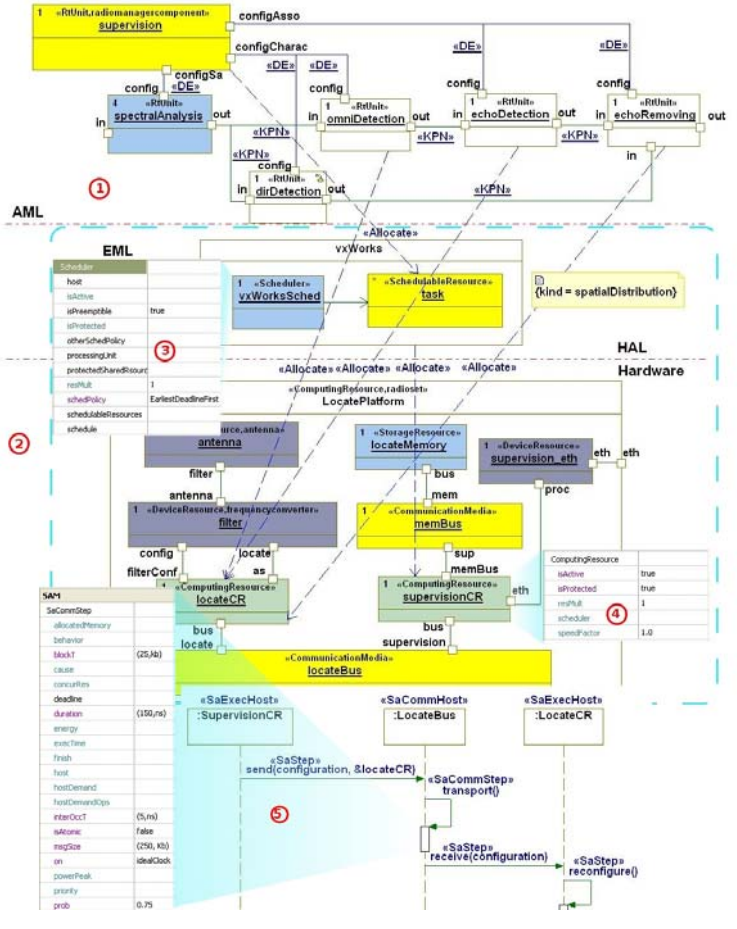


Fig. 5. AML to EML Platform Allocation

Allocation is defined with respect to expected performance or other non-functional concerns and clocks are used to measure the progression of physical time in order to check real-time constraints.

In order to model abstract execution platforms, the MARTE profile provides the *Generic Resource Modeling* (GRM) sub-package. It allows modeling physical or logical persistent entities providing a set of resource services. A resource is characterized by several non-functional properties such as latency, cost or power consumption. Besides, it deals with resources usage and must be used in conjunction with the *Generic Quantitative Analysis Modeling* (GQAM) package and their subpackages: *Performance Analysis Modeling* (PAM) and *Schedulability Analysis Modeling* (SAM) in purpose of analysis. The main goals of this level are communication bottlenecks detection and coarse grain performance estimations.

The figure 5 shows a simplified view of the allocation of the AML architecture ① onto the EML platform ②. For instance, this platform contains two levels of allocation: one representing the HAL and one representing the hardware nodes. Computing resources can be targetted by either spatial or temporal allocations (or both). For example, the *Supervision Computing Resource* drives the configuration of each element and dynamic reconfiguration and is characterized by a relative speed factor equal to 1.0 ④. The HAL contains one scheduler managing tasks with an Earliest Deadline First (EDF) policy ③. Interfaces and communication protocols are defined on a transactional level such as OCP/IP TLM [22]. For instance, the communication media implements a *transport* primitive and owns several NFP characteristics ⑤.

Analysis scenarios emphasize data rates and latencies as well as memory size or context switching in the case of dynamic reconfiguration. For example, we show in figure 5 an analysis scenario highlighting costs of resources usage ⑤. Analysis feedbacks drive potential refactorings on the platform as well as on the MoC or the functional architecture. Actually, those refactorings depend mainly on the skill of the engineers to proceed to the right adjustments.

7 Detailed Modeling Level

This levels aims at providing more accuracy to the previous model in order to allow relevant analysis leading to RTL code generation, e.g. C or VHDL code. The topology that has been defined in the EML level is refined. For instance, computing nodes are turned into processors or FPGAs, storage nodes become SRAM or ROM, scheduling nodes become Operating Systems, etc.

Two levels of descriptions of the hardware parts are provided: one focuses on the logical features (services) while the other focuses on the physical feature (non-functional properties). For instance, hardware services can be classified in computing, storage or communication services and hardware non-functional properties include features like dimension, weight or even price. Refinement of the platform involves data, interfaces or protocol refinements.

Hardware abstraction layers provide set of services dedicated to tasks and resources management. Actually, HAL are characterized by their API. Examples of such API can be found in the POSIX(http://standards.ieee.org/regauth/posix/) or OCP/IP (http://www.ocpip.org/home) specifications.

The *Software Resource Modeling* (SRM) and *Hardware Resource Modeling* (HRM) MARTE sub-profiles provide concepts to describe such a detailed platform. The SRM package provides a set of concepts related to HAL modeling including operating systems as well as middlewares while the HRM package provides concept to model platforms from logical and physical perspectives.

Time model is cycle accurate and data are bit accurate. Allocation and analysis from the previous level are refined in order to take into account platform refinement.

8 MoPCoM Tooling

The MoPCoM process tooling relies on tools related to OMG standards (MDA, UML, MOF, XMI) and Eclipse (EMF, EMOF, ECore):

- The KerMeta language from INRIA[23] is used to formalize and validate the metamodels (concepts and constraints),
- The Rhapsody UML Modeler is used to model applications as well as platforms according to the defined levels,
- The Sodius MDWorkbench tool is used to transform models (model-to-model) and generate code or documentation from models (model-to-text).

The generator is delivered as a white-box add-on, where all transformations and generation rules are available for any customization. In addition, the MoPCoM methodology has been modeled using a SPEM [24] derived metamodel dedicated to co-design process modeling in purpose of better capitalization or improvement.

RTES modeling requires an action language for low-level expressions to complete the high-level UML semantics and diagrams, and to specify operation bodies, trigger/guard/action on transitions and states as well as data declarations. The selection of the right action language raises questions about textual or graphical notation, and general versus HDL-specific language accessible to designers, taking into account learning curves. The C++ language turned to be a convenient choice and only a C++ subset is used in the models (along with some macros for event and port handling). C++ expressions are parsed for VHDL generation thanks to a C++ syntactic metamodel allowing grammar to model transformations.

VHDL code generator, whose deliverable is synthesizable VHDL code, takes as input DML allocated model. Structural parts are derived from the platform model, where VHDL entities are derived from hierarchy of instances. UML ports are translated to VHDL ports thanks to communication protocols and data. Behavioral parts are derived from application models, where VHDL architectures are mainly issued from attributes, operations and state machines.

Briefly, a finite state machine leads to the definition of an enumerated type for the active state, one per composite state (containing sub-states). The code

structure is based on an edge-clocked case VHDL statement, and all trigger, guard and action expressions (on transitions, entering, in or exiting states) can be generated either in line, or in single procedures. The allocation package brings additional information about the mapping of the application on the platform. The generator combines the declared entity ports and the data/control needs of the architecture to map the components and if required (if not point-to-point), instantiate the control code (or state machine) of the communication channel protocols. Depending on the communication channel, several basic mechanisms are provided to handle events (transient or registered) and the required glue code is automatically inserted.

9 Conclusion

In this paper, we have presented our SoC/SoPC Design Flow based on the use of UML and dedicated profiles. The tool has generated 55420 lines of code for 289 classes including business classes, data structures, interfaces and communications. Although some improvements can be done, particularly in the MoCs support, we have shown that MDD techniques, based on the use of MARTE profile, can fit into Co-Design through an example of Cognitive Radio Application implemented on FPGA. This MDD approach refines the MDA Y-Chart in order to tackle achievements of the ESL community. Further works will be focused on providing a better support for High Level Synthesis code generation. This work is part of the MoPCoM project (http://www.mopcom.fr), gathering academic and industrial organizations and supported by the French Agence Nationale de la Recherche (RNTL 2006 TLOG 022 01), the "Media and Networks" "cluster of clusters" and Brittany and Pays de la Loire regions.

References

1. ITRS: Design. Technical report, International Technology Roadmap For Semiconductors (2007)
2. OMG: UML 2.0 superstructure. Technical Report formal/05-07-04, Object Management Group (2005)
3. OMG: Mda guide version 1.0.1. Technical report, Object Management Group (2003)
4. OMG: Uml profile for marte, beta 1. Technical Report ptc/07-08-04, Object Management Group (2007)
5. Stitt, G., Vahid, F., Najjar, W.: A code refinement methodology for performance-improved synthesis from c. In: ICCAD 2006: Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design, pp. 716–723. ACM, New York (2006)
6. Koudri, A., Meftali, S., Dekeyser, J.-L.: IP integration in embedded systems modeling. In: 14th IP Based SoC Design Conference (IP-SoC 2005), Grenoble, France (December 2005)
7. Sangiovanni-Vincentelli, A., Carloni, L., Bernardinis, F.D., Sgroi, M.: Benefits and challenges for platform-based design. In: DAC 2004: Proceedings of the 41st annual conference on Design automation, San Diego, CA, USA, pp. 409–414. ACM, New York (2004)

8. Gerard, S., Terrier, F.: Uml for real-time: which native concepts to use? ACM 13, 17–51 (2003)
9. Chen, R., Sgroi, M., Lavagno, L., Martin, G., Sangiovanni-Vincentelli, A., Rabaey, J.: Uml and platform-based design
10. Edwards, M., Green, P.: Uml for hardware and software object modeling, pp. 127–147 (2003)
11. Riccobene, E., Scandurra, P., Rosti, A., Bocchio, S.: Designing a unified process for embedded systems. In: The Fourth International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES), Braga, Portugal. IEEE Computer Society, Los Alamitos (2007)
12. Piel, E., Attitalah, R.B., Marquet, P., Meftali, S., Niar, S., Etien, A., Dekeyser, J.L., Boulet, P.: Gaspard2: from marte to systemc simulation (March 2008)
13. Douglass, B.P.: Real-Time Agility: The Harmony Method for Real-Time and Embedded Systems Development. Addison-Wesley Professional, Reading (2009)
14. OMG: Systems modeling language specification v1.1. Technical Report ptc/2008-05-16, Object Management Group (2008)
15. OMG: Uml profile for schedulability, performance, and time, version 1.1. Technical Report formal/2005-01-02, Object Management Group (2005)
16. Mitola Joseph, I.: Cognitive radio for flexible mobile multimedia communications. Mob. Netw. Appl. 6(5), 435–441 (2001)
17. Hachemani, R., Palicot, J., Moy, C.: A new standard recognition sensor for cognitive radio terminals. In: EURASIP, Kessariani, Greece (2007)
18. Gamma, E., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
19. Buck, J., Ha, S., Lee, E.A., Messerschmitt, D.G.: Ptolemy: a framework for simulating and prototyping heterogeneous systems. IEEE 10, 527–543 (2002)
20. Douglass, B.P.: Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems. Addison-Wesley Professional, Reading (2002)
21. Kountouris, A.A.: Safe and efficient elimination of infeasible execution paths in wcet estimation. In: RTCSA 1996: Proceedings of the Third International Workshop on Real-Time Computing Systems Application (RTCSA 1996), Washington, DC, USA, p. 187. IEEE Computer Society, Los Alamitos (1996)
22. Bacchini, F., Maillet-Contoz, L., Kashiwagi, H., Donovan, J., Makelainen, T., Gajski, D.D., Greenbaum, J., Nikhil, R.S.: Tlm: crossing over from buzz to adoption. In: DAC 2007: Proceedings of the 44th annual conference on Design automation, pp. 444–445. ACM, New York (2007)
23. Muller, P.-A., Fleurey, F., Jézéquel, J.-M.: Weaving executability into object-oriented meta-languages. In: Proc. of MODELS/UML, Montego Bay, Jamaica. LNCS. Springer, Heidelberg (2005)
24. OMG: SPEM 1.1. Technical Report ptc/05-01-06, Object Management Group (2005)

Managing Flexibility: Modeling Binding-Times in Simulink

Danilo Beuche[1] and Jens Weiland[2]

[1] pure-systems GmbH, D-39106 Magdeburg, Germany
danilo.beuche@pure-systems.com
[2] Reutlingen University, Department of Technology, D-72762 Reutlingen, Germany
jens.weiland@reutlingen-university.de

Abstract. Model-based development is supposed to improve the development efficiency by raising the abstraction level and generating applications instead of manually coding the application in low level languages like C. One of the successful incarnations of this idea is the MATLAB Simulink tool chain. These tools are now widely used in the automotive industry not only to simulate control devices but also to generate product quality code from it. Like with traditional concepts reuse of created models is an issue. When this can be done efficiently, an additional level of effort reduction (and quality improvement) will be achieved. While MATLAB Simulink in combination with code generators provides good support for creating models for single application, and libraries of models, it does not provide sufficient support for more complex reuse scenarios with fine grained variations across the model(s). This paper will extend the approach developed by an automotive car manufacturer to address these issues. After a discussion of the basic concept the paper will put a special focus on support for flexible binding times since this is one of the crucial issues for reusing models across different projects with different need for run-time switchable variations (development/testing) and static decisions for product generation (resource efficiency in terms of code size and run-time). A concrete application supporting the deployment of these concepts in a project, developing mass production control units, is discussed in the following section.

1 Introduction

While in its early days, the automotive industry produced relatively simple cars with not much variability, like the Ford Model T, todays cars are highly configurable and provide quite a lot of variability. Due to the huge amount of vehicle configurations and varying requirements related to differences in hardware, regulatory, and market behavior there are many aspects of this variability [14]. Today embedded software plays a central role since most functions in a car are software-based. Thus, functional variability leads directly to variability in the embedded software. Development of embedded software is increasingly done model-based, using code generators to produce production code directly from

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, pp. 289–300, 2009.

models. Mathworks Matlab tool chain with its extensions Simulink and Stateflow is an important exponent used in the automotive industry. The graphical modeling languages enable developers to specify, model, and simulate signal flow oriented and state based systems. The implementation can be done using code generators like The Mathworks Realtime Workshop Embedded Coder or dSpaces TargetLink. Simulink models are basically signal flow graphs composed of elementary blocks and state charts. Complex subfunctions can be represented as sub-system blocks, which internally are again signal flow graphs. Signal connections represent data that flows between model blocks while simulating the model. Simulink provides a large set of blocks for functions like logical operations or signal routing. Each block has a set of parameters, which allow the configuration of the blocks behavior, properties, and its presentation. Code generators like TargetLink add additional parameters to maintain code generation related information. To properly deal with variability in the automotive context, an economic view on reuse of specifications, models, architectures, components, or even documentation and tests, is essential. Model-based software development, with a holistic use of models in all phases of development including abstraction from concrete target platforms, is an important step towards improved reuse. However, when it comes to systematic consideration of variability (e.g. variable data or functions), additional concepts are a necessity. Especially, when applying model-based development using Simulink and Stateflow, deficits with respect to variability handling become obvious:

- There are only insufficient description capabilities for modeling of structural variability in Simulink. This variability cannot be described unambiguously, and process safe. Process safe in this context means that in every phase of software development variability and its implications can be traced unambiguously.
- Due to missing variability handling concepts in Simulink checking of model configurations in order to detect invalid variability configurations is not possible.

[6] describes basic concepts for systematic modeling and configuration of variability in Simulink, used to develop automotive software. The article presented here goes further and discusses a concept for implementing flexible binding time support. Binding times specify the point in time at which variability is removed and a concrete instantiation of a vehicle function is selected. Depending on the use case, an optional vehicle function may be removed completely, thus, there is no code for it on the vehicles electronic control unit (ECU) at all, or is just turned off using a data parameter, but remains present in the code and can be activated later, e.g. at a service station or at run-time. Selection of binding times has a strong influence on the amount of code present in a vehicle (in general, the later the decision is made, the more code is generated). Flexible selection of binding times permits to control this by choosing for each point of variation the optimal binding time. Due to the high number of produced hardware units even small difference in code size can have a huge impact on production cost of these units. The remainder of this paper describes the approach for flexible

binding time support based on the concepts from [6]. In section 2 basic concepts are presented. Section 3 takes a closer look at binding times, followed by section 4, which discusses in detail the taken approach for modeling and configuration ofing binding times. The application of these concepts is shown in section 5. Section 6 provides a summary of the paper. The presented concepts are based on the tool chain Matlab 7.1, TargetLink 2.1 and pure::variants 2.4.

2 Modeling and Configuring Variability in Simulink

The concepts for handling variability in Simulink presented in [6] are based on the Generative Software Development Model [4]. Based on the engineering of system families its main goal is an automated generation of individual family members. The main element of this concept is the Generative Domain Model. It separates application oriented concepts (problem domain concepts) from the implementation (solution domain concepts). Configuration knowledge is used for mapping both domains onto each other and, thus, it explicitly links them together. To describe the application oriented concepts feature modeling is being used. Feature models provide an abstract representation of variability in a system family on the problem domain level. Feature models contain common and variable features and their dependencies [2,4,12]. Starting point for describing variability in Simulink-models is the variation point (see meta model shown in Fig. 1). The variation point encapsulates variability information, which has to be added to the Simulink model in order to allow systematic handling of variability.

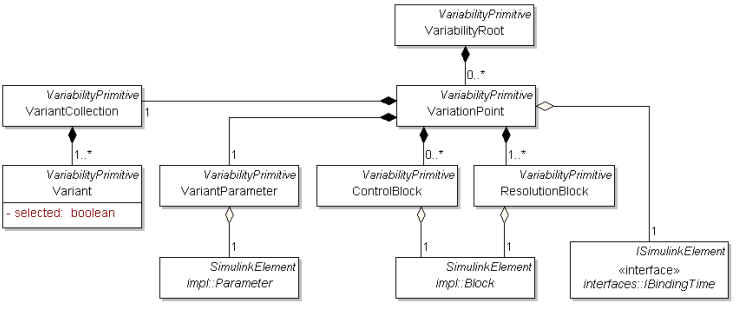


Fig. 1. Variation point meta model

Central element of a variation point is its uniquely identifiable *VariantParameter*. This parameter is the actual point where functional variants are created by applying different settings to this parameter. The variation point permits only for a finite number of choices. These choices are represented in the *VariantCollection* and are mapped to unique variant parameter values. According to [9] a variation point identifies one or more locations at which variation will occur.. These locations are internally represented by mechanisms, which remove variability once the variation point has been configured. The block library of

Simulink provides a number of blocks which can be used to realize variability mechanisms [10,15]. Examples are:

- Conditionally executed subsystems (*Enabled Subsystem*-block, *Function Call Subsystem*-block),
- Signal routing blocks (*Switch*-block, *Multiport Switch*-block),
- Logical gates (*AND*-block, *OR*-block),
- Configurable subsystems (*Configurable Subsystem*-block),
- ...

Each of these blocks has its own way of controlling the variable functionality. E.g. a function encapsulated in an *Enabled Subsystem* can be (de)activated depending on the value of the *Enabled* signal sent to this block. Thus, the *Enabled Subsystem* is well suited for representing optional functionality in the model. In similar ways logical conditions represented by *AND* or *OR* blocks are able to control the execution of optional functionality by controlling the transmission of input signals depending on the setting of a selected input signal. The *Switch*-block selects a variant based on a *Control*-signal, similar to the if-else found in programming languages like C. Therefore the *Switch*-block is well suited especially for modeling alternative functionality. Blocks which resolve variability are called *Resolution Blocks* in our context. Most of these blocks require an input signal, which controls the execution of the block. To perform the actual selection of a variant a designated *Control Block* should be used for providing the input signal. This block can be represented by a parameterized *Constant*-block or a *Data Store Read*-block. Depending on the value represented by the control block, the respective variant is executed.

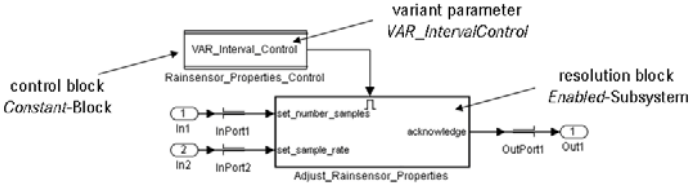


Fig. 2. Variability mechanism example

Fig. 2 shows an Enabled Subsystem whose execution is controlled by a *Constant*-block. The *Constant*-block references the workspace parameter `VAR_IntervalControl`. In our example this parameter represents the Variant-Parameter. Depending on the value of this parameter one or zero the Enabled Subsystem `Adjust_Rainsensor_Properties` will be executed or not. The set of control and resolution blocks blocks is called *Variant Blocks*. To clearly separate these blocks from other Simulink blocks, an additional mask parameter, e.g. `VarInfo`, is used. Mask parameters extend *Simulink*-blocks with additional properties. By adding the parameter `VarInfo` to a block, the regarded block will be treated as variant block. The mask parameter references the variation point, to which the block is related.

3 Taking Binding Time into Account

A variation point is a kind of place holder, which must be filled by a valid variant instance at a certain point during creation of a system variant. The decision, which of the possible instances for variation point to select, is called *binding*. And consequently the time at which this decision occurs is called *binding time*. There are different binding time models described in literature ([3,8,11]), there are even specialized models for embedded automotive software [7]. The binding time model used for the approach in this paper is based on the FODA binding time model [11]. It distinguishes between three different binding times:

- CompileTime: The variation point is bound during compilation.
- LoadTime: The variation point is bound when the system starts up, e.g. by reading parameters from databases or from the system environment.
- RunTime: Variation points can be bound at any time during the execution of the system.

Since model based development introduces additional abstraction level at which variation points can be bound, the approach introduces a new binding time:

- ModelConfigurationTime: Variation points are bound when a variant-rich model is instantiated / (in our context configured) to represent a single variant.

In the automotive context most decisions are taken before run-time due to improved performance and resource consumption. There are also regulatory requirements, demanding that certain decisions cannot be changed while the car is running, i.e. the software is being executed. This is somewhat contradictory with the wish to bind the variations as late as possible during the production or even the delivery of the car , , since this would reduce logistic efforts (just one programmed hardware unit for a set of supported engine types vs. differently programmed hardware units for each engine type).

4 Towards a Binding Time Concept for Simulink Models

A flexible selection of a binding time for individual variation points adds another dimension of decision since for each variation point there could be one or more binding times at which the decision can be made. The decision, which binding times to offer should be independent from the functional variability in the problem domain. With traditional software approaches based on manually written code, this introduces a huge challenge, since code supporting different binding times will look different. This becomes evident in scenarios where organizations are forced to provide the same functionality with different binding times. An example is the concept of pre-compile time configured components vs. post-build configured components, in C. To provide support for the first binding time (basically compile time) #define/#ifdef preprocessor statements are used.

For post build configuration normal if statements are used. Due to the different semantics of these two concepts more or less separate implementations are necessary. In model-based development such problems can be solved much more efficiently with an integrated binding time model, since the code generator can take the requested binding time into account when generating code from models.

4.1 Influence of Code Generation on Variability in Simulink Models

The challenge of introducing binding time support in Simulink results from the original purpose of Simulink as a simulation tool, not intended for generating code. With respect to the functionality of a (simulated) system, there is no need for anything but run-time binding, since this is the most flexible approach and hardware resources such as memory and processing power are readily available. The situation is different for code generation when it comes to production code generation, where the memory and also processing power on the embedded system are a very limited resource. The selection of appropriate variant blocks and their configuration as well as the selection and configuration of the used code generator have a significant impact on the code generated from variant-rich Simulink-models and, thus, on the binding time of variation points. For example variability represented by Configurable Subsystems is controlled by a local parameter of the *Configurable Subsystem*-block and already resolved on model level, i.e. before code generation. Unselected subsystems are removed before code generation. [5] refer to this as preprocessing of the model . The advantage of this is that a concrete model variant can be expressed graphical using the normal modeling language.

To provide later binding times, not only the chosen model elements play a role, but also how the code generator transforms model elements to source code. Kalix et. al. investigated in [10] the influence of Simulink and Stateflow model elements on code generated by the Real Time Workshop Embedded Coder and TargetLink, two widely used code generators for Simulink and Stateflow. Their idea was to remove variability at code generation time by relying on the optimizations provided by the code generators. These optimizations effectively remove irrelevant parts of the models from the generated source code. Their conclusion was that at the present time the result was a collection of workarounds. [10]. The result heavily relies on the cleverness of the code generator. The following paragraph investigates the capabilities of TargetLink available for representation of earlier than run-time binding.

TargetLink, itself, offers an own set of blocks to be used inside Simulink (referred to as TargetLink blocks). Most of these blocks have their equivalent in Simulink, but extend the equivalent Simulink- blocks by additional properties and behavior regarding code generation and optimization. Analogous to Simulink, TargetLink does not have explicit concepts for managing variability.

During code generation variability on model-level is mapped to data and control structures in the target source code language. In case of TargetLink the language is C and generated constructs include e.g. #define/#ifdef and if-else

statements. The removal of variability, i.e. the binding, is done by assigning values to variables. The declaration of these variables decides at which point in time the variability is removed. TargetLink provides so called *Variable Classes* for this purpose. Variable classes define the appearance of a variable in the generated code. These variables represent outputs, states, and parameters of a block. Accordingly, Variable Classes has to be specified in a TargetLink block dialog, e.g. for the output of a *Constant*-block or the threshold of a *Switch*-block. Each Variable Class has a set of properties, e.g. related to optimization, where the variables of this class are stored, whether their values could be changed during run-time or not and so forth. TargetLink provides a set of predefined classes, but permits definition of new classes as well. Depending on the chosen Variable Class of a model element, this model element is represented e.g. as a constant or as a global variable in the generated source code. In combination with the optimization of the code generator some blocks will not be present at all in the generated code. Fig. 3 shows this for a *Constant*-block controlling a *Switch*-block. In the context of variability the *Constant*-block represents the control block and the *Switch*-block represents the resolution block.

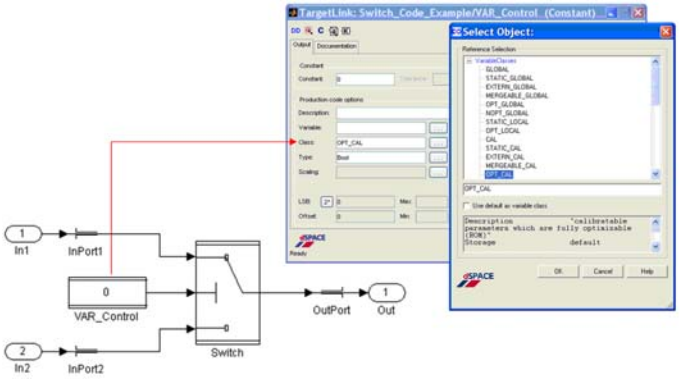


Fig. 3. Signal Routing based on *Constant*- and *Switch*-block

Depending on the selection of the Variable Class for the TargetLink *Constant*-block (in this case `OPT_CAL` or `OPT_LOCAL`), the *Constant*- and *Switch*-block are present in the generated code or not. Fig. 4 shows the first case, Fig. 5 the later case. Variable Classes with the prefix OPT are predefined classes which allow the code generator to optimize the generated code, i.e., if possible to remove variables out of the source code.

Setting the Variable Class to `OPT_CAL` tells the generator to declare the variable `Sa1_VAR_Control`, generated out of the TargetLink Constant-Block block `VAR_Control`, as `const volatile`. This makes the variable unchangeable for the generated program but the value might be changed for instance by an interrupt service routine, e.g. for calibration of certain data parameters in the vehicle. The

```
#define OPT_CAL const volatile

OPT_CAL Bool Sa1_VAR_Control = 0;

Int16 Sa1_InPort1;
Int16 Sa1_InPort2;
Int16 Sa1_OutPort;

Void Switch_Code_Example(Void)
{
   /* SwitchSwitch_Code_Example/Switch
      Switch_Code_Example/Switch: Omitted comparison with constant. */
   if (Sa1_VAR_Control) {
      /* # combined # TargetLink outport: Switch_Code_Example/OutPort */
      Sa1_OutPort = Sa1_InPort1;
   }
   else {
      /* # combined # TargetLink outport: Switch_Code_Example/OutPort */
      Sa1_OutPort = Sa1_InPort2;
   }
}
```

Fig. 4. Extract from source code generated for a *Switch*-block using Variable Class `OPT_CAL`

```
Int16 Sa1_InPort2;
Int16 Sa1_OutPort;

Void Switch_Code_Example(Void)
{
   /* # combined # TargetLink outport: Switch_Code_Example/OutPort */
   Sa1_OutPort = Sa1_InPort2;
}
```

Fig. 5. Extract from source code generated for a *Switch*-block using Variable Class `OPT_LOCAL`

code generator must assume for the generated code that the variable could be changed during run-time. Since this change influences the signal routing of the *Switch*-block, an if-else construct must be generated for the block.

However, if the `OPT_LOCAL` class is used, the code generator assumes, that the variable will not be changeable by the program (as before), but also knows that it will not be referenced from code not generated from this model. Thus, the code generator can generate code, which uses the knowledge about the setting of the Constant-parameter and uses the right input signal for the out-port (see Fig. 5). There will be no change in the routing during the run-time of the program.

4.2 Specification of Binding Times for Variation Points in Simulink-Models

The following concept is based on the assumption that variability not resolved on model level is mapped to variables in the source code by the code generator. These variables are either used as data assignments to adapt behavior of an algorithm or to change the control flow in the application. In case of blocks directly provided by TargetLink it is possible to set the Variable Class declaration of blocks according to the desired binding time explicitly.

From Simulink blocks the code generator uses the context it sees at the time of code generation to decide how to generate optimal code. These Simulink blocks contain either additional TargetLink blocks (e.g. in Fig. 3 a TargetLink

Constant- and *Switch*-block), from which code is generated regarding the configuration of their associated Variable Classes, or they are explicitly controlled by TargetLink blocks. This means that for every mechanism, which removes variability in Simulink, an appropriate TargetLink block is available, which can be used to specify a binding time.

In order to allow modeling of binding times an additional mask parameter `VarBinding` is introduced to complement the `VarInfo` parameter for all blocks where binding times shall be specifiable. In the context of code generation, this parameter is especially relevant for TargetLink blocks, which express variability in the Simulink model. The content of the parameter `VarBinding` is represented in form of a structure.

As shown in the example in Fig. 6, besides the binding time specification itself (here: *CompileTime*), some additional information is stored, e.g. the model elements of the specific TargetLink block, which have Variable Classes assigned. Another important information is whether the binding time might be changed during model configuration (`Alterable` set to `true`). For blocks such as the *Configurable Subsystem*-block which only provides binding at *ModelConfigurationTime*, the value `Alterable` is set to `false`.

```
VarBinding= {'Class', 'output.class', ...,
             'BindingTime', 'CompileTime',
             'Alterable', 'true'}
```

Fig. 6. Example for the data store in a VarBinding parameter

In order to map the available binding times (e.g. *ModelConfigurationTime, CompileTime, LoadTime,* or *RunTime*) onto the expected code representation, Variable Classes, representing the equivalent binding time, has to be defined and configured. As part of the model configuration, the setting for binding time as defined in the `VarBinding` parameter is mapped to the equivalent Variable Class. Part of this process is also validation of the provided information against consistency rules, e.g. that all Control blocks referencing the same VariantParameter may only control resolution blocks which allow the selected binding time.

5 Application of the Binding Time Concept as Part of the Simulink Configurator

The approach described in [6] permits systematic modeling, management and configuration of variability in variant-rich Simulink models. The extension for handling binding time described in this paper provides improved optimization for resource usage and performance of code generated for configured model variants. The original implementation of the Simulink Configurator based on MATLAB 7.1, TargetLink 2.1 and pure::variants 2.4 was extended to support the presented binding time concepts. The Simulink Configurator consist of two main parts, a set of blocks for modeling variability in Simulink, a set of dialogs integrated

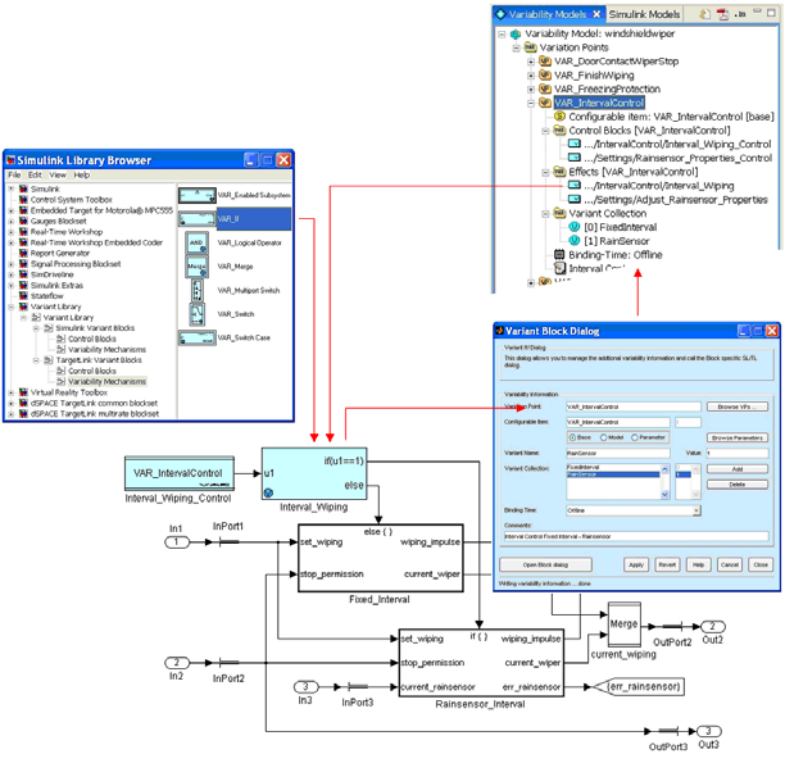


Fig. 7. Design of variable functions in Simulink

into Simulink and a configuration application including feature modeling support based on pure::variants. Fig. 7 shows the basic elements of this tool chain including the support for binding time specification. The workflow is as follows: The model developer selects variant blocks from the library (Fig 7. upper left corner) and inserts those blocks into a Simulink model (Fig 7. lower left corner). These blocks are Simulink- and TargetLink blocks with the additional property and behavior for modeling variability. Mainly, these blocks contain the two mask parameters `VarInfo` and `VarBinding`, with their associated information. Afterwards the model developer links the variant blocks to variation points. The definition of new variation points is done using a variant block dialog inside Simulink (Fig 7. lower right corner). Here the binding time support is just a field allowing the specification of a default binding time. The permitted values are Offline (corresponding to *CompileTime*), StartUp and Online (aka *Runtime*). The integration of binding time concepts into the existing Simulink Configurator tool chain was straightforward, since it basically links already existing functionalities together.

The pure::variants based configuration application reads the information stored in Simulink (Fig 7. upper right corner) to permit the linkage of these variation points to features and to describe dependencies between variation points. The model configuration including the specification of variant specific binding times for variation points itself is done in this application. The validity of the selected feature combination is checked and mapped to the corresponding setting for VariantParameters in Simulink and applied to the Simulink model in real-time.

6 Summary

The presented concept for binding time specification extends the concepts for formal description, and systematic modeling, and configuration of variant rich systems based on Simulink. The implementation of these concepts in the Simulink Configurator is based on MATLAB, TargetLink and pure::variants functionality. The Simulink- configurator proved to work on real-world problems, since the tool chain is being used in a project for production code development for Mercedes passenger cars.

With the addition of binding time support the Simulink-configurator is able to extend the reach of model reuse beyond individual project. The easily changeable binding time permits flexible configurations for developing and testing of systems based on function-rich models, the scaling down to individual variant instances, and for production code, meeting the resource constraints of final product hardware. While the implementation of the binding time concept in its current form is based on a specific code generator (TargetLink), on variability level it can be ported to other code generators as well since most of the information is stored in an abstract form only and later mapped internal to the concrete mechanism provided by TargetLink. With a standardization of variability information within Simulink, applicable to all blocks (not just the ones from one code generator), it would be possible to flexibly exchange between code generators depending on the use case (e.g. when generating for different target systems or interface standards like AUTOSAR). This is one of the benefits of raising the abstraction level for variability modeling from a rather informal level to a systematic and formal level providing clear separation of concerns.

There are also open issues left. While code generators provide the necessary functions to simulate a binding time concept it would be a much more reliable approach if code generators would include a direct and simple way to control the code generation in this respect. Another issue for further research is the efficient mapping of binding times to appropriate code patterns, which provide traceability and optimal performance.

References

1. AUTOSAR (2009), http://www.autosar.org
2. Beuche, D.: Composition and Construction of Embedded Software Families. Dissertation, Universität Magdeburg (2003)

3. Bayer, J., Forster, T., Kiebusch, S., Lehner, T., Ocampo, A., Weiland, J.: Feature- und Entscheidungsmodell-basierte Varianteninstanziierung im PESOA-Prozess. (engl.: Feature- and Decision-Model-based Variant Instantiation within the PESOA-process). PESOA-Report Nr. 21/2005 (2005)
4. Czarnecki, K., Eisenecker, U.W.: Generative Programming– Methods, Tools, and Applications. Addison-Wesley, Reading (2000)
5. Creutzburg, U., Kalix, E.: Process Integration of Model-Based Design and Production-Code Generation in the Multi-User / Multi-Project Development Environment at Continental Teves– Part 2. In: Proceedings of the Int. Automotive Conference (2004)
6. Dziobek, C., Loew, J., Przystas, W., Weiland, J.: Von Vielfalt und Variabilität – Hand-habung von Funktionsvarianten in Simulink-Modellen. (engl.: Model Diversity and Variability - Handling of Functional Variants in Simulink-Models). Elektronik automotive (February 2008)
7. Fritsch, C., Lehn, A., Strohm, T.: Evaluating Variability Implementation Mechanisms. In: Proceedings of the 2nd Int. Workshop on Product Line Engineering – The Early Steps (PLEES 2002), Seattle, USA (2002)
8. Geyer, L.: Variabilitätsmanagement in Produktfamilien (engl.: Variability Management in Product Families). Dissertation, Universität Kaiserslautern (2003)
9. Jacobson, I., Griss, M., Jonsson, P.: Software Reuse– Architecture, Process, and Organization for Business Success. Addison-Wesley, Reading (1997)
10. Kalix, E., Bunzel, S., Judaschke, U.: Variant Coding in Model-Based Design. In: 9th World Multiconference on Systemics, Cybernetics, and Informatics (WMSCI), Orlando, USA (2005)
11. Kang, K.C., Cohen, S.G., Hess, J.A., Nowak, W.E., Peterson, A.S.: Feature Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Carnegie Mellon University, Pittsburgh, PA (1990)
12. Lee, K., Kang, K.C., Lee, J.: Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In: Gacek, C. (ed.) ICSR 2002. LNCS, vol. 2319, p. 62. Springer, Heidelberg (2002)
13. pure-systems, pure:variants Eclipse Plugin User Guide (2008)
14. Schäuffele, J., Zurawka, T.: Automotive Software Engineering – Grundlagen, Prozesse, Methoden und Werkzeuge (engl.: Automotive Software Engineering – Foundations, Processes, Methods, and Tools) Vieweg, Wiesbaden (July 2003)
15. dSpace GmbH: TargetLink Advanced Practices Guide – for TargetLink 2.2. dSpace, Paderborn (2006)

Experiences of Developing a Network Modeling Tool Using the Eclipse Environment

Andy Evans[1], Miguel A. Fernández[2], and Parastoo Mohagheghi[3]

[1] Xactium Ltd. UK, Sheffield - UK
andy.evans@xactium.com
[2] Telefónica Research & Development – Valladolid, Spain
mafg@tid.es
[3] SINTEF, P.O. Box 124- Blindern, N-0314 Oslo, Norway
parastoo.mohagheghi@sintef.no

Abstract. Domain-specific modeling solutions have been promoted for some time in order to improve the productivity of software developers by providing them with modeling environments that are easier to learn, integrate best solutions and provide the possibility to automate software development by generating code from models. This paper presents experiences of developing a network modeling tool in Telefónica using Eclipse GMF. A metamodel based on Common Information Model was used in this development. While we experienced benefits in terms of better usability by domain experts, we also faced challenges such as the high level of expertise required to develop a good enough language and tool, the shortcomings of the tools in providing support for modeling at different abstraction levels, and the difficulties in updating the modeling tool with changes in the metamodel. These challenges must be overcome before the tool can be a part of our development environment.

Keywords: domain-specific modeling, language design, metamodeling, Eclipse.

1 Introduction

Throughout the history of software, developers have always sought to increase their productivity by improving abstraction. Domain-Specific Modeling (DSM) raises the abstraction level by offering the possibility to specify solutions directly using problem domain concepts [8]. Other artifacts are then generated from these high-level specifications. A modeling environment that fits to the concepts of the domain and the problem in hand is expected to be easier to be used by domain experts. A domain here is an area of interest; either a horizontal functional domain (such as user interface or persistency) or a vertical business domain such as telecommunications or retail. In this paper, network modeling in telecommunication is the domain of interest.

DSML tools are visual modeling tools based on some Domain-Specific Modeling Language (DMSL). DSMLs are promoted by some as the next big thing after General-Purpose Languages (GPLs) and there are reports of successful application in industry such as in Motorola [2] and examples presented in [8]. However, reports of experience are still few and far between, as a review of literature on industry experience with

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, pp. 301–312, 2009.

Model-Driven Engineering (MDE) has confirmed [1]. There may be several reasons for that; firstly the development of DSM solutions does not have a long history; and secondly DSM solutions are valuable assets that give companies competitive advantage so experience reports may be kept private and not published.

The European research project MODELPLEX (MODELing solution for comPLEX software systems)[1] aims at evolving MDE tools and technologies to be applicable for developing complex software systems and evaluating them in the context of four, very different, industrial scenarios. As an industry case provider in MODELPLEX, Telefónica has participated in specifying requirements regarding tools and technologies, customizing the solutions, evaluating them and providing feedback to both tool providers and industry interested in applying these solutions. One of the areas of research has been the development of a DSML for network modeling based on the CIM (Common Information Model) metamodel defined by the DMTF [5].

Taking into account the high cost of developing a DSML, a company needs to make a serious evaluation of the Return On Investment (ROI), balancing the expected benefits and productivity improvement against the cost of development and future maintenance of the tools before moving to a new development environment. The purpose of this paper is to report on our experiences of developing a DSML based around the aforementioned Telefónica case using the widely used Eclipse GMF tool development framework. We also identify useful criteria for evaluating the resulting DSML tool which might be part of a ROI analysis when a solution is developed.

The remainder of the paper is organized as follows. Requirements of the DSML and the criteria for evaluating it are presented in Section 2. Section 3 presents the steps of the development while Sections 4 and 5 present our experience with the environment used for developing the DSM tool and the DSM tool itself. Section 6 presents the challenges for developing a DSML based on our experience (Xactium has long experience with language-driven development[2]). Finally Section 7 presents a conclusion and discussion of future work.

2 Requirements of DSML and Criteria for Evaluation

2.1 Requirements

As mentioned in the introduction, the purpose of this report is to discuss the development of a specific DSML for network service modeling. The key driver for this DSML is the wide recognition that it is becoming increasingly difficult to manage the complexity and size of modern telecom networks [4]. To address these challenges, it was proposed that a DSML be developed to enable the modeling of complex networks delivering services to private subscribers via a range of different devices. This approach would provide Telefónica with a modeling language to capture the key features of these networks and services at a level of abstraction that enables the management of complex networks more efficient and more manageable. Some specific areas of requirement were identified as follows.

[1] http://www.modelplex-ist.org/

[2] The idea of language-driven development is providing developers with an integrated collection of semantically rich languages that specifically target their development needs.

By Telefónica's requirement, the Network DSML had to include, but was not necessarily limited to, the following concepts; a) network topology (e.g., sub-network addressing); b) device properties (e.g., interfaces, firmware version, etc.); and c) types of network traffic and service features (e.g., protocols, port ranges, QoS, etc.) In addition, a key requirement of the Network DSML tool was to allow modeling at different levels of abstraction, at least the following: a) device, showing internal device details, e.g. network interfaces; b) network topology, showing how devices connect to each other; and c) service, showing higher-level interactions and roles of whole sub-networks in the deployment of a service.

From these models, a wide range of artifacts could be generated, the first ones on the list being device configuration specifications to be fed, in the appropriate format, to Telefónica's OSS subsystems in charge of device configuration and monitoring.

Rather than develop these concepts from scratch, it was proposed by Telefónica that the Common Information Model (CIM) [5] would provide a useful starting point. This model provides many concepts useful to the modeling of networks and devices. A significant part of this model was identified as being relevant to the requirements of the project, and with some additional changes, this became the core model around which the DSML was based. CIM was also relevant as it is the underlying model in many COTS products dealing with management and instrumentation of network equipment, some of which are part of current Telefónica's OSS.

Finally, there were a number of generic features of the DSML tool, which were required in order to meet the needs of users of the tool. These included: a) a visual, user-friendly interface; b) scalability – enabling thousands of model elements to be managed; c) interoperability with other tools and standards; d) flexibility – enabling the rapid adaptation of the tool to support new abstractions (preferably done by the engineers themselves); and e) support for model validation and checking.

2.2 A Framework for Evaluating the DSML

Industrial participants in the MODELPLEX project have defined a set of research questions for evaluating the solutions. There, Telefónica has stated that, "We expect to develop a DSML that helps us create these models in a way that proves convenient for business experts with no technical background in modeling". The questions are:

> *"Can a telecommunication business expert model services by means of a DSML? How valid are the models in terms of completeness and usefulness for the generation of other artifacts?"*

As part of the MODELPLEX project we have also performed a state of the art analysis regarding evaluation of languages (in this case a DSML) and identified the stakeholders and their points of interest, as depicted in Fig. 1:

- *Language Engineers (LE)* are those developing the language. They evaluate a language based on whether the language features are easy to implement or whether it is easy to develop compilers or generators.
- *Language Users (LU)* are personnel using the DSML for modeling, which are interested in ease of use, increased productivity, etc.

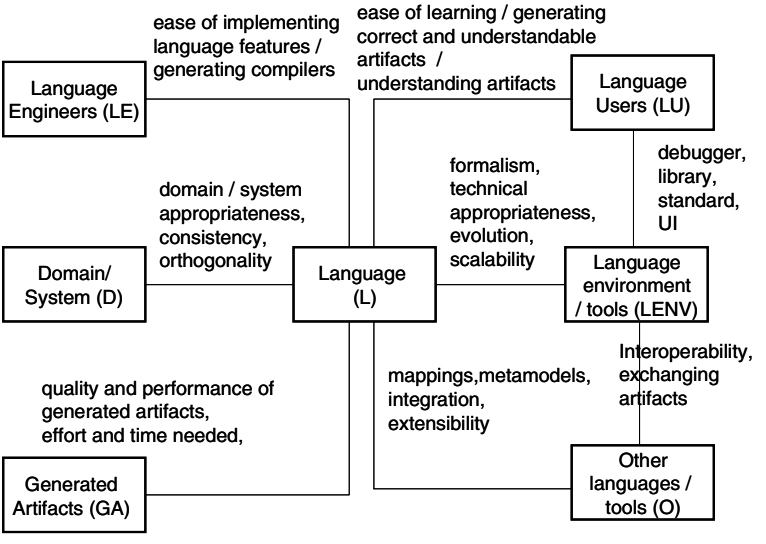


Fig. 1. Evaluating a language from multiple views

- *Language Environment (LENV)* is the tool (including editors and transformations) which is developed around the DSML. There are requirements such as whether the language is formal, evolvable or scalable. Besides, including debuggers and libraries, being standards-based and compatible with other tools, and having a pleasant User Interface (UI) all increase the value of the LENV.
- *Domain/System (D)* is the domain of interest. A language should be appropriate for the domain, the concepts should be consistent, etc.
- *Other languages / tools (O)* cover requirements for interoperability, mappings between languages, building future extensions, etc.
- *Generated artifacts (GA)* may have requirements regarding quality, performance, and effort or time needed for generation.

While the framework shown in Fig. 1 is developed with languages in mind, in MODELPLEX we also take advantage of an extended version of the Technology Acceptance Model (TAM) for evaluating tools and technologies. The original TAM, by Davies, is widely referenced [9] and used in information science research. It explains users' intention to use a new system through two beliefs, *perceived usefulness* and *perceived ease of use*. There are several extensions to TAM and we use the model described in [11] for evaluation of the DSML (and the base DSL) as depicted in Fig. 2 where we have also inserted requirements identified in the previous section and the stakeholders as defined above. We define these factors as:

- *Perceived Usefulness (PU)* is the degree to which a person believes that using a particular method or tool will enhance their job performance.
- *Perceived Ease of use (PE)* refers to the degree to which a person believes that using a particular method or tool would be free of effort.

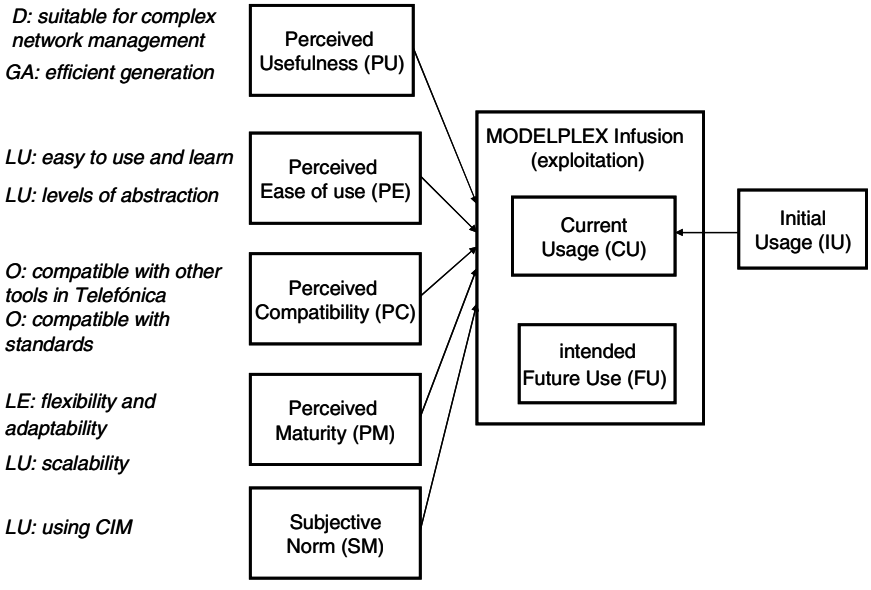


Fig. 2. The model used for evaluating DSM

- *Perceived Compatibility (PC)* is the degree to which an innovation is perceived as being consistent with existing practices, standards and tools, and the past experience of potential adopters.
- *Perceived tool Maturity (PM)* is the degree to which tools are perceived as mature and suitable for the tasks in hand.
- *Subjective Norm (SM)* is the degree to which software developers think that others who are important to them think that they should use that particular method or tool.

For evaluating each factor, a set of questions is defined from the stakeholders' viewpoint and their interest in the developed language in the context of the company. These questions are listed in Section 5 together with the answers. Originally, the evaluation was intended as a questionnaire. However, we performed a qualitative analysis by three engineers form the MODELPLEX research team at Telefónica instead, since the DSML was not used by a significant number of developers then.

3 Design and Implementation of the DSM

For this project we used the Eclipse Graphical Modeling Framework (GMF) [6] plug-in to develop the DSML. Our reasons for choosing GMF were its relative popularity and maturity and the fact that it is open source and based on Eclipse (one of the key platforms mandated by MODELPLEX by virtue of its interoperability and openness). GMF provides the ability to develop a working tool for the graphical representation of data, based entirely on an EMF (Eclipse Modeling Framework) [7] model and complying to all the relationships and constraints specified in that model.

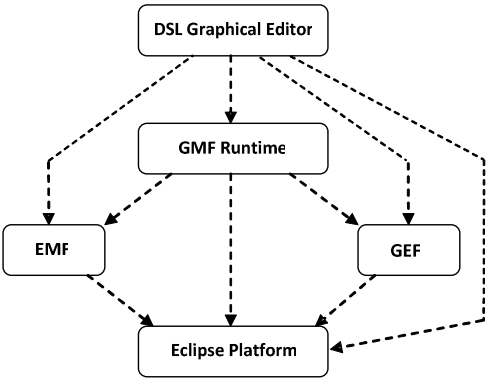


Fig. 3. Relations between the developed DSML editor and the Eclipse components

The process of creating a graphical model editor in GMF requires the use of other Eclipse components such as the Eclipse Graphical Editing Framework (GEF). So when trying to understand the relationship between GMF and EMF it is also important to take into account their relationship to the Eclipse Platform, on which they are built. Fig. 3 is a representation of that relationship. As we can see, a GMF-based DSML graphical editor depends on the GMF runtime component but also makes direct use of EMF, GEF and the Eclipse platform.

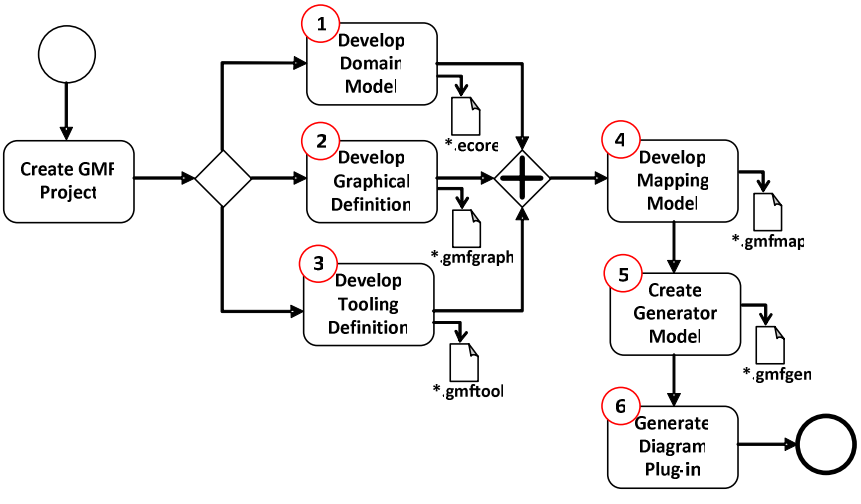


Fig. 4. Steps in creating the DSML editor

The first step in the development of this tool was the creation of an EMF model (or metamodel). From this model specification, a set of Java classes are produced which can later be used as the foundation for our tool. The CIM metamodel was initially transformed into Ecore by Xactium and then modified by Telefónica to the needs of their specific domain. A significant challenge of this stage of development is the size

of the CIM metamodel which contains over 1500 concepts, hence a reduced subset was used in the first implementations, consisting of more than 200 concepts.

The next step was to take our (CIM) metamodel and begin the development of the Eclipse graphical modeling editor. The basic components of the GMF model we developed are depicted in Fig. 4 and described below:

1. The domain model defines the non-graphical information managed by the editor (this can be generated directly from our EMF model).
2. The graphical definition model contains information related to the graphical elements that will appear in a GEF-based runtime, but has no direct connection to the domain models for which they provide representation and editing.
3. An optional tooling definition model is used to design the palette and other periphery (menus, toolbars, etc).
4. The diagram mapping model defines mappings/relationships between domain model elements and graphical elements.
5. Once the appropriate mappings are defined, GMF can produce a generator model from which the code could be generated.

This process was repeated over a number of iterations, to produce a fully working tool. An example of the tool in action is shown in Fig. 5.

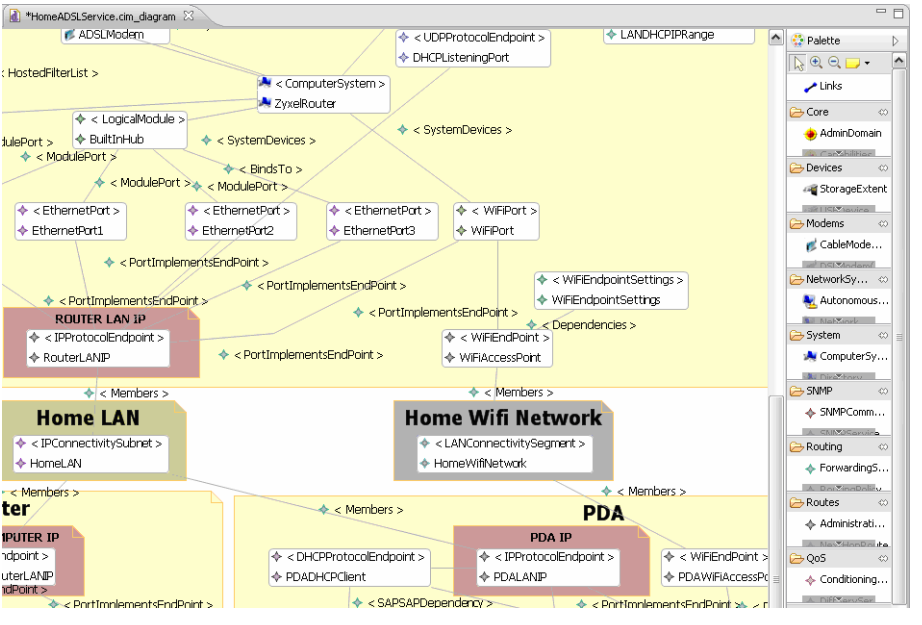


Fig. 5. A fragment of a model in the network DSML tool

4 Experiences with GMF/EMF

One of the most challenging aspects of this DSML was the large number of modeling abstractions and relationships in the CIM model. As a result, it was decided to develop

the tool as a graph editor, with each node in the graph representing a class instance and each edge in the graph representing a relationship between classes. As an example, a device would be represented as a node, labeled "Device", and its relationships (both direct and inherited from parent classes) as edges.

When developing the mapping model it was apparent that there was an issue in managing containment relationships. In GMF, containment relationships by default map to containment structures in the diagram model. However, given the large number of containment relationships, it was not practical to make special cases for each diagram mapping as many of these relationships would need to be represented in different ways, e.g. as a sub-node or sub-diagram. Moreover, for many concepts, it did not make sense to treat their diagram representation as a container in any case.

To address this issue we adopted a GMF development technique called phantom (or shadow) nodes. These are simply nodes without their containment feature set. The use of this technique was necessary but we knew this would cause problems at a later stage since, when using it, the top-level nodes should still have a containment relationship to the canvas and this was not acceptable in our model. This issue was solved by making changes to the generated code, which involved editing the create function for each of the nodes on the diagram to give them a containment relationship to the canvas when they are drawn on the diagram. Due to the use of shadow nodes, all nodes could be given a containment relationship of a different type by the user.

Another challenge was that of making the tool as usable as possible, which involved changing the tooling definition. Again, the large number of abstractions was an issue which had to be addressed as simply as possible. To do this, we grouped classes into groups and then also grouped diagram components to a tool based on their types; this reduced the number of palette elements and increased usability significantly. We made further changes to the tooling palette as well by changing the *Icons* in the *.edit* file by grouping tools with icons.

When using the editor we discovered that the automatically generated popup menus and connection handles were more of a hindrance than help, due to the complexity of the model. The popup menu was overly large due to the number of creation tools for each component in the diagram, also connection handles produced an incorrect output due to the scope of the model that the tool is built on. To solve this issue we removed the popup menus and connection handles by adding some code to the *diagramEditPart* of each diagram element (including the nodes and the canvas).

One of the key requirements of the DSML tool was that it provided sufficient flexibility to enable rapid changes to the metamodel, thus enabling the tool to adapt to changing modeling requirements. Unfortunately, this was not supported well by GMF. Even small changes to the model require repeating the code generation steps and there was always significant risk that errors would creep into the generated code. Furthermore, any changes required someone with strong technical expertise in GMF.

Another important aspect missing from GMF is the provision of a facility to encapsulate levels of abstraction through the use of components or product line concepts. For example, in the case of a 'router' concept, it could be thought of as being composed of a collection of more primitive elements. Ideally, the tool needs to provide an easy to use mechanism for creating abstractions as patterns of more fundamental elements. Again, the facility should not be reliant on the re-generation of the tool, but should provide the ability to create new abstractions dynamically.

5 User Experience with the DSML Tool

Here we present the opinion of the users of the tool to a set of questions related to the TAM evaluation criteria presented in Section 2.2.

Table 1. Results of the DSML tool evaluation

Perceived Usefulness	
1. Is the CIM metamodel suitable for modeling network management in Telefónica?	Yes, they are suitable for this purpose but need constant revision and extension to keep up with the evolution of the domain and the standard of reference (CIM).
2. Do the DSML and the artifact generation capabilities affect quality, performance and productivity of the work?	Yes, the DSML has the potential to improve productivity and quality but additional work and training, as well as other tools like model transformation languages, etc., are needed to achieve those objectives.
Perceived Ease of use	
1. Is the DSML tool easy to use? Is the UI acceptable?	Not enough, largely due to the sheer size of the metamodel which resulted in having to add a large number of connection and node tools.
2. How can the abstraction layers improve models and their understanding?	Abstraction layers are necessary in cases such as this and can improve greatly the understanding and usefulness of the models. The problem is that abstraction layers are not supported in GMF and, even with the addition of a model composition framework, the level of integration achieved was not sufficient.
Perceived Compatibility	
1. Is the DSML compatible with the standards?	Yes, using CIM provides such compatibility but brings problems due to its size.
2. Is the DSML compatible with other tools?	Many tools used in the network management domain are based on CIM, but as the DSML transforms the CIM metamodel into EMF, this leads to compatibility issues with CIM-based off-the-shelf products that need to be resolved.
Perceived Maturity	
1. Is the solution scalable?	GMF does not scale well because of some shortcomings in the implementation that have been already discussed.
2. Is the solution flexible?	The same applies to flexibility. A more dynamic, metamodel-driven tool generation approach is needed.
Subjective Norm	
1. How would others judge our use of the DSML? Do we think that it improves our reputation and image as innovative?	Yes, the image and reputation of innovation can be greatly improved by the use of tools and approaches such as the one presented herein.

6 Challenges for DSML Technologies

We recognized two main challenges (or shall we say obstacles) during developing the DSML solutions: a) developing a DSML in an environment such as Eclipse requires high language expertise and tool expertise, which make developing DSMLs out of

reach of domain experts with some IT expertise; and b) the resulting DSML is not changeable or flexible enough. We describe these in more detail below with outline of solutions.

a) More user-centric development environment

Understand that DSMLs are a business solution, not a technical one. While the community of users of Eclipse and GMF is growing, and there are many examples of DSML tools that have been developed using the technology, there is too much emphasis on GMF as a technical solution rather than a business solution. This is widely reflected by the large number of academic conferences on the subject of DSMLs, the relative lack of involvement of business users in the development and use of DSMLs, and the fact that the development of DSL technologies is largely being driven by programming experts and IT groups. Until this is addressed, DSML tools, certainly in the case of those built using Eclipse, will not achieve critical mass for business users, and will largely remain the domain of academic interest and IT research departments. Such lack of critical mass poses a significant issue to large companies like Telefónica, who have to take into account the costs of supporting and maintaining non-mainstream technologies in the long term.

Enable DSMLs to be developed by end users. A problem encountered with GMF was the significant technical expertise required to develop DSML tools, even simple ones. This is a significant challenge for users who are not technically minded as, in practice, they will have the best understanding of their domain. It seems particularly strange that although a key objective of DSMLs is to provide a more targeted and flexible domain solution, the ability to create DSMLs is only accessible to experienced programmers.

Address abstraction zoom-in and zoom-out, to help simplifying the models and allow reuse. During this project we have identified a specific example of flexibility which is an important requirement for modeling (certainly in the telecom domain). Because many telecom systems are built up of components, which are themselves composed of more granular components, there is a requirement to be able to create pre-defined combinations of components which can be combined together in new ways. While it is possible to create component models in UML, there is an advantage of being able to generically combine different DSML concepts into reusable components. Whilst this capability is not available 'out of the box' with GMF and other DSML tools, we are examining how the Eclipse Reuseware [10] initiative might address this.

Support multi-user, multi-tenancy DSML tools. Once a DSML has been developed and is in use, a significant challenge is to scale its use to multiple users. While models created in Eclipse can be exported to others users, there is no simple mechanism for ensuring changes to models by one user are kept in sync with changes made by other users. Data can soon become out of step, and the effort to resolve changes becomes prohibitive for successful commercial use. In other areas of business software, for example, database applications, such issues have been recognized and addressed through multi-user support, while multi-tenancy solutions address critical issues of managing and upgrading data when underlying changes to the database are made. This challenge is not specific to DSMLs but to modeling tools in general.

b) Need for flexible solutions:

Enable non-programming customization and adaptability. As identified above, a problem encountered in the development and use of the CIM DSML tool was the problem of adapting it to new requirements. These adaptations were primarily around the changes to the underlying metamodel, including changes to properties, relationships and the addition of new domain concepts. However, they could also include changes to diagrammatical representations, and also hiding and showing of user relevant information, for example fields. GMF completely failed in this regard due to the fact that any changes (whether simple or complex) required re-generation of the code.

Support dynamic management of data and metadata. Another key requirement for adaptability is the ability to accommodate changes to the metamodel without making existing model data redundant. While this was not tested fully, in a number of cases, models became corrupted due to changes in the DSML tool and could not be reused without significant modification of the underlying XML file. The alternative, of creating a model-to-model mapping to transform the data would again necessitate significant programming expertise. We will investigate other solutions to this problem in the MODELPLEX project but we fear it is a complex issue to solve.

7 Conclusions and Future Work

The purpose of this paper is to report on our experiences with a widely used, open source framework for building DSML tools- the Eclipse GMF framework. While we do not wish to claim that the challenges identified in its use are applicable to all DSML technologies, we do believe the challenges we have identified are an important consideration when evaluating and developing DSML technologies (particularly when assessing ROI). One particular issue with regard to GMF, was the need for a more user-centric tool development process that would enable end users and domain experts to participate more fully with the tool design process. A second issue is the need to encapsulate levels of abstraction, again provided in a user friendly way. We view both of these challenges as an essential requirement for the wider commercial uptake of DSML technologies. Whilst there may be existing technologies available which overcome these challenges, we believe our focus on GMF is important as it is one of the leading technologies in the marketplace.

We also believe that a more flexible approach to DSML development is required, which would support the dynamic creation of DSMLs as opposed to the generative approach taken by GMF.

Solving the above issues would enable DSML tools to be created by domain experts rather than software developers, thus providing a more interactive and user-centric approach to DSML development. Some key features of this approach must be; a) the use of metadata to configure and customize the resulting tools on the fly; b) a user friendly interface for customizing the editors – probably via a DSML tool; c) the ability to easily upgrade model data without the need for complex transformations; d) the ability to easily customize the resulting tool, for example, in terms of look and feel, icons, etc; and finally e) the ability to represent patterns of concepts of higher level abstractions, which can themselves be reused in the tool.

We are at the moment exploring ways of adapting the existing GMF technologies to provide a DSML tool generation engine. This would provide a way of dynamically creating tools by loading the tool model into the engine (rather than generating the code). Another contribution of the work has been developing a framework for evaluating DSML solutions which will be reused in future work. We hope to report on all this as part of the MODELPLEX project.

Acknowledgments. This work has been done in the MODELPLEX project (IST-FP6-2006 Contract No. 34081), co-funded by the European Commission as part of the 6th Framework Program.

References

1. Mohagheghi, P., Dehlen, V.: Where is the Proof? A Review of Experiences from Applying MDE in Industry. In: Schieferdecker, I., Hartman, A. (eds.) ECMDA-FA 2008. LNCS, vol. 5095, pp. 432–443. Springer, Heidelberg (2008)
2. Baker, P., Loh, P.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. Mohagheghi, P., Fernandez, M., Martell, J.A., Fritzsche, M., Gilani, W.: MDE Adoption in Industry: Challenges and Success Criteria. In: ChaMDE Workshop at MoDELS 2008, To be publised in the Proc. of Workshops at MoDELS 2008 (2008), ftp://ftp.umh.ac.be/pub/ftp_infofs/2008/ChaMDE-report.pdf
4. Wong, D., Ting, C., Yeh, C.: From Network Management to Service Management – A Challenge to Telecom Service Providers. In: Proc. 2nd International Conference on Innovative Computing, Information and Control (ICICIC 2007) (2007)
5. DMTF's Common Information Model Website, http://www.dmtf.org/standards/cim/
6. Eclipse Graphical Modeling Framework (GMF), http://www.eclipse.org/gmf/
7. Eclipse Modeling Framework (EMF), http://www.eclipse.org/emf/
8. Kelly, S., Tolvanen, J.-P.: Domain-Specific Modeling- Enabling Full Code Generation. IEEE Computer Society Publications, Los Alamitos (2008)
9. Davis, F.: Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology. MIS Quarterly 13(3), 318–339
10. Reuseware Composition Framework, http://www.reuseware.org/
11. Dybå, T., Moe, N.B., Mikkelsen, E.M.: An Empirical Investigation on Factors Affecting Software Development Acceptance and Utilization of Electronic Process Guides. In: Proc. 10th International Symposium on Software Metrics (Metrics 2004), pp. 220–231 (2004)

MBT4Chor: A Model-Based Testing Approach for Service Choreographies

Alin Stefanescu[1], Sebastian Wieczorek[1], and Andrei Kirshin[2]

[1] SAP Research, CEC Darmstadt, Bleichstr. 8, 64283 Darmstadt, Germany
{alin.stefanescu,sebastian.wieczorek}@sap.com
[2] IBM Haifa Research Lab, Haifa University, Mount Carmel, 31905 Haifa, Israel
kirshin@il.ibm.com

Abstract. Service choreographies describe the global communication protocols between services and testing these choreographies is an important task in the context of service-oriented architectures (SOA). Formal modeling of service choreographies makes a model-based testing (MBT) approach feasible. In this paper we present an MBT approach for SOA integration testing based on SAP proprietary choreography models called Message Choreography Models (MCM). In our approach, MCMs are translated into executable UML models using Java as action language. These UML models are used by a UML model execution engine developed by IBM for test generation and model debugging. We describe the achievements and challenges of our approach based on first experimental evaluation conducted at SAP.

Keywords: Choreography Modeling, SOA, Service Integration, Model-Based Testing, Model Transformation, Domain Specific Language.

1 Introduction

Enterprise Resource Planning (ERP) software [15] supports business processes for whole companies, with SAP being a leading provider of ERP software. ERP software integrates many organizational parts and functions into one logical software system, posing unique challenges to software development and also testing [12,21]. Recently, service-oriented architecture (SOA) has come to be regarded as the next evolutionary step in coping with the software complexity of ERP systems where monolithic approaches are no longer applicable [23,16]. SAP simplifies the SOA adoption by delivering SOA-enabled software, SOA methodology guidelines, and professional services. Using the SAP approach, independent business components exhibit enterprise services that can be composed individually to implement customized business processes. For service integration to occur on a higher level of abstraction than component development, complex service interactions need to be modeled: hence, choreography languages have emerged. According to the W3C Web Service Glossary [17], "a *choreography* defines the sequence and conditions under which multiple cooperating independent agents exchange messages in order to perform a

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, pp. 313–324, 2009.

task to achieve a goal state". Thus, a choreography model describes the interaction protocol from the perspective of a global observer between a set of loosely coupled components communicating over message channels.

Choreography models play an important role in SOA development and can provide a basis for ensuring quality at several levels. In previous work [19], we defined precise requirements on choreography modeling languages that support the three software quality related use cases of design, verification, and testing. However, we observed that state-of-the-art choreography languages such as WS-CDL [13], Let's Dance [24], and BPMN [3] do not fulfill all these requirements simultaneously, mainly due to the high abstraction level, imprecise semantics, lack of a formal foundation, assumption of ideal channels, lack of termination symbols, etc. Recently SAP Research developed a proprietary choreography modeling language called Message Choreography Modeling (MCM) that satisfies the requirements from [19] (e.g., graphical state-based representation, explicit concurrency, detailed message types, local viewpoints, determinism, a distinction between global and local constraints) and implemented an MCM editor with verification and testing plugins.

In this paper, we describe a model-based testing (MBT) approach called MBT4Chor for service integration based on MCM. The goal is to generate integration tests for message-based communication between business components. We achieve this with two complementary means: (a) by directly implementing graph-coverage algorithms for the choreography models and (b) by a model transformation to executable UML where we can use an IBM research prototype tool for generating random tests.

The contributions of the paper are the following:

1. to present our experience of using MBT on a domain-specific language (DSL),
2. to sketch a transformation from our DSL (MCM) to a general purpose one (UML),
3. to describe a new UML based test generation tool that relies on the UML execution engine presented in [5], and
4. to share the lessons learned and challenges of an MBT approach in an industrial setting of SOA applications.

The paper is structured as follows. Section 2 provides an overview of MCM. Section 3 describes the proposed MBT approach and Sections 4, 5, and 6 present the translation from MCM to UML, the test generation tool for UML and the test concretization at SAP. Sections 7 and 8 provide related work and future activities.

2 Choreography Modeling

The SAP approach to SOA and SAP's internal software development lifecycle includes a rich modeling environment. While external SOA artifacts are based on web service open standards like SOAP, WSDL, and WS-Reliability, internal development is based on various proprietary models for business objects, service components, component interaction and integration scenarios. Recently, SAP Research developed a proprietary domain-specific language for modeling service choreographies called Message Choreography Modeling (MCM) together with a customized Eclipse-based modeling environment.

MCM complements the structural information of the communicating components (e.g., service interface descriptions and message types) with information on the message exchange among them. MCM consist of different model types each defining different aspects of service composition. The remainder of this section informally describes these model types and their relations are informally described.

2.1 Global Choreography Model

The global choreography model (GCM) specifies a high-level view of the conversation between service components. Its purpose is to define every allowed sequence of message exchanges. Similar to an extended finite state machine (EFSM), a GCM consists of transitions with guards and side effects over different datatypes, leading to a possibly infinite state space. In the GCM, transitions are fired only when a global observer is able to detect that a message was consumed by the receiving component (i.e., the message left the communication channel). In contrast to common choreography languages that have their semantics based on send events, we noticed that the receive semantics of GCM provides a better testing approach because message racing can also be captured.

The left side of Figure 1 shows an example of a GCM for two components, a seller and a buyer. The buyer is able to send the messages *Request* and *Cancel* that are then received by the seller, while the seller can send the message *Response,* which is received by the buyer. For simplicity, in this example no guards, side effects or concurrent states are used, although such features are present in the MCM metamodel.

One of the requirements for GCMs is determinism. In the example in Figure 1 this is achieved already because in each state the outgoing transitions have different messages. In scenarios where this is not the case, determinism has to be enforced by assigning mutually exclusive guards.

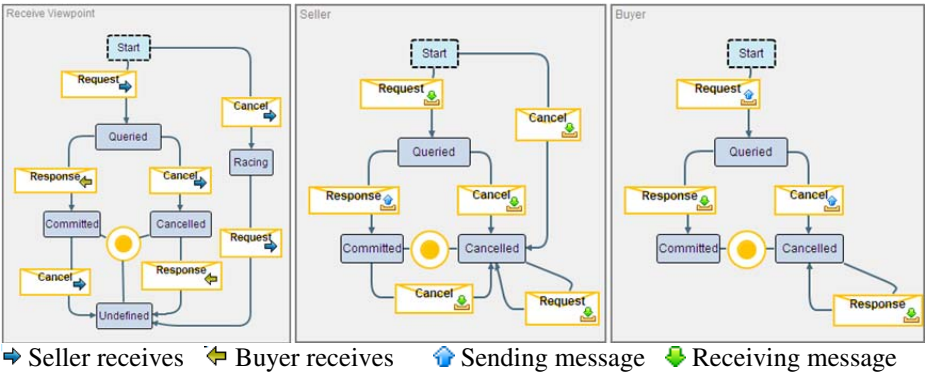


Fig. 1. Global Choreography Model (GCM) on the left and the two corresponding Local Partner Models (LPMs)

Another requirement of GCMs is the marking of initial and target states in which the communication reaches an agreed goal of conversation. The test generators must generate test sequences starting in an initial state and ending in a target state. Note

that in MCM, the target states allow outgoing transition because of business scenarios where one of the communicating partners is allowed to restart a negotiation process infinitely often. This is in contrast to the final states of the UML state machines. In the GCM of Figure 1 the states *Committed*, *Undefined,* and *Cancelled* are target states, while the state *Start* is marked as an initial state.

2.2 Local Partner Model

Local Partner Models (LPMs) specify the communication-relevant behavior for exactly one participating service component. Due to the design process of MCM, each LPM is a structural copy of the GCM with extra constraints on some of the local transitions, usually leading to the affected transitions being removed. For exactly two involved partners, the global message receive events are copied to the receiving components, while the receive events of other partners are transformed into send events.

Figure 1 shows one possible set of LPMs for the GCM. Compliant to the global specification GCM, the LPM of the buyer consists of send events for the *Request* and *Cancel* messages and receive events for the *Response* message (and vice versa for the seller).

As mentioned, some of the events copied from the GCM have been removed in each of the LPMs. Note that despite these local reductions, in our example each global transition can still be reached, that is, all message sequences possible in the GCM can be simulated with local sequences of the sends and receives in the LPMs. These send and receive events take into account the communication channel properties (see below).

2.3 Channel Model

The channel model describes the characteristics of the communication channel on which messages are exchanged between the service components. It determines, for example, whether messages sent by one component preserve their order during transmission. For the channel definition, MCM uses the Web Service Reliable Messaging (WS-RM) standard [18]. In the example in Figure 1, we assume a reliable channel on which each sent message is received exactly once, but which does not necessarily preserve the message order. Therefore, the buyer may send the *Cancel* message only after sending *Request,* while the seller has to be prepared to receive either *Request* or *Cancel* first. Because the GCM describes the order of receive events, it also reflects the possible switching of the *Request* and *Cancel* messages on the channel.

3 Overview of MBT4Chor

In addition to a holistic software design purpose, the development of MCM was driven by the requirements of automatic model verification and test generation. This section gives an overview about the utilization of MCM for testing.

The core idea of model-based testing (MBT) is to use formal specifications for test generation. This implies that tests can only be as precise as the modeled content they use. By design, MCM offers the necessary information to drive the generation of test suites covering the specified interaction protocol. The generated test suites have to be

supplemented with additional information, because even though the local behavior is modeled in the LPMs, triggers for the local message sending events are not specified. This information cannot be easily modeled as it is deeply rooted in the internal behavior of the components. Note that MCM is not suited for the derivation of component tests because apart from the missing triggers mentioned, the behavior modeled in the LPMs focuses on communication only, leaving out internal steps that may happen in between the communication events.

However, using MBT for service integration promises to reduce the manual effort by automatically generating minimal sets of test cases for desired coverage of the choreography model. In [20] we discussed different coverage criteria that can be used to drive service integration testing and how to choose them accordingly, depending on effort and fault assumptions. Among the different possible coverage criteria, we decided to start with the transition coverage of the GCM, which requires the following MCM-specific three-step approach for test generation:

– *Step 1*: A test generator generates a set of globally observable message sequences that cover each transition of the GCM. According to our example from Figure 1, a generated sequence could be <*Seller:Request, Buyer:Response, Seller:Cancel*>. Given the receive semantics of GCM, this reads:

<*Seller receives Request, Buyer receives Response, Seller receives Cancel*>.

– *Step 2*: The local event sequences corresponding to the test cases are computed. This is necessary because the GCM only specifies the order of receive events. Therefore the receive sequences have to be enhanced by their corresponding ***send*** events, taking the LPMs and channel model into account. According to Figure 1, the only possibility of achieving the generated sequence in Step 1 is this:

<***Buyer sends Request***, *Seller receives Request,* ***Seller sends Response***, ***Buyer sends Cancel***, *Buyer receives Response, Seller receives Cancel* >.

This step is MCM-specific and relies on the receive semantics of the choreography model. It can be done automatically without major issues.

– *Step 3*: Generated abstract test cases are translated into executable test suites. This step is semi-automatic. We can automatically generate the concrete test steps as well as the state checks on the local components. However, as triggers are not fully modeled in MCM, this information has to be added manually to the test sequences. The test concretization is further described in Section 6.

Step 1 is concerned with the integration of a test generator. The fact that the MCM editor is based on the Eclipse plugin technology opens up the possibility of integrating multiple test generators. For the moment we have experimented with the following two test generators:

– An SAP in-house solution for test generation – We implemented classical graph-coverage algorithms working directly on the global state space of the GCM. The major disadvantage is that it takes time and effort to reach the state of the art in MBT, so we also used an external tool (below) for that.

– An MBT prototype from IBM Research – The usage of this tool, described in Section 5, is enabled by the transformation from MCM to UML explained in Section 4. However the current version of the tool has a complementary coverage

criteria based on input coverage of the operation calls and their parameters. The operation calls are used to trigger the transitions in the UML state machine corresponding to the choreography.

Figure 2 depicts the tool architecture based on the MODELPLEX platform. In the middle it shows the MCM module including an MCM editor and a model importer from SAP's existing models. The toolset on the top right of the figure shows the UML test generator and debugger, which are part of the Simulation, Verification and Testing Workbench developed by the MODELPLEX project [14]. This workbench also includes tools for performance simulation and model verification that are not presented here. The connection between MCM and UML is made via the MCM2UML and the TPTP2SAP transformer modules. The SAP in-house MBT solution is not depicted.

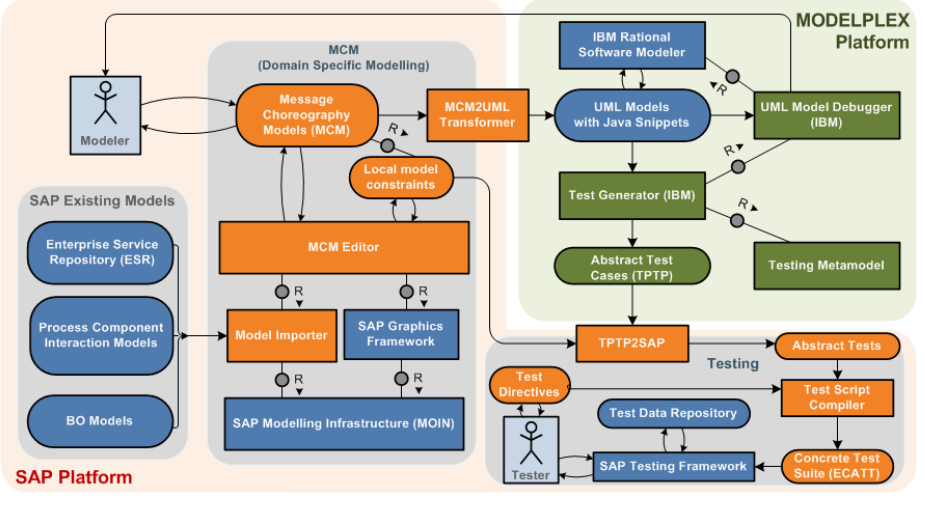


Fig. 2. Tool architecture of the MBT4Chor approach

4 Translation from MCM to UML with Java Annotations

This section describes how the MCM models are transformed into the annotated UML model that serves as the test model. These UML models include the UML classes and composite structures as structural constructs, and the state machines as behavioral constructs. The transformation was programmed in Java using the APIs provided by the MCM tooling and the APIs provided by the EMF-implementation *UML2*[1] for the Eclipse platform of the UML 2 OMG metamodel.

The mapping used in the translation of MCM to UML is sketched below:

a. For each message type *mt* of the choreography, we generate an UML Signal *signal_mt* (denoted by a classifier symbol with the keyword <*signal*>). The data type

[1] http://www.eclipse.org/modeling/mdt/?project=uml2

of the signal is similar to the data type of the message. To keep the test model simple, we ignore the structural information that is not relevant for testing. Additionally, we create a UML SignalEvent associated with each Signal. The signal events trigger the transitions of the UML state machine for the choreography.

b. For each partner p in the choreography, we generate a UML Class class_p. Then:
   - For each service *s_p* provided by a partner *p*, we generate a method *method_s* of the class *class_p*, with the same parameters as the service. The parameters are calculated from the WSDL description of the service. The data types of the parameters are, of course, compliant with the messages exchanged via the service.
   - The action associated with *method_s* is given as an opaque behavior with Java as the language. The code implements the sending of the signal *signal_s_p* associated with the service *s_p* using the signal sending API of the Model Execution engine (*MexSystem).* This is the corresponding code:

     ```
     Signal_S_P signal = new Signal_S_P();
     signal.parameter=parameter;
     MexSystem.send(choreography_instance,signal);
     ```

c. For the choreography protocol, we create a UML Class class_chor and create associations from each of the partners to this choreography class (such that they can reference a choreography instance and send signals to it).
d. We create the initial configuration of the system as a UML composite structure with a choreography instance and an instance for each of the choreography partners. The instances of the partners are connected by UML connectors to the choreography instance.
e. The core of the transformation is given by the translation of the MCM choreography protocol into a UML state machine, which is associated as behavior to the *class_chor* defined above.
   - The concurrent states of MCM are translated into concurrent regions of the state machine.
   - MCM activation of message interaction can involve *OR* and *AND* operations over the MCM local parallel states. They are simulated with junction and join pseudo-states in the UML state machine. Moreover, the effect of an MCM message on the local parallel states is simulated using fork pseudo-states. The initial and end states are mapped to UML initial and final pseudo-states.
   - The MCM guards on the messages are translated into Java guards. Note that complex decision procedures need to be encoded in Java functions. For example, the existential and universal first-order quantifiers, for all ($\forall$) and there exists ($\exists$), we need Java helper methods to be able to implement constraints such as this:

     ```
     forall x: msg [SalesOrder. Item] (x [ProcessingStatusCode] ==
     CONFIRMED);
     ```

     meaning that all items of the sales order transmitted in the message are confirmed.

– The translation of MCM action code and MCM global variables to Java is straightforward and we do not explain it in detail.

Note that the above translation takes into account the special semantics of the executable UML models supported by the IBM tool described in Section 5.

5 UML Test Generator and Model Debugger

This section describes tools that were developed at the IBM Haifa Research Lab and used in our experiment. They are extensions of Rational Software Architect (RSA), which is used to import the MCM descriptions that have been transformed to UML. As explained in Section 4, the structure of the system is described using class diagrams and composite structure diagrams. State machines, activities diagrams, and Java code snippets are used to describe the behavior. In our scenario, the Model Execution Engine [5] executes the model "behind the scenes", while the "main actors" are the model debugger and test generator (see Figure 3, left).

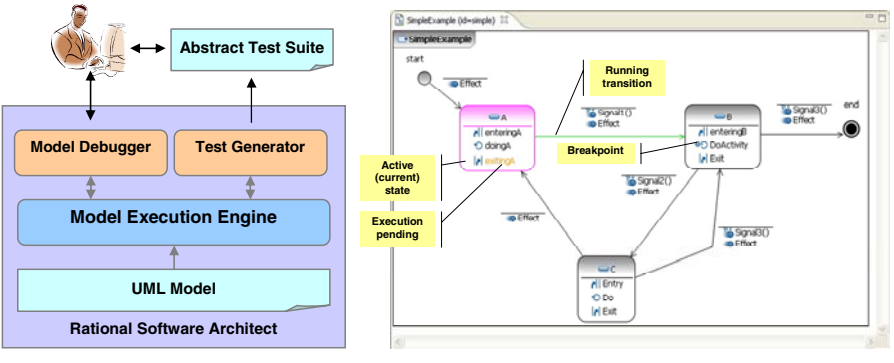


Fig. 3. The architecture of the IBM prototype (left) and a debugging session of a UML model (right)

The model debugger verifies that the model describes the correct expected behavior of the System Under Test (SUT). It enables the user to interact with the executable model in two ways:

– *To control the execution* by sending inputs to the model: that is, create instances, invoke operations on instances, and send signals to instances.
– *To observe the execution* by observing the outputs of the model: that is, the attribute values, active states, and signals to the environment.

The model debugger helps answer the question "*Is there a defect in the model?*" at two different stages: before test generation during the modeling of the SUT, and after test generation, when a test case fails. In the latter case, the defect can be either in the SUT or in the model. If the defect is in the model (its behavior is wrong), the debugger localizes and fixes the defect, thus answering the question, "*Where is the defect?*". The debugger allows the setting of breakpoints on model elements. Figure 3 illustrates

a running state machine with a highlighted active state and running transition. It also shows breakpoints and execution pending elements.

The test generator also uses the model execution engine and acts similarly to the model debugger by sending inputs and observing outputs. The inputs are recorded as test sequences of stimuli applied to the SUT. The outputs are also recorded as expected outcomes. In other words, the model is used as a test oracle predicting the correct behavior. During test generation the next input for a test sequence is selected by the test generator depending on the coverage task chosen by the user:

- *Random* generates random sequences of stimuli (no coverage)
- *Input Coverage* covers as many different inputs as it can (a specific input)
- *Input Step Coverage* covers as many different inputs in each step in the test case as it can (a specific input in a specific step in the test case)
- *Input Step Pair Coverage* covers as many different pairs of inputs in each pair of steps in the test case as it can (a specific pair of input values in a specific pair of steps).

The current version of the test generator only implements *input coverage algorithms*. Their advantage is that such coverages do not face the problem of state explosion. This problem occurs when test generators explore the whole state space of the test model, which can grow very quickly for large test data (as is the case with MCM).

Another advantage of the test generator is that it actually executes the model of the SUT. This enables us not only to generate the sequences of stimuli but also to precisely predict the expected behavior (outputs of the SUT). Test generators that statically analyze the model often have problems to predict these expected results.

Another strength of the test generator is that it uses the same model execution engine as the model debugger. In case of wrongly generated tests resulting from an incorrect test model, the problem can be easily located and fixed in the model using the model debugger. Test generators using their own execution engine might interpret the semantics of the model differently, which complicates the maintenance of the original model.

For the test suite format, we use the Eclipse Test & Performance Tools Platform[2] (TPTP). Test steps (stimuli and observations) reference model elements and a special editor was developed at the IBM Haifa Research Lab for convenient viewing and editing the generated tests. In addition to using the described test generator, tests can also be created manually using the editor. Finally, the test can be either translated to a specific test script for execution on the SUT or alternatively, TPTP can be extended to execute the test directly on the SUT.

6 Test Case Execution in SAP Backend

Once the abstract test cases (in TPTP format) are generated, they must be transformed into executable test scripts. As mentioned in [17], this task is very important and as observed in practice it can take up to half of the time spent on the whole model-based testing approach.

[2] http://www.eclipse.org/tptp

For our approach we have implemented a transformation from the abstract test cases to an internal SAP test language for integration testing. This language follows the keyword-driven testing principles (see [17, Chapter 2]), i.e., it has a higher level of abstraction than the SAP's eCATT script language usually used for testing SAP applications [12]. According to nomenclature of [17, Chapter 8], we use a *mixed approach* for our test concretization, which is a combination of the test adaptation and test transformation modes. To increase the usability and help the testers to visualize the generated test case, we also implemented a transformation of the generated message exchange sequence into UML sequence diagrams.

Since the test data used for the real tests that can be executed on the SUT is very complex, it that it relies on existing master data and different system configurations, we currently do not generate it automatically. Instead we leverage the experience of the testers to reduce the effort and risk of rather difficult test data modeling in a new environment that could be error-prone.

7 Related Work

MBT is an active research area, but it is still not adopted by mainstream industry. In particular, we are not aware of any MBT tools directly running on models using classical choreography languages such as WS-CDL, BPMN, or Let's Dance. Existing MBT approaches for web service (WS) testing concentrate either on testing a single WS by adding state machines to the WSDL interface descriptions [7] or testing WS orchestrations based on BPEL [8]. No approach addresses the utilization of global choreography models for WS interaction testing. In a choreography setting similar to ours, we found only one tool called WS-Engineer [6] which transforms WS-CDL as choreography models and BPEL4WS as local models into labeled transition systems (LTSs). These LTSs are used for model verification, but not for test generation.

The fact that we generate UML models on the basis of MCM opens up the possibility of making use of existing MBT approaches and tools for UML. For instance, approaches for component integration testing based on UML are described in [1,11]. Examples of commercial tools for MBT based on UML are: ATG tool[3] from Rhapsody based on UML, Test Designer from Smart Testing[4] for UML with OCL as action code, and QTronic[5] from Conformiq for UML with Java annotations. However, using any of these tools or approaches should be decided after a careful analysis of capabilities, input test modeling and output test format, and semantics of the used executable UML.

8 Conclusions

This paper presents an end-to-end process for model-based testing of service choreographies, starting with the modeling using the domain specific language MCM, its transformation to UML, the test generation and finally test execution in the backend. Although we managed to automate a large part of this complex process, there is still room for improvement. This section describes some of the experiences we gained.

[3] http://modeling.telelogic.com/products/rhapsody/test/automated-test-generation.cfm
[4] http://www.smartesting.com
[5] http://www.conformiq.com/qtronic.php

Lessons learned and challenges encountered: Given the proprietary modeling stack at SAP, we had to design a DSL called MCM for modeling choreography with the purpose of test generation rather than directly using UML (see also [9]). The designed language has a precise semantics and incorporates existing SAP metamodels and feedback from SAP architects and testers to foster internal adoption. Our approach relies on the mature test execution environment of SAP that provides keyword-driven testing tools. The disadvantage of a DSL is that mature MBT tools cannot be directly applied, but model transformations to general purpose languages are needed. We learned from the experiences of the AGEDIS project [10] and chose UML with Java annotations as the target language. The Java language has an imperative semantics (as opposed to OCL, for instance) and is expressive enough to capture our complex guards based on first-order logic. The model execution engine [5] used by the test generator described in Section 5 is able to execute UML with Java annotations. While the test generation and test execution can be automated, we are currently not able to fully automate the test data generation. This is due to the complex master data and test data constraints in an ERP system [21] that cannot be easily be modeled. Although some of the constraints on the test data can be captured in the MCM guards, it is still possible that we generate infeasible paths for which we cannot provide proper test data. These must be filtered out manually by the testers and the feedback incorporated into the test generator that must provide alternative paths. We will look at ways to improve these inconveniencies based on the test pilots currently running at SAP.

Future Work: Our plans are driven by the above mentioned challenges in the test data provision area. For that, we are currently working towards incorporating a model-checker and constraint-based solver that could help to reduce the number of infeasible paths due to data inconsistencies. This is, however, a difficult problem (even undecidable) in general. We also plan to better support and validate the semantical relation between the local and global views of MCM and the traceability link between MCM and the generated UML. Moreover, we will evaluate test effectiveness and bug detection capabilities based on different model coverage criteria. We also want to experiment with the UML-based MBT tools from Section 7, but first the effort to accommodate their different semantics (see also [4]) and corresponding model transformations must be evaluated.

Acknowledgments. This work was partially supported by the EC-funded project MODELPLEX [14]. We thank Roger Kilian-Kehr for useful comments on a draft of this paper.

References

1. Ali, S., Briand, L., Jaffar-Ur Rehman, M., Asghar, H., Iqbal, M.Z., Nadeem, A.: A State-Based Approach to Integration Testing Based on UML Models. Information & Software Technology 49(11–12), 1087–1106 (2007)
2. Benedetto, C.: SOA and Integration Testing: The End-to-end View. SOA World Magazine 6(8) (2006)
3. Business Process Modeling Notation (BPMN) Specification, Final Adopted Specification. Technical report, Object Management Group (OMG), `http://www.bpmn.org`
4. Crane, M., Dingel, J.: UML vs. Classical vs. Rhapsody Statecharts: Not All Models Are Created Equal. Software and System Modeling 6(4), 415–435 (2007)

5. Dotan, D., Kirshin, A.: Debugging and Testing Behavioral UML Models. In: OOPSLA Companion 2007, pp. 838–839. ACM Press, New York (2007)
6. Foster, H., Uchitel, S., Magee, J., Kramer, J.: WS-Engineer: A Model-Based Approach to Engineering Web Service Compositions and Choreography. In: Test and Analysis of Web Services, pp. 87–119. Springer, Heidelberg (2007)
7. Frantzen, L., Huerta, M.N., Kiss, Z.G., Wallet, T.: On-The-Fly Model-Based Testing of Web Services with Jambition. In: 5th Int. Workshop on Web Services and Formal Methods (WS-FM 2008). LNCS. Springer, Heidelberg (2009) (to appear)
8. García-Fanjul, J., de la Riva, C., Tuya, J.: Generation of Conformance Test Suites for Compositions of Web Services Using Model Checking. In: Proc. of TAIC PART 2006, pp. 127–130. IEEE Computer Society, Los Alamitos (2006)
9. Hartman, A., Katara, M., Olvovsky, S.: Choosing a Test Modeling Language: A Survey. In: Bin, E., Ziv, A., Ur, S. (eds.) HVC 2006. LNCS, vol. 4383, pp. 204–218. Springer, Heidelberg (2007)
10. Hartman, A., Nagin, K.: The AGEDIS Tools for Model Based Testing. In: Jardim Nunes, N., Selic, B., Rodrigues da Silva, A., Toval Alvarez, A. (eds.) UML Satellite Activities 2004. LNCS, vol. 3297, pp. 277–280. Springer, Heidelberg (2005)
11. Hartmann, J., Imoberdorf, C., Meisingerm, M.: UML-Based Integration Testing. In: Proc. of ISSTA 2000, pp. 60–70. ACM Press, New York (2000)
12. Helfen, M., Lauer, M., Trautwein, H.M.: Testing SAP Solutions. SAP Press (2007)
13. Kavantzas, N., Burdett, D., Ritzinger, G., Lafon, Y.: Web Services Choreography Description Language Version 1.0. W3C Candidate Recomm, Technical report (2005)
14. MODELPLEX Project. Funded by European Commission, FP6, Grant no. 034081, http://www.modelplex.org
15. O'Leary, D.E.: Enterprise Resource Planning Systems – Systems, Life Cycle, Electronic Commerce and Risks. Cambridge University Press, Cambridge (2000)
16. SAP AG, Enterprise SOA in a Nutshell (2007), http://help.sap.com/redirect_sdn_esoa/redirect_esoainanutshell.htm
17. Utting, U., Legeard, B.: Practical Model-Based Testing – A Tools Approach. Morgan Kaufmann Publ., San Francisco (2007)
18. Web Services Reliable Messaging (WS-ReliableMessaging), Version 1.1. OASIS Consortiom, http://docs.oasis-open.org/ws-rx/wsrm/v1.1/wsrm.pdf
19. Wieczorek, S., Roth, A., Stefanescu, A., Charfi, A.: Precise Steps for Choreography Modeling for SOA Validation and Verification. In: International Symposium on Service-Oriented Software Engineering (SOSE 2008), pp. 148–153. IEEE Computer Society, Los Alamitos (2008)
20. Wieczorek, S., Stefanescu, A., Großmann, J.: Enabling Model-Based Testing for SOA Integration Testing. In: Proc. of 1st Model-based testing in practice workshop (MOTIP 2008), pp. 77–82. Fraunhofer IRB Verlag (2008)
21. Wieczorek, S., Stefanescu, A., Schieferdecker, I.: Test Data Provision for ERP Systems. In: Int. Conf. on Software Testing, Verification and Validation (ICST 2008), pp. 396–403. IEEE Computer Society, Los Alamitos (2008)
22. World Wide Web Consortium (W3C): Web Service Glossary. Version 20040211, http://www.w3.org/TR/ws-gloss
23. Woods, D., Mattern, T.: Enterprise SOA – Designing IT for Business Innovation. O'Reilly, Sebastopol (2006)
24. Zaha, J.M., Barros, A., Dumas, M., ter Hofstede, A.: Let's Dance: A Language for Service Behavior Modeling. In: Meersman, R., Tari, Z. (eds.) OTM 2006. LNCS, vol. 4275, pp. 145–162. Springer, Heidelberg (2006)

Model-Based Interoperability of Heterogeneous Information Systems: An Industrial Case Study

Nikola Milanovic[1], Mario Cartsburg[1], Ralf Kutsche[1], Jürgen Widiker[1], and Frank Kschonsak[2]

[1] TU Berlin
{nmilanov,mcartsbg,rkutsche,jwidiker}@cs.tu-berlin.de
[2] Klopotek AG
f.kschonsak@klopotek.de

Abstract. Integration of heterogeneous and distributed IT-systems is one of the major cost-driving factors in the software industry. We introduce a model-based approach for information system integration and demonstrate it on the industrial case-study of data integration between the Oracle database management system and the SAP R/3 enterprise resource planning system. Particular focus is on multi-level modeling abstractions, integration conflict analysis (automatic data model matching), semantic reasoning, code generation and tool support.

1 Introduction and Related Work

Integration of complex and heterogeneous IT-systems is one of the major cost-driving factors in the software industry today. There is an increasing need to systematically address integration in accidental architectures, that have grown in an uncontrolled manner in heterogeneous enterprise environments.

Schema matching approaches [1] detect dependencies between data model elements at the model or instance levels. Extract-Transform-Load (ETL) tools, such as CloverETL, use schema matching methodology to enable integration of multiple data sources. Such tools are today primarily used for data warehousing. Furthermore, languages exist that enable specification of transformations between data models, such as Ensemble [2]. However, the applicability of all mentioned approaches diminishes with the increased heterogeneity of the underlying systems, for example, when they are not relational, or when data model is not immediately accessible in form of an ER model or UML class diagram.

Another approach is the Service Oriented Architecture (SOA). In SOA, data sources are wrapped as services and accessible to the Enterprise Service Bus (ESB) engine, which orchestrates data and functional logic. The business process orchestration standard is de facto BPEL, and there are numerous commercial and open-source ESB engines. ESB however expects that all service endpoints are compatible and no data or behavior conflicts will occur between them. Otherwise, either the endpoint itself has to be modified (frequently impossible), or the transformation is performed at the message level (XSLT or BPELJ). From

R. Paige, A. Hartman, and A. Rensink (Eds.): ECMDA-FA 2009, LNCS 5562, pp. 325–336, 2009.

the architectural standpoint, this is bad as it introduces mixing of orchestration, data model and implementation. Furthermore, all ESB implementations assume mandatory SOAP/XML serialization, which restricts application domain and hampers performance. Finally, approaches such as mapping editors (e.g., Altova MapForce) and extended UML Editors (e.g., E2E Bridge) lack code generation, and as such are more suitable for analysis than for development.

For these reasons, as a part of the R&D program of the German government, project BIZYCLE (www.bizycle.de) was started in early 2007 in order to investigate in large-scale the potential of model-based software and data integration methodologies, tool support and practical applicability for different industrial domains. In this paper we present one BIZYCLE industrial case-study and try to identify potential advantages of the proposed process, platform and tools.

2 Integration Scenario and the Current Solution

In the media and publishing industry, there are often several IT-systems which have to synchronize their data. We describe a scenario where the Oracle database server of an IT software provider has to send Customer Master Data to the SAP ERP System, which is hosted on its customer side. The Oracle database stores customer information and is therefore the basis of any financial transaction. The SAP system manages financial accounting services.

In the current solution, Oracle database exports required data in a CTM file. This is a proprietary data format created by the software provider. It is then converted into plain-text ASCII file, stored on a file server (provider's endpoint). A dedicated ABAP program (SAP legacy language) on the customer side is reading this file and converting it into a proprietary Batch Input structure which is then imported. The scenario is point-to-point, asynchronous and with several modal fragmentations. Many formats are used and responsibilities in case of inconsistent data are not always clear to address. Furthermore, fragmentations make the evolution very difficult. Additional 3rd party supplier has also to be hired to develop the import ABAP routine.

Although outdated and inflexible, the solution represents the industrial practice in the field of the large-scale data integration. It is influenced by factors such as presence of the legacy code (e.g., CTM export, ABAP import), time-to-market pressure, mixture of the business, presentation and data layers, and high investments and unaffordable learning curve required for the refactoring. Furthermore, sector-specific factors in the media and publishing sector (complex and dynamic business processes, strong graphical and aesthetic requirements and outdated technologies) contribute to the current unfavorable situation.

3 BIZYCLE Integration Process

Integration tasks are performed today by experienced software engineers and application experts, manually programming the *connectors* i.e. the glue among

software components or systems. This requires domain-specific as well as integration requirements' analysis skills. In order to make the error-prone and expensive integration tasks more feasible, we developed a methodology for data and software integration, calling it the BIZYCLE integration process [3,4,5,6]. The essence of the BIZYCLE integration process is application of model-based methodologies for providing multi-level integration abstractions, thus decoupling the knowledge required for integration and making the process partially automatic.

The levels of abstraction supported are the computation independent model (CIM), platform specific model (PSM) and platform independent model (PIM). The CIM level captures the integration business process. This model is refined at the PSM level where properties of interfaces realizing the integration scenario are described, such as structure, semantics, communication, non-functional properties and behavior. All PSM models are transformed to the PIM level, which represents a common abstraction level which is used to discover integration conflicts (incompatible interfaces). Note that the ordering CIM-PSM-PIM departs from the usual MDA recommendation CIM-PIM-PSM. The reason is that we perform system integration and need to collect technical information about the existing interfaces first and then abstract them in order to compare and mediate between them. Based on the results of the conflict analysis, appropriate connector model and code for the connector component are generated. In the following sections we will demonstrate application of the BIZYCLE integration process using the Oracle-SAP data integration case study as an example.

3.1 Computation Independent Model

The integration requirements are described at the computation independent model (CIM) level. It encapsulates technology-agnostic details of the integration business process – no detailed knowledge about the underlying platform-specific technology is required. The CIM is visualized in four different graphical views (flow, component, object and connection), reducing the model complexity.

The CIM flow view consists of compartments similar to UML swim lanes which represent either a system/application or a connector. Systems have several import and export interfaces and business functions. A connector consists of special business functions called connector functions. Elements of a swim lane can be connected via directed control flow links and system interfaces and connector functions can be connected across the swim lanes by connections (information flow). In the present integration scenario, there are two systems (Oracle DBMS and SAP ERP) connected via one connector (Figure 1). At the CIM level there are two export business interfaces (*CustomerExport* and *SalesLedgerExport*) and one import business interface (*SAPCustomer*). The Oracle data are aggregated by a connector function. All business objects involved in the integration scenario are defined in the CIM object view. To be able to create complex business objects, aggregation and containment are supported. In our case there are three defined business objects: *OracleCustomer*, *OracleSalesLedger* and

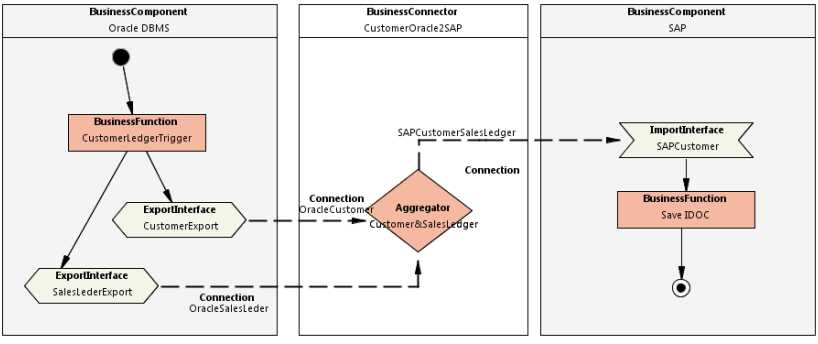


Fig. 1. CIM Flow View

SAPCustomerSalesLedger, which are also semantically annotated (see Figure 4a and Table 1). Connections defined in the CIM flow view transport business objects, which can be associated with the corresponding business functions.

3.2 Platform Specific and Platform Independent Models

Interfaces realizing the Oracle-SAP scenario have to be described in a technology-specific way now. This is necessary to perform the conflict analysis, generate application endpoints for every interface, and specify the communication between systems and connectors. The language for interface descriptions is a platform specific model (PSM). There is one PSM metamodel for each platform currently supported: relational databases, XML files, Web Services, J2EE and .NET components and ERP systems. PSMs can be generated manually using a graphical model editor or they can be extracted automatically from the system interfaces. Oracle models are generated based on the underlying Oracle data model, and SAP model is generated by reading the Business Object Repository component. Models presented in this paper are excerpts from the real scenario.

There are three technical interfaces involved in the Oracle-SAP scenario. On the Oracle side, interfaces `Customer` and `SalesLedger` are shown in Figure 2. The respective models contain two SQL query interface elements with access information (not shown in Figure). A query element consists of an output transfer table. The SAP side exposes the interface `Customer` of the IDOC type, consisting of `DEBMA05` segments. While the whole IDOC PSM contains 35 segments with 613 fields, Figure 3 shows only the small excerpt (`E1KNA1M` and `E1VCKUN` segments) to illustrate the integration scenario. The columns in the Oracle PSM and all segment fields in the SAP PSM are annotated with semantic concepts (Table 1 and Figure 4a). Business requirements at the CIM level are thus matched with the technical elements at the PSM level. Semantic annotations are realized by a model weaving component. The semantic knowledge is represented using a metamodel mirroring Resource Description Framework (RDF).

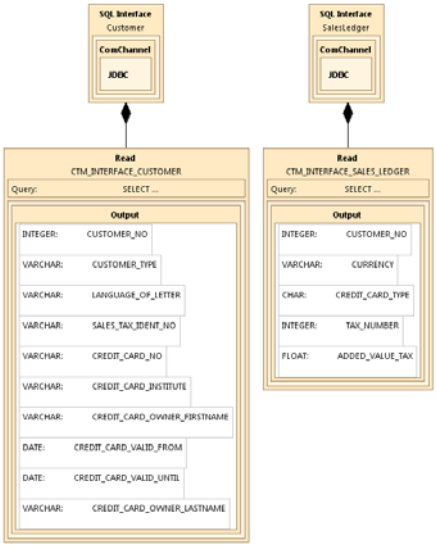


Fig. 2. Oracle Platform Specific Model (excerpt)

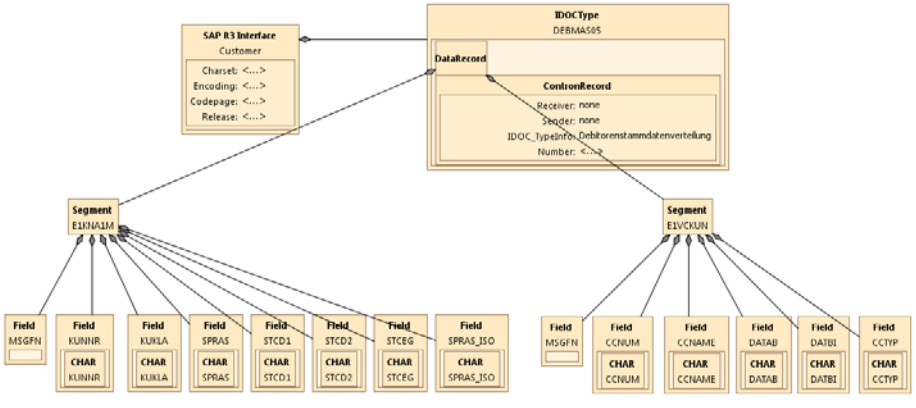


Fig. 3. SAP Platform Specific Model (excerpt)

Platform independent model (PIM) is used to abstract platform-specific properties from interface description models. This is a precondition for the conflict analysis process. The PIM core contains interface, structure, property, behavior, communication and semantic packages. Depending on the source PSM, an interface can be functional, object-oriented/method, document or data. Each interface and its parameters have a structure defined within platform independent, common type system. Platform specific types are then mapped into this type system in the process of model transformation. Properties of platform specific communication channels are abstracted using `AccessPattern` and

`MessageExchangePattern` elements, describing function- and document-based communication styles. The PIM is generated automatically by the BIZYCLE integration framework using model-to-model transformation component. Table 1 shows semantic annotations for the PIM, which are transferred from respective PSMs. After the model transformation, system interfaces are described in a common, technology-independent language and ready for the conflict analysis.

3.3 Conflict Analysis

The purpose of the conflict analysis is detection of mismatches between component interfaces. The conflict analysis process consists of five ordered analysis steps which handle conflicts of the corresponding type: semantic, behavior, property, communication and structure analysis.

The semantic analysis phase uses platform-independent models (PIMs), domain ontology and the list of defined requirements generated on the basis of the CIM model. It outputs a list of requirement mappings in the context of a shared domain ontology. Inputs of the behavior and property conflict analysis are PIMs, CIM, application protocol logic (APL) (set of interface execution paths) and the requirement mappings. Behavior analysis checks control and data flows specified in CIM as well as requirement mappings against the behavior constraints described with APL, to determine a conflict-free call order within the connector logic, using simulation and bisimulation [7] as well as invariant preservation proofs [8]. Behavior and property conflict analysis phases deliver connector call logic (CCL) containing a conflict-free interface call order and the list of message processors which resolve discovered conflicts. Communication conflict analysis examines communication properties of interfaces, such as synchronicity or message exchange patterns, using process algebra [9]. Finally, structural analysis addresses data types and the runtime value format to overcome structural heterogeneity using typing and subtyping rules [10]. More information about the conflict analysis may be found in [11].

The conflict analysis of the Oracle-SAP scenario consists of two parts: semantic (SemCA) and structural (StCA) analysis. As there is no functional coupling, behavior and property analysis are skipped. The starting point of SemCA is creation of the requirements. Based on the data flow, we distinguish between Business Object Export Requirements (BOER) and Business Object Import Requirements (BOIR). BOERs are responsible for the extraction of business objects from the Oracle database (*OracleCustomer* and *OracleSalesLedger*) while BOIRs cover the passing of the business object to the SAP system (*SAPCustomerSalesLedger*). SemCA creates a requirement mapping for each identified requirement. Additional requirements can emerge during the conflict analysis process. A requirement mapping references the corresponding business requirement and the (parf of) PIM interface, which meets the requirement: the mapped elements refer to the same or equivalent ontology concept. The ontology and annotations are given in Table 1 and Figure 4a.

The business object *OracleSalesLedger* is mapped to the export parameters `CREDIT_CARD_VALID_FROM`, `CREDIT_CARD_VALID_UNTIL` and `CREDIT_CARD_NO`

of the interface CTM_INTERFACE_CUSTOMER as well as CREDIT_CARD_TYPE of the CTM_INTERFACE_SALES_LEDGER. Two mappings for the concept *Customer Number* are created since both Oracle interfaces provide export parameters annotated with this concept (CUSTOMER_NO). The same applies to the business object *OracleCustomer*. At this point the user has an option to eliminate one of the redundant mappings. Since no export parameters annotated with the concept *Name* can be found, the algorithm uses logical reasoning to identify the domain concept *Name Concatenation* which delivers a value annotated with the required concept (*Name*) and its implementation *ConcatenateName*. The newly generated requirements are fulfilled by mappings to CREDIT_CARD_OWNER_FIRSTNAME and CREDIT_CARD_OWNER_LASTNAME annotated with the concepts *Firstname* and *Lastname*.The remaining parts are mapped to the export parameters CUSTOMER_TYPE, ADDED_VALUE_TAX, LANGUAGE_OF_LETTER, TURNOVER_TAX and TAX_NUMBER.

Table 1. Semantic Annotations

Oracle DBMS

PSM: SQLInterface Customer
PIM: FunctionInterface CTM_INTERFACE_CUSTOMER

Element Name	Annotated with
CUSTOMER_NO	Customer Number
CUSTOMER_TYPE	Customer Type
LANGUAGE_OF_LETTER	Customer Language
SALES_TAX_IDENT_NO	Turnover Tax
CREDIT_CARD_NO	Credit Card Number
CREDIT_CARD_INSTITUTE	Credit Card Institute
CREDIT_CARD_OWNER_FIRSTNAME	Firstname
CREDIT_CARD_VALID_FROM	Credit Card Valid From
CREDIT_CARD_VALID_UNTIL	Credit Card Valid To
CREDIT_CARD_OWNER_LASTNAME	Lastname

PSM: SQLInterface SalesLedger
PIM: FunctionInterface CTM_INTERFACE_SALES_LEDGER

Element Name	Annotated with
CUSTOMER_NO	Customer Number
TAX_NUMBER	Tax Number
CREDIT_CARD_TYPE	Credit Card Type
ADDED_VALUE_TAX	Added Value Tax

SAP R3

PSM: Segment E1KNA1M
PIM: Document Interface DEBMAS05

Element Name	Annotated with
KUNNR	Customer Number
KUKLA	Customer Type
SPRAS	Customer Language
STCD1	Added Value Tax
STCD2	Tax
STCEG	Tax Number
SPRAS_ISO	Customer Language

PSM: Segment E1VCKUN
PIM: Document Interface DEBMAS05

Element Name	Annotated with
CCNUM	Credit Card Number
CCNAME	Credit Card Owner
DATAB	Credit Card Valid From
DATBI	Credit Card Valid To
CCTYP	Credit Card Type

Provided Function

Element Name	Annotated with
ConcatenateName (Function)	Name Concatenation
Firstname (Import Parameter)	Firstname
Lastname (Import Parameter)	Lastname
Name (Export Parameter)	Name

Business Objects

Business Object Name	Annotated with
OracleCustomer	Customer Number Customer Type Name Customer Language Tax Number Turnover Tax Added Value Tax
OracleSalesLedger	Customer Number Credit Card Number Credit Card Valid From Credit Card Valid To Credit Card Type
SAPCustomerSalesLedger	Customer Number Customer Type Name Customer Language Tax Number Turnover Tax Added Value Tax Credit Card Number Credit Card Valid From Credit Card Valid To Credit Card Type

The algorithm now uses logic reasoning to map the part of the business object *SAPCustomerSalesLedger* annotated with *TurnoverTax* to the import parameter STCD2 annotated with *Tax*. This is possible due to the directed relation *isA* between both concepts. Since no import parameters annotated with the concept *Name* can be automatically detected, the algorithm yields a non-resolvable semantic conflict, which can be overcome if CCNAME is correctly annotated with *Name* as it contains only a name string. After this correction, the conflict is

removed. The business object part annotated with *Customer Language* is mapped to both input parameters SPRAS and SPRAS_ISO.

The first step of the structural conflict analysis is the building of abstract data blocks to compose input parameters of the PIM interfaces. Using semantic mappings the interface call order is determined to obtain the input parameter set of each concrete technical interface. Next, for each required input parameter the algorithm uses semantic mappings to determine corresponding export parameter that will be passed to the import interface and to compare data types of both elements. Detected disparity of data types indicates a structure conflict that is eliminated by adding the appropriate type transformer to the connector model, if possible (determined by the subtyping and conversion rules of the common type system). In the scenario a structure conflict between KUNNR of type *String* and CUSTOMER_NO of the type *Number* in both interfaces CTM_INTERFACE_CUSTOMER and CTM_INTERFACE_SALES_LEDGER is detected. Therefore, a conversion from number to string will be performed. Another conflict exists between CREDIT_CARD_VALID_FROM and CREAD_CARD_VALID_UNTIL of the type *Date* with DATAB and DATABI of the type *String*. Therefore, a conversion from date to string will be performed here. Finally, there is a structural conflict between TAX_NUMBER and ADDED_VALUE_TAX of the types *Number* and *Float* on the CTM_INTERFACE_SALES_LEDGER side and STCEG and STCD1 of the type *String* on the E1KNA1M side. The first conflict requires number to string, and the second float to string conversion. The results of the conflict analysis process (requirement mapping and transformations) are shown in the Figure 4b.

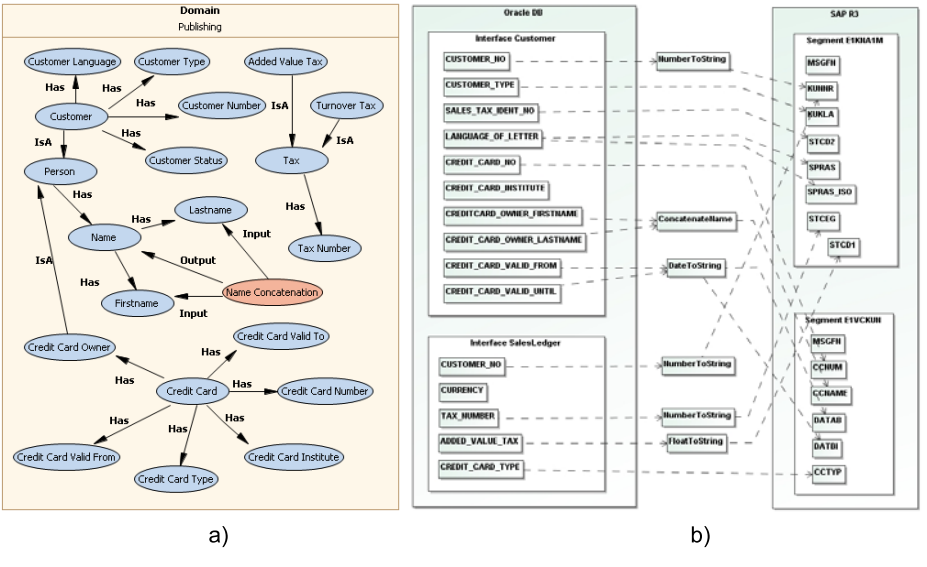


Fig. 4. a) Domain Ontology (excerpt) b) Requirement Mappings

3.4 Connector and Code Generation

The connector is a mediator component that resolves integration conflicts by matching capabilities and requirements of the components that are to be integrated [12]. Connector generation is based on the generative programming and code generation methods, as the formal analysis will first produce a connector model, based on which the code for the target runtime middleware is then generated. The connector generation takes the PSM/PIM models and results of the conflict analysis process as inputs and performs generation of application (service) endpoints and transformations that ensure conflict-free integration. At the runtime level we treat all exchanged information as messages. The metamodel abstraction used for resolving semantic and structural conflicts are message processors: senders, receivers, generators (timers), filters, transformers, aggregators and routers, implementig a subset of EAI patterns [13]. Messages themselves can encapsulate events, data objects or function calls. Transformation requirements are specified during the conflict analysis process and formalized using UML action semantics. In this case, two data messages are received from the Oracle endpoint, aggregated, transformed (*NumberToString*, *FloatToString*, *DateToString* and *ConcatenateName*) and finally received by the SAP endpoint.

We currently support Java Emitter Template (JET), open Architecture Ware (oAW) and Java reflection for the code generation. The code generation using JET and oAW is based on code templates. In case of Java reflection a generic Call object is deployed which is able to interpret any given PSM. One advantage of this concept is the flexibility of a deployed connector, which can be modified online without stopping. A disadvantage may be the absence of any visible source code which can discourage end users who want to retain source code ownership.

4 Implementation: BIZYCLE Integration Framework

The framework used to realize this case-study, BIZYCLE Model-Based Integration Framework (MBIF), is the prototypic realization of the project results. It consists of the Integration Platform, Repository and the Runtime Environment.

BIZYCLE Integration Platform is based on the Eclipse Rich Client Platform (RCP) and is implemented using RCP plugins. The integration platform supports project management, model creation, editing, extraction and transformation, conflict analysis, connector and code generation.

The integration project management component consists of the workspace, workflow, repository and deployment/runtime management components. The integration process is managed by a workflow engine. Each integration step is interpreted as a workflow task which can be assigned to multiple users. Pre- and post-conditions, as well as the appropriate integration artifact types, are defined for every task. The project management component further offers user management capabilities such as definitions of roles and access rights. Runtime environment management supports connector deployment and testing. Different models represent specific levels of abstraction. In the Oracle-SAP scenario those

are four CIM views, various PSMs and the PIMs, ontology model and finally the connector model. All models are represented, instantiated and persisted as instances of the Ecore metamodel. Beside the tree-based model editors, a graphical editor for each model type is provided. All figures describing the scenario in this paper are directly exported from these graphical model editors, generated using Eclipse Graphical Modeling Framework (GMF). Platform specific models can be extracted automatically from the underlying systems using the PSM extraction component, which investigates the interface structure and generates an adequate PSM. After this step, PSMs can be viewed, changed or annotated using model editors. The PSM-PIM model transformation component is based on the Eclipse M2M project and rules are specified using ATL. The conflict analysis component uses SWI-Prolog engine to find/solve integration conflicts. The artifacts such as CIM, PIM and semantic model are transformed to a set of Prolog facts that represents the necessary knowledge base. The component then generates queries and creates appropriate connector model objects according to the received facts. The connector generation component generates application endpoints and the connector core. The generator component supports different technologies: JET, oAW and Java reflection. The generated connector parts can be deployed in the BIZYCLE Runtime Environment using deployment management capability of the project management.

The BIZYCLE Repository [5] stores and manages model artifacts and their metadata. Supported artifacts are metamodels, models, transformation rules, code and documentation. User and project information, workflow descriptions and definitions are also stored in the repository. The (long-lived) metamodels and their domain-oriented specializations can be shared among many partners using the repository as a conceptual basis for development and standardization. Browsing the project artifacts, going back in the version history of an artifact and the check-in/check-out functionality is supported by the repository. It offers a consistency preservation mechanism, based on the consistency rules and actions which are triggered whenever a repository may be in an inconsistent state. The current implementation is based on the Subversion (SVN) repository for content and Jena Semantic Web Framework for Java for metadata.

BIZYCLE Runtime Environment is based on the Glassfish OpenESB technology. Deployed application endpoints offer a SOAP interface or a JMS gateway. The connector core consists of different message processors. A specific message processing workflow, determined in the conflict analysis phase and generated as BPEL, is executed to resolve integration conflicts. We omitted several runtime environment capabilities and features from this paper, as their inclusion would make models' size prohibitive, such as modeling security and handling platform-specific exceptions. BIZYCLE can store sensitive authentication/authorization data in the repository, or include them directly into models. For providing business data security, services of the underlying middleware, such as encryption, are used. The runtime is also able to catch platform-specific exceptions (insofar as they are part of the PSMs), handle them and depending on the exception type abort or rollback the part or entire integration process.

5 Lessons Learned

The model-based integration process and the tool suite described in this paper have entered industrial validation phase. We summarize the feedback below.

The *automation* is achieved through model extraction, systematic conflict analysis process and code generation. Up to now, these error-prone steps have been performed manually, frequently resulting in costly data inconsistency and incompatibility problems. *Reuse* is supported at the model-level, as interface descriptions, transformation rules and semantic annotations can be shared via BIZYCLE Repository. One of the biggest integration challenges is the evolution of business requirements or backend systems. *Evolution* and *maintenance* are now supported at the model level, and code generation methods enable smooth transitions. *Metamodeling* enables very fast tool prototyping and we had only eighteen months between requirements specification and the first industrial prototype. However, metamodels also improve understanding of the problem domain. *Multiple abstraction levels*, such as CIM, PSM, PIM and code, offer business architects the possibility not to start at the data model or code level directly. The essential benefit offered by the multi-level modeling environment is based on the capability of performing *automated model transformations*, abstracting and refining over the given level hierarchy. We also observed *smoother communication* between project managers and developers using the CIM and PSM abstractions. Furthermore, the entire integration process is *retained in-house*, generated code is owned, and outsourcing (e.g., ABAP programming) is not required. Endpoint generation enables the use of the *standardized and native system interfaces* (e.g., SAP IDOC), instead of making costly workarounds (CTM and ABAP conversion). We were able to successfully *exchange the underlying middleware* and to generate integration code based on the provided models with no manual intervention. Thanks to the *use of Domain Specific Languages (DSL)*, the practice showed an *affordable modeling learning curve*, even with staff with no previous modeling experience. The *scalability* is supported through automatic model extraction and the use of tree-based editors.

However, model-based methods are not the panacea for large-scale integration problems, as certain open-issues remain. *Metamodeling learning curve* remains prohibitive, even for market-leaders. Therefore, a knowledge base provider (community or standardization board) is required to provide support for the metamodel and platform evolution. *Performance* is an issue pending further investigation. Introducing additional levels of indirection (model levels and transformations, code generators and the runtime) may impact scenario performance. Optimizations at the model transformation and the runtime levels are being investigated. Regardless of tree-based editors, *scalability* is still an issue of large scenarios. We worked with models with tens of thousands of nodes (complex ERP data models) and identified an acute need for adequate *model partitioning* methods and *model search* functions. The BIZYCLE Repository partially addresses this problem already. With large models, *automatic semantic annotation* becomes a must. An additional effort is currently underway to provide a tool which can extract semantic information from the Oracle, SAP and Web Service

platforms and unify it under a single ontology metamodel. We experienced the *lack of standardization of the semantic knowledge* and focus on further research in automatic ontology generation, ontology merging, and on the integration of metamodel knowledge and ontology knowledge representations. Finally, there is a need to support the connector life after deployment, primarily with model-based *testing* and *dynamic configuration*.

While acknowledging mentioned issues and drawbacks, experience shows that model-based methods, particularly domain specific languages such as CIM, PSM and PIM, as well as model transformation and code generation techniques, show promise to improve several aspects of the industrial practice in the area of large-scale, heterogeneous information systems integration, primarily by introducing a disciplined, methodical and reusable integration process.

References

1. Rahm, E., Bernstein, P.: A survey of approaches to automatic schema matching. VLDB Journal 10(4), 334–350 (2001)
2. InterSystems: Ensemble data transformation language (2006), `http://www.intersystems.com/ensemble/docs/4/PDFS/DataTransformationLanguage.pdf`
3. Kutsche, R., Milanovic, N.: (Meta-)Models, Tools and Infrastructures for Business Application Integration. In: UNISCON 2008, pp. 579–584. Springer, Heidelberg (2008)
4. Kutsche, R., Milanovic, N., Bauhoff, G., Baum, T., Cartsburg, M., Kumpe, D., Widiker, J.: BIZYCLE: Model-based Interoperability Platform for Software and Data Integration. In: Proceedings of the Model-Driven Tool and Process Integration Workshop at ECMDA (2008)
5. Milanovic, N., Kutsche, R., Baum, T., Cartsburg, M., Elmasgunes, H., Pohl, M., Widiker, J.: Model & Metamodel, Metadata and Document Repository for Software and Data Integration. In: Proc. ACM/IEEE 11th International Conference on Model Driven Engineering Languages and Systems (MODELS), pp. 416–430 (2008)
6. Agt, H., Bauhoff, G., Cartsburg, M., Kumpe, D., Kutsche, R., Milanovic, N.: Metamodeling Foundation for Software and Data Integration. In: Proc. 8th International Conference on Information Systems Technology and Applications (ISTA) (2009)
7. Leicher, A.: Analysis of Compositional Conflicts in Component-based Systems. Ph.D Dissertation, TU Berlin (2005)
8. Milanovic, N.: Contract-based Web Service Composition. HU, Berlin (2006)
9. Milner, R.: The Polyadic π-calculus: A tutorial. In: Logic and Algebra of Specification. Springer, Heidelberg (2003)
10. Busse, S., Leicher, A., Süss, J.: Analysis of Compositional Conflicts in Component-Based Systems. In: Gschwind, T., Aßmann, U., Nierstrasz, O. (eds.) SC 2005. LNCS, vol. 3628, pp. 67–82. Springer, Heidelberg (2005)
11. Agt, H., Widiker, J., Bauhoff, G., Milanovic, N., Kutsche, R.: Model-based Semantic Conflict Analysis for Software- and Data-integration Scenarios. Technical Report 2009/7, Berlin University of Technology (2008)
12. Mehta, N., Medvidovic, N., Phadke, S.: Towards a Taxonomy of Softwar Connectors. In: Proceedings of the 22nd International Conference on Software Engineering, Toronto, Canada, pp. 178–187. ACM Press, New York (2000)
13. Hohpe, G., Woolf, B.: Enterprise Integration Patterns. Addison-Wesley, Reading (2003)

Author Index