MoDeV²a 2006: Perspectives on Integrating MDA and V&V

http://modeva.itee.uq.edu.au

ACM/IEEE 9th International Conference on Model Driven Engineering Languages and Systems

Proceedings
3rd International Workshop

MoDeV²a: Model Development, Validation and Verification.

Edited by
Benoît Baudry
David Hearnden
Nicolas Rapin
Jörn Guy Süß

2 October 2006
Genova, Italy

Preface

The MoDeVa series of workshops offers a forum for researchers and practitioners with varying backgrounds to discuss new ideas concerning links between model-based design and model-based validation. It aims to introduce models as a link between the theoretic foundations of Validation and Verification, and the practice of model-based testing. Also, it seeks to critically examine the pivotal parts of the Model-Driven Architecture paradigm: Models, Metamodels and Transformations.

MDA and its related approaches (DSL, MDE, ...) primarily revolve around manual refinement and automated transformation of models. This approach is successful at quickly generating results. However, it is difficult to gauge the quality of those results. Is the result of a transformation really what the user intended? Does the computed result of a transformation really conform with its specified result? Such questions about intended and specified behaviour usually delineate the domain of Validation and Verification (V&V). V&V is an established area of research, and a transfer of ideas between V&V and MDA might help to improve quality and reliability of MDA and induce a new conceptual way of thinking in established V&V. The emergence of model-based testing can be seen as a first result of such a transfer. However, we believe that to go beyond model-based testing and take a truly model-driven-development approach to V&V would yield even greater benefits. This workshop has fostered a discussion in this direction presenting work related on the following topics:

- The application of V&V using MDA
- The application of 'traditional' V&V *to* MDA
- The integration of testing tools and MDA tools
- The application of V&V to MDA transformations
- The application of 'novel' V&V to MDA
- V&V due to the evolution of models on any level
- The extension of UML in a tool-independent way to allow V&V

October 2006

David Hearnden
Jörn Guy Süß
Nicolas Rapin
Benoît Baudry

Organisers
MoDeV^{2}a 2006

Organization

MoDeV^{2}a 2006 is jointly organized by the the School of Information Technology and Electrical Engineering (ITEE) of the University of Queensland, the Institut National de Recherche en Informatique Automatique (INRIA) and the Commissariat à l'Energie Atomique (CEA).

Organisers

David Hearnden	the University of Queensland
Jörn Guy Süß	the University of Queensland
Nicolas Rapin	le Commissariat à l'Energie Atomique
Benoît Baudry	l'Institut National de Recherche en Informatique Automatique

Program Committee

David Akehurst	University of Kent (UK)
Gabriela Arévalo	Universität Bern (CH)
Thomas Baar	l'Ecole Polytechnique Fédérale de Lausanne (CH)
Fabrice Bouquet	l'Université de Franche-Comté (FR)
Lionel C. Briand	Carleton University, (CA)
Alessandra Cavarra	University of Oxford (UK)
Robert B. France	Colorado State University (US)
Pascale Le Gall	l'Université d'Évry (FR)
Martin Gogolla	Universität Bremen (DE)
Antti Huima	Conformiq (FI)
Bruno Legeard	Leirios Technologies (FR)
Eda Marchetti	Istituto di Scienza e Tecnologie della Informazione "A. Fædo" (IT)
Claudia Pons	Universidad Nacional de La Plata (AG)
Alexander Pretschner	Eidgenössische Technische Hochschule Zürich (CH)
Kerry Raymond	Queensland University of Technology (AU)
Harry Robinson	Google (US)
Bernhard Rumpe	Technische Universität Braunschweig (DE)
Paul Strooper	The University of Queensland (AU)
Jim Steel	IRISA (FR)
Mark Utting	University of Waikato (NZ)
Dániel Varró	Budapesti Muszaki és Gazdaságtudományi Egyetem (HU)

Keynote Address

Jon Whittle George Mason University (US)

Sponsoring Institutions


The University of Queensland


Le Commissariat à l'Energie Atomique

Acknowledgements

We would like to acknowledge the support of Thomas Kühne in guiding us through the inception phase of MoDeV^{2}a 2006. We thank Anna Reggio and Maura Cerioli for their support in organising the venue and Jon Whittle for his willingness to provide an inspiring keynote address. The Australian part of the work was supported by ARC Discovery grant DP0557972: Enhancing MDA with support for verification and validation.

Table of Contents

MoDeV²a 2006

Validating Transformations on Domain-Specific Models 1
Jon Whittle (George Mason University)

Extending the Unified Process with Model-Based Testing 2
Fabrice Bouquet, Stéphane Debricon, Bruno Legeard, Jean-Daniel Nicolet

Towards Model-Driven Unit Testing 16
Gregor Engels, Baris Güldali,, Marc Lohmann

Automated Object's Statechart Generation and Testing from Class Method Contracts 30
Atul Gupta

Using B to verify UML Transformations 46
Kevin Lano

Validation of Model Transformations - First Experiences using a White Box Approach 62
Jochen M. Küster and Mohamed Abd-El-Razik

Towards Verified Model Transformations 78
Holger Giese, Sabine Glesner, Johannes Leitner, Wilhelm Schäfer, Robert Wagner

Model Checking Dynamic and Hierarchical UML State Machines 94
Toni Jussila, Jori Dubrovin, Tommi Junttila, Timo Latvala, Ivan Porres

Static Verification of UML Model Consistency 111
Andrea Baruzzo and Marco Comini

Author Index 127

Validating Transformations on Domain-Specific Models

Keynote Address

Jon Whittle[1]

George Mason University, 4400 University Drive Fairfax, VA 22030-4444, USA,
jwhittle@ise.gmu.edu,
WWW home page: http://ise.gmu.edu/ jwhittle/

Abstract. Model-Driven Software Development brings new hopes and new challenges for the V&V of safety- and mission-critical systems. On the one hand, automated model transformations have the potential to reduce testing effort since a significant amount of code can be auto-generated. On the other hand, for practical systems, the transformations themselves may contain errors and therefore need to be validated. This talk will present work on verifying key properties of auto-generated code by augmenting transformations to output not only code, but also information that can be used to show the correctness of the code. These ideas were implemented in the AUTOFILTER code generator at NASA Ames Research Center. AUTOFILTER generates key portions of flight control software based on a domain-specific model. As an experiment, AUTOFILTER was used to generate code equivalent to that flown on the DEEP SPACE I probe. The talk discusses techniques for validating AUTOFILTER's code generator and will offer hints at future research directions.

Biography

Jon Whittle is an Associate Professor at George Mason University, Fairfax, Virginia. He has a PhD and MSc from the University of Edinburgh and a Bachelors from the University of Oxford. Before returning to academia, he was a research lead at NASA Ames Research Center where he developed and applied new techniques in model-driven software development. Jon is an Associate Editor of the Software and Systems Modeling Journal, Vice Chair of the Steering Committee of the International Conference on Model Driven Engineering, Languages and Systems (MoDELS) and PC member for a number of IEEE/ACM conferences. He has conducted research in artificial intelligence, formal methods, requirements engineering, and software modeling. He has taught software engineering across the world, most notably at India's prestigious Indian Institute of Technology.

Extending the Unified Process with Model-Based Testing

Fabrice Bouquet, Stéphane Debricon, Bruno Legeard, and Jean-Daniel Nicolet

Laboratoire d'Informatique de Franche-Comté (LIFC),
{bouquet,debricon}@lifc.univ-fcomte.fr
Leirios Technologie,
legeard@leirios.com
Centre des Technologies de l'Information de l'état de Genève (CTI),
jean-daniel.nicolet@etat.ge.ch

Abstract. The Unified Process (UP) is a software development technique that includes modeling of specifications and testing workflow. This workflow is achieved by information interpretation of specification to produce manual tests. In this paper, we extend the UP with model-based testing (MBT) where models resulting of the UP will be used for MBT. We describe how model-based testing introduces new test design activities in parallel with the application design activities. We give guidelines to derive the test model from the analysis model produced by the UP. We illustrate this tailored process with the example of a Geneva State taxation.

Key words: Unified Process, Model-Based Testing, Class Diagrams, Statecharts, OCL, Model specialization

1 Introduction

Model-based testing (MBT) exists as a process [1], describing each step needed from creating a test model, automatically generating test cases and executable test scripts from that model. MBT can then be viewed as an independent process in the development life cycle. To make it more efficient and widely used, our purpose is to include MBT in an existing and recognize process, that is to extend the Unified Process (UP) to support model-based testing key features.
The UP is a software engineering process. Its goal is to transform requirements into software. To realize this transformation we model requirements, using for example the Unified Modeling Language (UML). And then models evolve to be more and more precise and finally an implementation of the system is made.
We propose guidelines to specialize model produced by the UP into model suitable for test generation. The new process called UPMBT will then support the usual manual writing of test cases, but also the automatic generation of tests from a new artefact called test model. This is a key concept of UP, the ability to be tailored. The most famous version of the UP is the RUP (Rational Unified

Process), which comes with tools and support. There are also other example of tailored UP such as the EUP (Enterprise Unified Process)[2].
In this paper we define an extension of the UP to include the model-based testing process. In the first section we will introduce the unified process, and the model-based testing process. In the second section we will describe how the UP evolves to support MBT. We also explain benefits and gains expected in the development process. Then we will propose a way to produce a test model from a model provided by the UP. The last section will present a case study based on the life cycle of a taxation in Switzerland.

2 Overview of the Unified Process and Model-Based Testing

In this section we introduce in more details UP and MBT.

2.1 The Unified Process (UP)

We want to consider a software development technique called the Unified Process. The goal of the UP is to define who does what, when and how. The UP is strongly tied to UML [3], but they are actually independent. The UP is the methodology explaining how to transform requirements into software. UML is the visual language necessary to the methodology. Note that UML has been standardized by the OMG. A meta-model exists for both, each of which being an instance of the meta-meta-model named Meta-Object Facility (MOF), defined as the nearly universal root from which several other meta-model derive, like for example XMI (the XML format used to store UML models) and CWM (Common Warehouse Metamodel). The meta-model used for the UP is called Software Process Engineering Metamodel (SPEM).

First research leading to the creation of the UP have been done by Ericsson (the Ericsson approach 1967) and Rational (the Rational Objectory Process, 1996 to 1997) now a part of IBM. In addition it is also based on other best practice and guidelines such as managing requirements, visual modeling of software, component-based architectures, iterative development, verifying software quality and managing changes of software.

The UP is organized into four phases, often with several iterations happening inside some of these phases [4]. The phases are:

Inception: The goal of the inception phase is to have an overview of the overall project; this phase should establish feasibility, from technical prototyping to functional behavior modeling. Stakeholders and project management should end up with an agreement on the scope of the project. Major risks have been identified and a business case expressing how the project delivers benefits to the business is available. The inception phase should evaluate the cost and produce a schedule for the project.

Elaboration: The goal of the elaboration phase is to produce a working skeleton of the architecture of the project. This skeleton will capture some of the most risky functional behavior expressed in the requirements. By the end of this stage, most of the use cases have been written, the hard engineering decisions have been made and the final commitment of whether or not continue with the project can be made.

Construction: The goal of this phase is to build the product. The construction phase is usually a series of development iterations that flesh out the functionality of the skeleton into a deliverable product. Each iteration goes for one to two months, has a fixed deadline and produce a release that can be tested and evaluated by users.

Transition: The goal of this phase is to deliver the finished product to the users. This includes deployment, beta-testing, completion of the user documentation, training of the end users and reacting to user feedback. A project review is done to evaluate the process.

2.2 Model-Based Testing (MBT)

The objective of testing is to find defects in software. To achieve this goal, one must be able to find differences between an implementation and what stakeholders expressed in requirements.

One way to do this is to use model-based testing; it relies on a behavior model, precise enough, of the requirements. The testing model should capture the expected behavior with sufficient abstraction; we don't want to consider every detail.

An implementation should offer an adequate solution to every requirement expressed whether they were formal or not. In fact, if we consider a model as detailed as the System Under Test (SUT), we would rather validate the implementation.

Abstraction is a necessity for model-based testing, but any omission will lead to the impossibility to generate tests for the omitted part. It is a question of choice, for the modeler, whether he wants tests for a requirement or not.

Models should be abstract but precise and complete enough to generate the required test case and the expected results (the test oracle). The models we consider are universal rather than existential abstractions - e.g. state machine diagrams or pre-post conditions as opposed to sequence diagrams [5].

We will consider a template for a process conducting the generation and execution of tests on a SUT as shown in Fig. 1.

Step 1 A model of behavior is deduced from requirements and specification documents. Abstractions will be made at that point; it can range from output abstraction (we don't care about a specific result) to functionality abstraction (a part of the SUT is excluded).

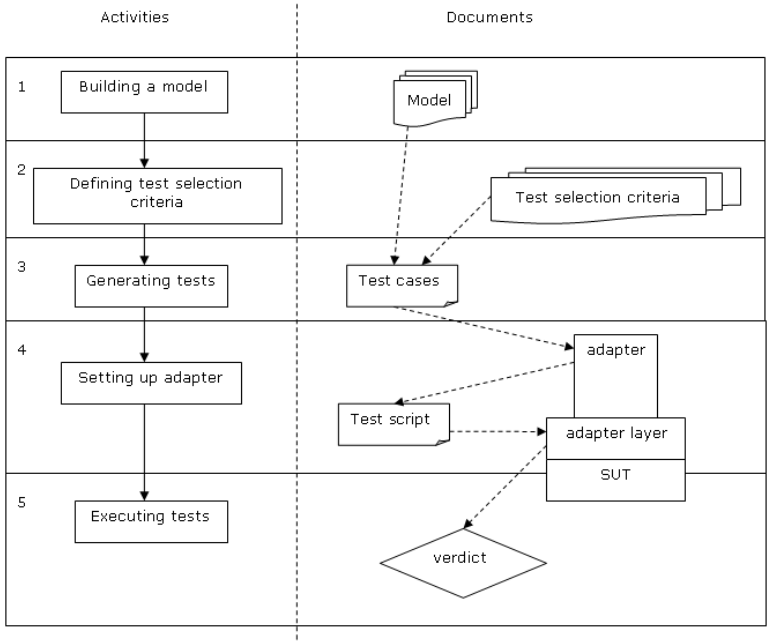


Fig. 1. model-based testing process

Step 2 This step of the process should defined test selection criteria. Arguably, a good test case is one that is likely to detect severe and likely failures at an acceptable cost, and that is helpful at identifying the underlying fault [5]. This definition is too general and does not explicit a way to define relevant test selection criterion. A good practice is to relate test selection criteria to a specific functionality in the SUT. The ability of the testing engineer is of great importance for the choice of criteria regarding the test suite that should be generated.

Step 3 Test selection criteria are then transformed into test case specification. They formalize the notion of test selection criteria and render them operational. The set of test case that satisfy a test case specification can be of any cardinality: it is empty if there is no satisfying test case; usually, they are many test cases that satisfy it.

Step 4 With the testing model and the test case specification, a test suite can be produced. The difference between a test case specification and a test suite is that the further is intentional while the latter is extensional: all tests are explicitly enumerated. This work is done by a test case generator taking the model and the test case specification to produce tests.

Step 5 Tests produced in the previous step will now be executed on the implementation of the SUT. But first we need to add some kind of interface between our test and the implementation. Remember that we made some

abstraction on the model, so we can assume that it will not be possible to execute directly the tests produced. We need to add what is called an *adapter* to bridge those different levels. The goal of this module is to concretize inputs and then to pass them to the implementation and record the output produce by it. Those results will be used to give a *verdict* on the test case. A verdict is the comparison result of the output of the SUT with the expected output as provided by the test case. There are 3 distinct verdicts: pass, fail and inconclusive. A test passes if outputs defined by the test case and produced by the SUT are equivalent. It fails if they are different. The inconclusive verdict applies to non determinist system.

3 Extending the Unified Process with Model-Based Testing

In this section we will extend the UP to support a MBT process, and we describe what benefits could be expected.

3.1 Evolutions in the UP phases

As we introduce MBT in the UP, we also have to describe what impacts can be expected on phases of the UP. The inception phase is not impacted by this approach. Changes mainly concern the elaboration and the construction phase. During those phases the system will grow from a prototype to a final product and the associated tests artifacts will evolve with iterations. The elaboration phase will produce a test model for basic functionalities, an adapter and an adapter layer. By the end of the phase, those artifacts should be mature enough and the adapter layer architecture should be precisely defined. The focus for this phase is on the structure of the adapter layer that should be able to map the implementation.

During the construction phase, artifacts will only evolve with the addition of new functionality that might produce new question to be elicited.

Each iteration made within the elaboration and construction phase will augment the number of test suite. System testers and users will have a complete testing framework and suitable tests.

Benefits of MBT will be the production of corrections in the analysis and design model earlier in the process. Following iterations introducing them in the implementation model. More important a defect might be found before the effective execution of a test. Manual and generated test scripts, produced by the entire process, will be executed in the transition phase. They will be included to the beta-testing for the software.

3.2 Evolutions in the UP workflows

We consider a typical construction iteration of the UP as described earlier. We propose to introduce the MBT activities right after the UP analysis workflow

that is in the design process. From the MBT viewpoint, the first activity is the production of a test model. This requires the analysis model to be available.

From this point on, we extend the UP by adding a new artefact for the design workflow, so that two different kinds of model are produced. The first model is the usual design model of the UP, while the second one is a test model. Both are derived from the analysis model and the requirements documents. The design workflow needs to be divided into two parallel processes. A development team will take in charge the execution of the usual design process and a test team will follow a model-based testing process. Those two point of view on the same analysis model may lead to deviation between models. This will be the result of a different or partial interpretation of specification present in the analysis model.

The unified process recommends assigning a team that will go through the entire process for a given functionality, rather than splitting team for a given activity. For testing activity it's a good practice to have a dedicated team, first because they must have specific skills and to be relevant a testing model must be independent enough. Obviously a team will be made of workers endorsing a role depending of the workflow or the phase. Team independence during this workflow is crucial. Any question raised while modeling for test, should be elicited by the system analyst or a stakeholder but not by the development team. A good independence ensures a good accuracy of the generated tests. This parallel process is shown in Fig. 2.

The next workflow in the UP is the implementation. The development team will have to produce an implementation for the design model. Meanwhile the test team will have to produce an adapter to interact with the SUT, that is to implement the generated test cases into executable test scripts. Both developments will occur approximately at the same time and they must be carefully coordinated. The interface must have been well defined for test scripts to interact with the SUT. They are called control and observation points. They must be available for the test team to apply scripts and get output from the SUT.

The test workflow needs to be described for each team with more details. This workflow will gather, for the development team, all activities of usual testing; that are: creating test case, evolving existing test scripts, running tests and reporting any defects detected.

The goal of this workflow, for the test team, is to generate test scripts from the adaptation layer and the generated test cases. Test cases have been generated during the previous activity using a MBT test generator. Any changes made on the model during the design phase will be integrated to the newly generated test scripts.

The next activity will be the execution of those tests under the system. Defects will not be reported directly as they can be produced by the model or the implementation (or both). A control of the model must be done before reporting the defect. If the failure is proved to be relative to the implementation, then a good diagnostic can be expressed (issued from our test cases).

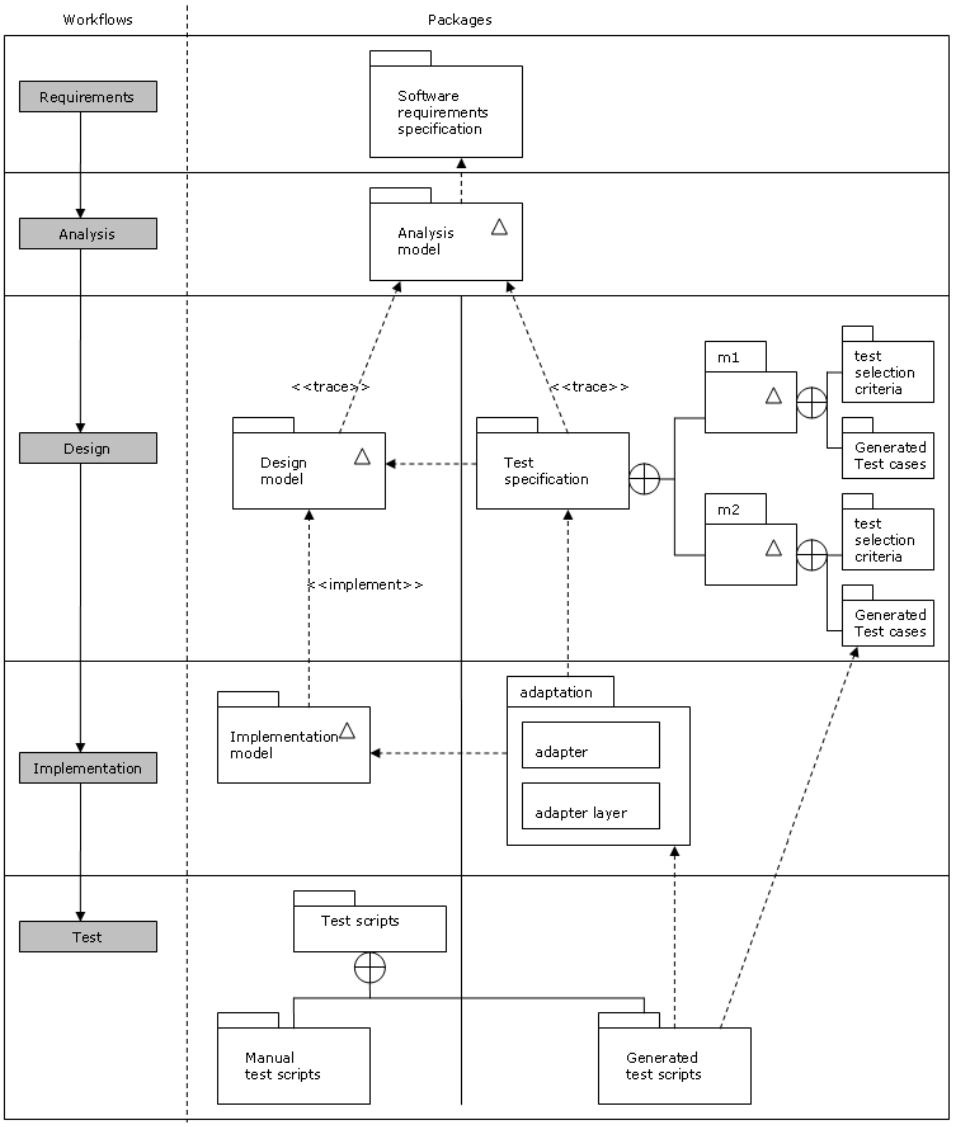


Fig. 2. Packages produced by the UP extended with MBT

3.3 Specializing an analysis model into a test model

We mentioned earlier that UML can be used as the modeling language for the unified process. It can also be used to generate tests (e.g. [6]), various works have been done [7],[8],[9],[10],[11] and they all have in common the subset of UML used. Models should represent the system structure and its expected behavior.

The analysis model reflects the system structure with class diagrams. Those diagrams contain classes, with attributes and operations, and relations between classes. The system behavior will be captured by statecharts and sequence diagrams. The first one is made of states and transitions linking them. And the second one expresses messages exchange between actors.

An analysis model may contain unnecessary informations (we probably does not want to test every functionality of the system) and in the same time be lacking of crucial informations in regards of our testing process. Those reasons explicit why we need to specialize our analysis model into a new one, the test model.

A test model derives from an analysis model. That means that classes and relations (from class diagrams), states and transitions (from statecharts) will be re-used in our test model. However those items cannot be used directly because abstraction is a necessity when modeling for test on a real system [12]. This is precisely the step 1 of the process described in section 2.2

Models should not contain every classes of the analysis model, only a subset of classes representing the function to be tested should appear. But on the other side, we need to add information to the test model. The goal of the model is to produce test based on what is expressed. The analysis workflow does not produce a model complete enough. A model should contain class operation with a minimum amount of information about the system behavior. The description can be placed on class diagrams, using pre- and post- conditions, or in state machines using transition effects.

We suggest the use of OCL [13] to express those informations. OCL is a formal language, non ambiguous and is part of the UML2.0 specification. Any relevant information expressed in an informal way on the analysis model should be specified using the OCL notation in diagrams used for tests.

4 Case Study: RTAXPM

To illustrate the need for specialization of an analysis model, we will introduce in this section a case study. This is a tax record life cycle of the Geneva State AFC (fiscal administration). The AFC is using RUP in combination with IBM Rational tools, which gives us a nearly complete and well structured model of the system.

4.1 A taxation life cycle

The functionality under test for this case study is a taxation life cycle. We only focus on the analysis model, the life cycle is described using three kind of diagram

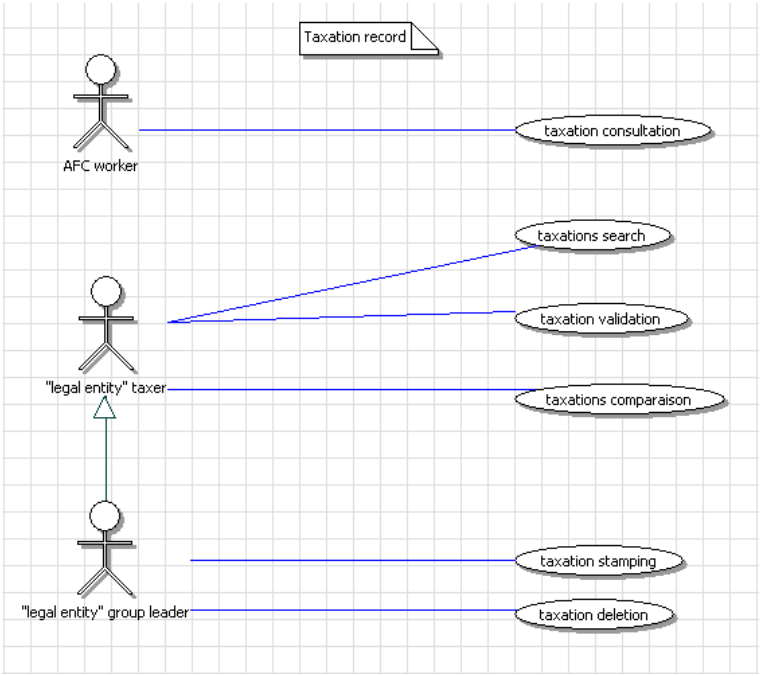


Fig. 3. Use cases (extract from the model).

in the AFC model: use case diagram (Fig. 3), statechart diagram (Fig. 4) and class diagram.

Our purpose is to generate tests from this model. We will use all provided diagrams to produce a model suitable for test. All informal informations will be added on diagrams using OCL. Taking into account the type of diagrams and the formal language used, its availability at the LIFC, we choose to generate test cases with the MBT test generator LEIRIOS Test Generator for UML (LTG)[1]. The test model have been made with Borland Together Designer 2006 for Eclipse, then exported and processed by LTG.

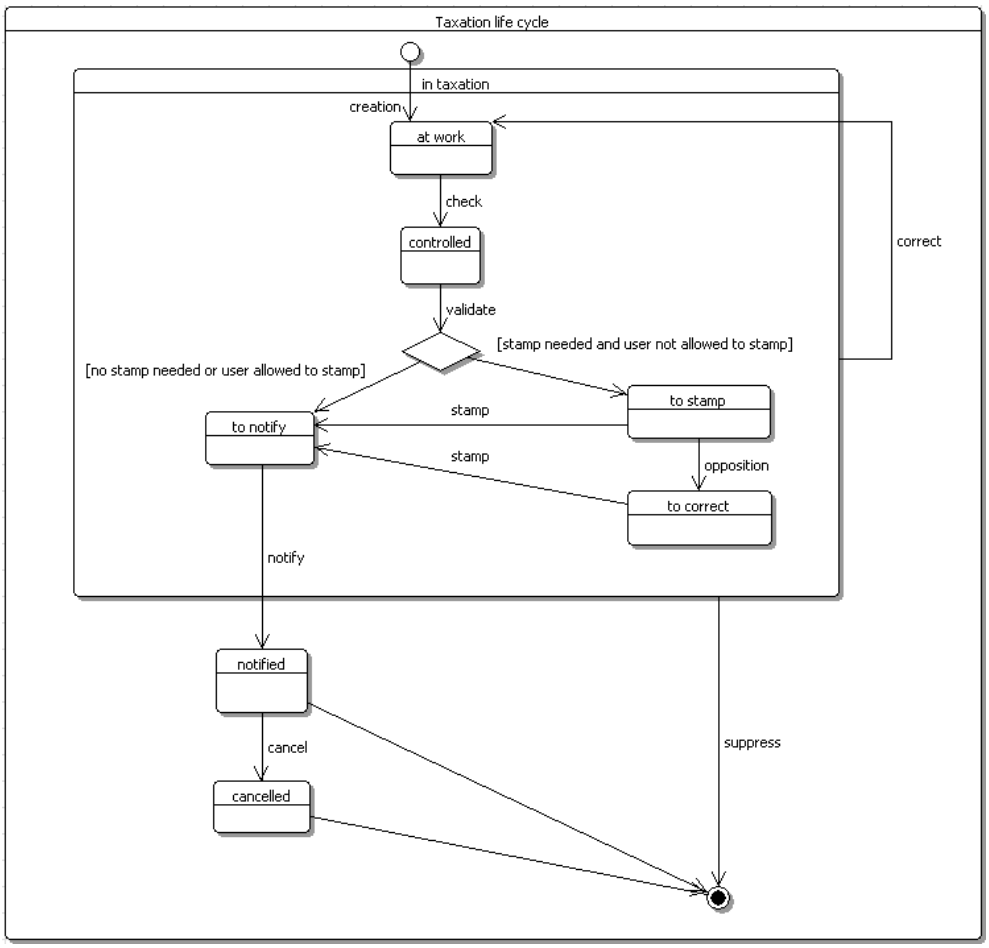


Fig. 4. Taxation life cycle (extract from the model).

[1] For more information about the model-Based testing tool LTG, see www.leirios.com

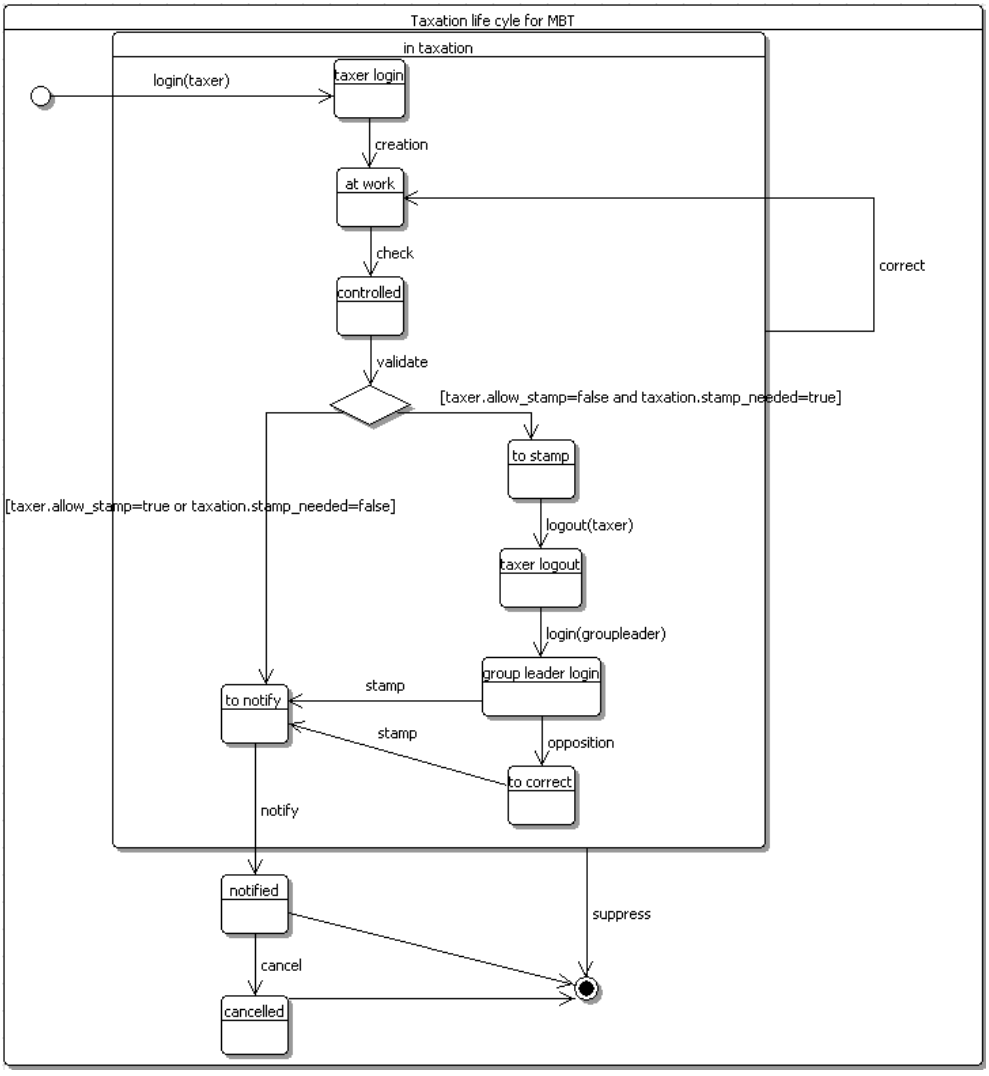


Fig. 5. Taxation life cycle for MBT.

4.2 Specializing the AFC analysis model into a new test model

In this section we will describe all changes induced by the specialization.

- We add '*taxer*' and '*group leader*' classes. Those classes appear in the model as actor (c.f. Fig. 3), and are issued of the security policy defined for the application framework. This information was clarified by the use case diagram and by an explanation of the project manager. In addition the use case explicit that a group leader is the only taxer being able to stamp or suppress a taxation. This will be introduce in the statechart diagram by new states, representing a login and logout from the system.
- We transform informal notation such as 'user allowed to stamp' and 'stamp needed', into formal notation:
  First the ability for stamping is defined as an attribute of *taxer* class.
  Added to this, the need for stamp is more complex to introduce. A use case (not included in this paper) describes in an informal way, how to define if a stamp is needed on a taxation. It depends on computation on a taxation record data. This rises a problem of choice:
  - we can either put computation rules in the model, it allows the use of exact guard on transitions.
  - or we can use a boolean-like value defining if a stamp is needed or not, and leave the computational work to the adaptation layer.

  We choose to use a boolean value. We place the following OCL guards in the test model:
  `[taxer.allow_stamp=true or taxation.stamp_needed=false]` and
  `[taxer.allow_stamp=false and taxation.stamp_needed=true]`
- We retain all attributes of the analysis class in our '*taxation*' class. They will not be all used for the test of the life cycle but can be useful for other test generation.
- We add, in our '*taxation*' class, operation needed for the transition on the statechart diagram. They will capture the behavior of our taxation life cycle. It is described by a taxation status (e.g. 'in taxation', 'controlled' or 'to stamp'). The taxation status will be our control point for the test oracle.

Figure 5 is the model produced by the specialization of an analysis model into a test model.

4.3 Generated test cases for the taxation life cycle

For each test target, the LTG tool computes, if possible, one or more test cases. We set up the tool with a depth-first algorithm and a search depth of 10. For our case study, 16 test cases were produced covering the 9 operations of a taxation.

Figure 6 is a screen shot of a test case example for the '*stamp*' operation and its oracle. This test case is transformed into a test script using the adapter. The produced test script is then ran on the AFC framework through the adaptation layer. The result returned by the framework is compared to the oracle which gives the verdict of our test case.

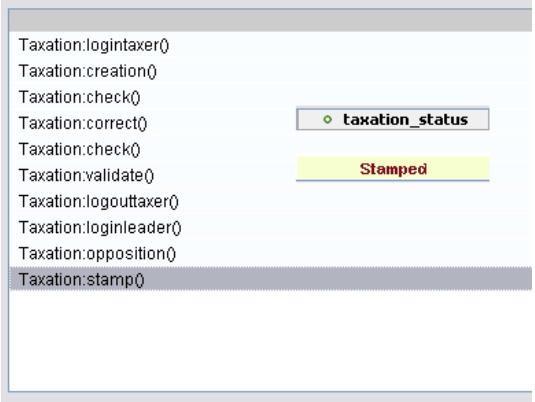


Fig. 6. Test cases for the stamp operation.

5 Conclusions

In this paper, we propose to extend the UP with MBT activities. This allow the automatic generation of validation test cases from a test model that inherits from the analysis model. Then, adapters are developed to be used for generating executable test scripts from the abstract generated test cases.

Including MBT into a software engineering process will accelerate its adoption, giving industrials a defined framework to introduce tests from models in their development process. One of the great difficulty of MBT, so far, is to keep the testing model alive. Specifications or design model evolve during a development life cycle and the test model will be useless if not modified. This is where the iterative feature of UP ensure that both models are at an identical state.

The chosen case study at the AFC gives us the first support for this approach, but we need to explore links between analysis, test and design models.

This UP process extended with MBT has been used in the real context of the development of a taxation system at the Geneva State (Switzerland).

The main benefits of the integration of MBT activities into the UP are the reusability of the analysis models to develop the test model, and the automatic generation of validation test cases for the considered application functionalities. Moreover, the development of the test model, which implies to formalize the expected behavior of the system under test, leads to a direct feedback on the analysis models (and design models) that help their validation.

This integration of MBT activities in well recognized and used software development process is a key issue for the dissemination of the MBT approach in the software development practices.

Acknowledgments. The authors would like to thank the CTI and Mark Utting for their participation.

References

1. Utting, M., Legeard, B.: Practical Model-Based Testing - A tools approach. Elsevier Science (2006) 550 pages, ISBN 0-12-372501-1. To Appear.
2. Ambler, S.W.: Introduction to the Enterprise Unified Process. www.enterpriseunifiedprocess.com. (2005)
3. Group, O.M.: UML Resource Page. http://www.uml.org (2005)
4. Arlow, J., Neustadt, I.: UML 2 and the Unified Process, Second Edition, Practical Object-Oriented Analysis and Design. Addison-Wesley (2003)
5. Utting, M., Pretschner, A., Legeard, B.: A Taxonomy of Model-Based Testing. Technical report 04/2006, Department of Computer Science, The University of Waikato (New Zealand) http://www.cs.waikato.ac.nz/pubs/wp/2006/uow-cs-wp-2006-04.pdf (2006)
6. Offutt, J., Abdurazik, A.: Generating Tests from UML Specifications. In France, R., Rumpe, B., eds.: UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings. Volume 1723., Springer (1999) 416–429
7. Kansomkeat, S., Rivepiboon, W.: Automated-generating test case using UML statechart diagrams. In: SAICSIT '03: Proceedings of the 2003 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology, , Republic of South Africa, South African Institute for Computer Scientists and Information Technologists (2003) 296–300
8. Basanieri, F., Bertolino, A., Marchetti, E., Ribolini, A., Lombardi, G., Nucera, G.: An Automated Test Strategy Based on UML Diagrams. In: Ericsson Rational User Conference, Upplands Vasby Sweden. (2001)
9. Hartmann, J., Imoberdof, C., Meisenger, M.: UML-Based Integration Testing. In: ISSTA. (2000)
10. Briand, L.C., Labiche, Y.: A UML-Based Approach to System Testing. In: Proceedings of the 4th International Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools, London, UK, Springer-Verlag (2001) 194–208
11. Kim, Y., Hong, H., Cho, S., Bae, D., Cha, S.: Test cases generation from UML state diagrams. In: IEEE Software vol. 146. (1999) 187–192
12. Prenninger, W., Pretschner, A.: Abstractions for Model-Based Testing. Theoretical Computer Science (2005)
13. Warmer, J., Kleppe, A.: The Object Constraint Language, Second Edition, Getting your models ready for MDA. Addison-Wesley (2003)

Towards Model-Driven Unit Testing

Gregor Engels[1,2], Baris Güldali[1], and Marc Lohmann[2]

[1] Software Quality Lab
[2] Department of Computer Science
University of Paderborn, Warburgerstr. 100, 33098 Paderborn, Germany
engels@upb.de, bguldali@s-lab.upb.de, mlohmann@upb.de

Abstract. The Model-Driven Architecture (MDA) approach for constructing software systems advocates a stepwise refinement and transformation process starting from high-level models to concrete program code. In contrast to numerous research efforts that try to generate executable function code from models, we propose a novel approach termed *model-driven monitoring*. On the model level the behavior of an operation is specified with a pair of UML composite structure diagrams (visual contract), a visual notation for pre- and post-conditions. The specified behavior is implemented by a programmer manually. An automatic translation from our visual contracts to JML assertions allows for monitoring the hand-coded programs during their execution.
In this paper we present an approach to extend our model-driven monitoring approach to allow for *model-driven unit testing*. In this approach we utilize the generated JML assertions as test oracles. Further, we present an idea how to generate sufficient test cases from our visual contracts with the help of model-checking techniques.

1 Introduction

Everyone who develops or uses software systems knows about the importance of software qualities, e.g. correctness and robustness. However, the growing size of applications and the demand for shorter time-to-market hampers the development of high-quality software systems. To get a better handle on the complexity, the paradigm of model-driven development (MDD) has been introduced. In particular, the Object Management Group (OMG) favored a model-driven approach to software development and pushed its Model-Driven Architecture (MDA) [1] initiative as well as standards such as the Unified Modeling Language (UML) that provides the foundation for MDA.

However, the MDA is still in its infancy compared to its ambitious goals of having a (semi-)automatic, tool-supported stepwise refinement process from vague requirements specifications to a fully-fledged running program. A lot of unresolved questions exist for modeling tasks as well as for automated model transformations.

Nevertheless, in today's software development processes models are an established part for describing the specification of software systems. In principle,

models provide an abstraction from the detailed problems of implementation technologies. They allow software designers to focus on the conceptual task of modeling static as well as behavioral aspects of the envisaged software system. Unfortunately, abstraction naturally conflicts with the desired automatic code generation from models as proposed by the MDA. To enable the latter, fairly complete and low-level models are needed. Today, a complete understanding of the appropriate level of detail and abstraction of models is still missing. Thus, in today's software development processes developers are normally building an application manually with respect to its abstract specification with models.

In our work, we introduced a new modeling approach. We do not follow the usual approach that models should operate as source for an automatic code generation step that produces the executable function code of the program. Rather, we restrict the modeling task to providing structural information and minimal requirements towards behavior for the subsequent implementation. We expect that only structural parts of an implementation are automatically generated, while the behavior is manually added by a programmer.

As a consequence it can not be guaranteed that the hand-coded implementation is correct with respect to the modeled requirements. Therefore, we have shown in previous publications [2–4] how models can be used to generate assertions which monitor the execution of the hand-coded implementation. Herewith, violations of the modeled requirements will be detected at runtime and reported to the environment. We call this novel approach *model-driven monitoring*.

Model-driven monitoring is based on the idea of Design by Contract (DbC) [5], where so-called contracts are used to specify the desired behavior of an operation. Contracts consist of pre- and post-conditions. Before an operation is executed, the pre-condition must hold, and in return, after the execution of an operation, it has to be guaranteed that the post-condition is satisfied. The DbC approach has been introduced at the level of programming languages. For instance, the Java Modeling Language (JML) extends Java with DbC concepts [6]. JML assertions are based on Java expressions and are annotated to the source code. During the execution of such an annotated Java program, the assertions are monitored. An exception is raised as soon as a violation of the assertions is detected.

With the concepts of *visual contracts* [2] we have lifted the idea of contracts to the level of models. A visual contract allows for specifying a contract by pairs of UML composite structure diagrams for the pre- and post-conditions. A transformation of our visual contracts into JML allows for monitoring a system that is implemented manually.

Now we want to extend our approach to allow for *model-driven unit testing*. The visual contracts respectively the generated JML assertions are viewed as test oracles to decide whether the results calculated by a hand-coded implementation are correct. Additionally, we want to generate test cases from our models with the help of model-checking techniques.

This paper is organized as follows: the following section gives an overview of our approach. In Sect. 3 the visual contracts are explained. Section 4 shortly

explains how to generate assertions from visual contracts. Section 5 describes our testing approach on a conceptual level. In Sect. 6 we give an overview of available tools that can be used for automation of our testing approach. In Sect. 7 we discuss related work and finally we conclude the paper.

2 Enabling Model-Driven Unit-Testing with Model-Driven Monitoring

Model-driven monitoring [2–4] constitutes a novel strategy for model-driven software development beyond the classical idea of model-driven development centered upon the automatic generation of function code. Model-driven monitoring lends itself to both model-driven and agile software development methods. We enable model-driven monitoring by embedding visual contracts in a model-driven software development process. Visual contracts are interpreted as models of behavior from which code for runtime assertion checking can be generated. The visual contracts also specify the behavior which is then manually implemented by programmers.

Test-driven development [7] is an important part of agile processes. E.g. Extreme Programming (XP) [8], the most prominent of several agile software development processes, emphasizes the test-first approach. When handling a programming task, programmers always begin writing unit tests. This test formalizes the requirements. If all tests run successfully then the coding is complete. To accent the agile part of our model-driven monitoring approach we want to support the test-driven development by enabling model-driven unit testing. Therefore, beside the generation of runtime assertions we want to automatically generate test cases from our models. Figure 1 shows our model-driven software development process enabling model-driven monitoring and model-driven unit testing.

On the design level, a software designer has to specify a model of the system under development. This model consists of class diagrams and visual contracts. The class diagrams describe the static aspects of the system. Each visual contract specifies the behavior of an operation. The behavior of the operation is given in terms of data state changes by pre- and post-conditions, which are modeled by a pair of UML composite structure diagrams as explained in Sect. 3.

In the next step, we generate code fragments from the design model. This generation process consists of two parts. First, we generate Java class skeletons from the design class diagrams. Second, we generate JML assertions from every visual contract and annotate each of the corresponding operations with the generated JML contract. The JML assertions allow us to check the consistency of models with manually derived code at runtime. The execution of such checks is transparent in that, unless an assertion is violated, the behavior of the original program remains unchanged.

Then, a programmer uses the generated Java fragments to fill in the missing behavioral code in order to build a complete and functional application. His programming task will emanate from the design model of the system. Particularly,

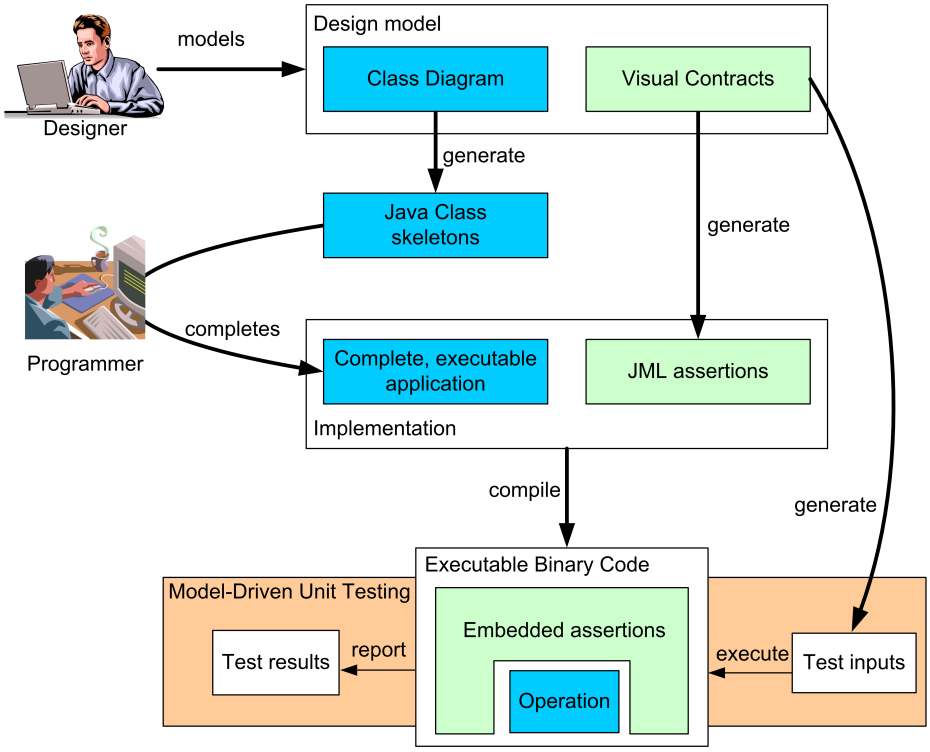


Fig. 1. Overview of the testing approach

he will use the visual contracts as reference for implementing the behavior of operations. He has to code the method bodies, and may add new operations to existing classes or even completely new classes, but he is not allowed to change the JML contracts. If new requirements for the system demand new functionality then the functionality has to be specified with visual contracts before the programmer can start programming. Using our visual contracts this way in a software development process resembles Agile Development and Extreme Programming approaches, where a developer has to specify a set of test cases before implementing the code.

When a programmer has implemented the behavioral code, he uses the JML compiler to build executable binary code. This binary code consists of the programmer's behavioral code and additional executable runtime checks which are generated by the JML compiler from the JML assertions. The manual implementation of a programmer leads to system state changes. The generated runtime checks monitor the pre- and post-conditions during the execution of the system. Thus, we support model-driven monitoring of implementations by transforming our visual contracts into contracts in JML.

To further integrate agile respectively extreme programming approaches in our model-driven software development process we additionally want to integrate *model-driven unit testing* in our development process. Therefore, we have to address the following three problems of model-driven testing [9]:

1. the generation of test cases from models,
2. the generation of a test oracle to determine the expected results of a test,
3. the execution of tests in test environments.

The basic idea of our testing approach is that the specification of an operation by a pre- and post-conditions (visual contract) can be viewed as a test oracle [10, 11] and runtime assertion checking can be used as a decision procedure for a test oracle. That means the runtime checks generated by the JML compiler can be used as test oracles. Thus, our visual contacts can be viewed as test oracles since the JML assertions are generated from our visual contracts. On the code level, this idea is supported by the tool JMLUnit which combines JML with the popular unit testing tool JUnit for Java. Still, we need to answer the problem of how to generate test cases from models. Therefore, we want to combine well-known testing techniques for the generation of test input parameters and model checking to be able to create concrete system states. The idea how to create test cases is described in detail in Sect. 5.1.

3 Modeling with Visual Contracts

We show how to specify a system with visual contracts by the example of an online shop. We distinguish between a static and a functional view.

UML class diagrams are used to represent the *static view* of a system specification. Figure 2 shows the class diagram of the sample online shop. We use the stereotypes `control` and `entity` as introduced in the Unified Process [12]. Each of these stereotypes expresses a different role of a class in the implementation. Instances of control classes encapsulate the control related to a specific use case and coordinate other objects. Entity classes model long-lived or persistent information. The stereotype `key` indicates key attributes of a class. A key attribute is a unique identifier for a set of objects of the same type. The control class `OnlineShop` is connected to the entity classes of the system via qualified associations. A rectangle at an association end with a qualifier (e.g. `productNo`) designates an attribute of the referenced class. In combination with the key-attributes of a class, the qualifier allows us to get direct access to a specific object.

Class diagrams are complemented by visual contracts that introduce a *functional view* integrating static and dynamic aspects. Visual contracts allow us to describe the effects of an operation on the system state of the system. Thus, for our visual contracts we take an operation-wise view on the internal behavior.

In the following, we want to explain our visual contracts by two examples. The operation `cartCreate` of the control class `OnlineShop` creates a new cart.

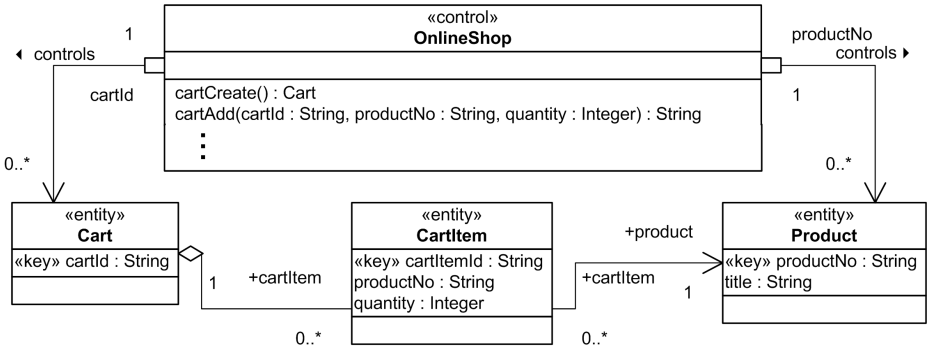


Fig. 2. Class diagram specifying static structure of online shop

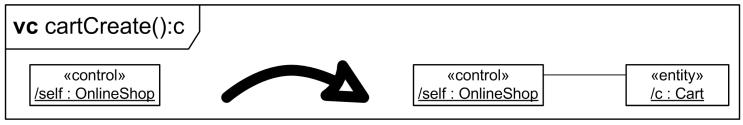


Fig. 3. Visual contract for operation *cartCreate*

Figure 3 shows a visual contract that describes the behavior of the operation. The visual contract is enclosed in a frame, containing a heading and a context area. The keyword `vc` in the heading refers to the type of diagram, visual contract in this case. The keyword is followed by the name of the operation that is specified by the visual contract. The operation name is followed by a parameter-list and a return-result if they are specified in the class diagram. The parameter-list is an ordered set of variables and the return-result is also a variable. The variables of the parameter-list and the return-result are used in the visual contract.

The visual contract is placed in the context area. Structurally, a visual contract consists of two graphs, representing the pre-condition and the post-condition of an operation. The graphs are visualized by UML composite structure diagrams [13]. Each of the graphs is typed over the design class diagram. The semantics of our visual contracts is defined by the loose semantics of open graph transformation systems [14]. The basic intuition for the interpretation of a visual contract is that every model element, which is only present on the right-hand side of the contract, is newly created, and every model element that is present only on the left-hand side of the contract, is being deleted. Elements that are present on both sides are unaffected by the contract. Additionally, we may extend the pre- or post-condition of a visual contract by negative pre-conditions (i.e., negative application conditions [15]) or respectively by negative post-conditions. A negative condition is represented by a dark rectangle in the frame. If the dark rectangle is on the left of the pre-condition, it specifies object structures that

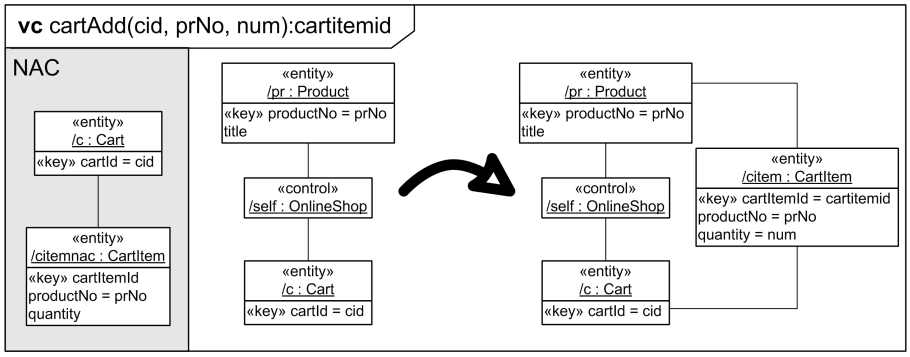


Fig. 4. Visual contract for operation *cartAdd*

are not allowed to be present before the operation is executed (see Fig. 4). If the dark rectangle is on the right of the post-condition, it specifies object structures that are not allowed to be present after the execution of the operation.

The contract as described in Fig. 3 expresses that the operation `cartCreate` can always be executed, because the pre-condition only contains the model element `self`, i.e. the object executing the operation. As an effect, the operation creates a new object of type `Cart` and a link between the object `self` and the new object of type `Cart`. Additionally, the object `c:Cart` is the return value of the operation `cartCreate` as indicated by the variable `c` used in the heading.

Figure 4 shows a more complex contract specifying the operation `cartAdd`. This operation adds a new `CartItem`, which references an existing `Product`, to an existing `Cart`. In contrast to the visual contract of Fig. 3, the variables of the parameter-list and the return-value are now used to specify values of attributes of different objects. For a successful execution of the operation, the object `self` must know two different objects with the following characteristics: an object of type `Cart` that has an attribute `cartId` with the value `cid`, and an object of type `Product` that has an attribute `productNo` with the value `prNo`. The concrete argument values are bound when the client calls the operation. The `Cart` object is reused in the negative pre-condition (compare object identifiers). The negative pre-condition extends the pre-condition by the requirement that the `Cart` object is not linked to any object of type `CartItem` that has an attribute `productNo` with the value `prNo`. This means, it is not permitted that the product is already contained in the cart. As a result, the operation creates a new object of type `CartItem` with additional links to previously identified objects. The return value of the operation is the content of the attribute `cartItemId` of the newly created object.

4 Translation to JML

After describing the modeling of a software system with visual contracts, we now present how the model-driven software development process continues from the design model. A transformation of visual contracts to JML constructs provides for model-driven monitoring of the contracts. The contracts can be automatically evaluated for a given state of a system, where the state is given by object configurations. The generation process as well as the kind of code that is generated from a class diagram and the structure of a JML assertion that is generated from a visual contract are described in detail in [2, 4]. Here we only describe the transformation more generally and from a methodical perspective.

4.1 Transformation of Class Diagrams to Java

Each UML class is translated to a corresponding Java class. Attributes and associations are complemented by the corresponding access methods (e.g., `get`, `set`). For multi-valued associations we use classes that implement the Java interface `Set`. Qualified associations are provided by classes that implement the Java interface `Map`. We add methods like `getProduct(int productNo)` that use the attributes of the qualified associations as input parameters. Operation signatures that are specified in the class diagram are translated to method declarations in the corresponding Java class up to syntactic modifications according to the Java syntax.

4.2 Transformation of Visual Contracts to JML

For each operation specified by a visual contract, the transformation of the contract to JML yields a Java method declaration that is annotated with JML assertions. The pre- and post-conditions of the generated JML assertions are interpretations of the graphical pre- and post-conditions of the visual contract. When any of the JML pre- and post-conditions is evaluated, an optimized breadth-first search is applied to find an occurrence of the pattern that is specified by the pre- or post-condition in the current system state. The search starts from the object `self` (object `this` in Java syntax) which is executing the specified behavior (compare [16]). If the JML pre- or post-condition finds a correct pattern, it returns true, otherwise it returns false.

5 Test Case Generation and Test Execution

In the previous sections we explained how a software designer develops a design model and how Java class skeletons and JML assertions can be generated from them. We also explained how a programmer can complete the generated code fragments to build a complete executable application. After these steps we want to test our application. In Sect. 2 we explained the three tasks of model-driven

testing. In this section we will explain how we handle the first and the third task, i.e. the generation of test cases and the execution of a test. The second task (the generation of a test oracle) is described in Sect. 4 since we can interpret the JML assertions as test oracles.

Similar to classical unit-testing, our test items are operations. The behavior of an operation is dependent of the input parameters and the system state. Thus, a test case has to consider the parameter values of an operation and a concrete system state.

5.1 Test Case Generation

A test case for an operation consists of concrete parameter values and a concrete system state. We can generate a test case for an operation from our model in three successive steps. In the following, we explain how to generate a sample test case for the operation `cartAdd` (Fig. 4). Figure 5 illustrates the three steps.

In the first step, we generate values for the input parameters of an operation as specified in the class diagram. In Fig. 5 we generated the parameter values for the operation `cartAdd` randomly. For the parameter `cid` of type `String` the value "abc" is generated. The parameter `prNo` of type `String` gets the value "def" and the variable `num` of type `Integer` gets the value "1". Beside a the random generation of input parameters, we could also use other techniques for test data generation, e.g. equivalence-class partitioning or boundary value analysis (see e.g. [17]).

To generate a sufficient system state for testing, we have to execute two further steps. Since the visual contracts specify system state requirements, we use them as source for generating the system states. Therefore, we initialize the pre-condition of a visual contract with the parameter values generated in step one. The variables in the parameter-list are used to restrict the attribute values of objects in the pre-condition as explained in Sect. 3. Thus, the initialization gives an object structure. In this object structure some of the attributes have concrete values. Figure 5 shows how the attributes `productNo` and `cartId` of the classes `Product` and `Cart` are initialized with the parameter values of step one according to the pre-condition in Fig. 4. It is important to notice that this object structure describes a system state only partially.

In the last step of our test case generation, we have to find out how to generate a system state which contains the object structure found in step two. Due to the fact that the object structure in the previous step defines a system state only partially, we cannot just build a system state by creating the known objects and attribute values. Such a system state would be incomplete and it would be artificial in a sense that the application would never create such a system state at runtime. Additional objects or attribute values can be created during the execution of the systems at runtime and these may have side-effects on the execution of an operation. Thus, tests should work on realistic system states. To avoid these artificial system states it would be useful to build a system state by using the control operations of the system itself. We assume that each

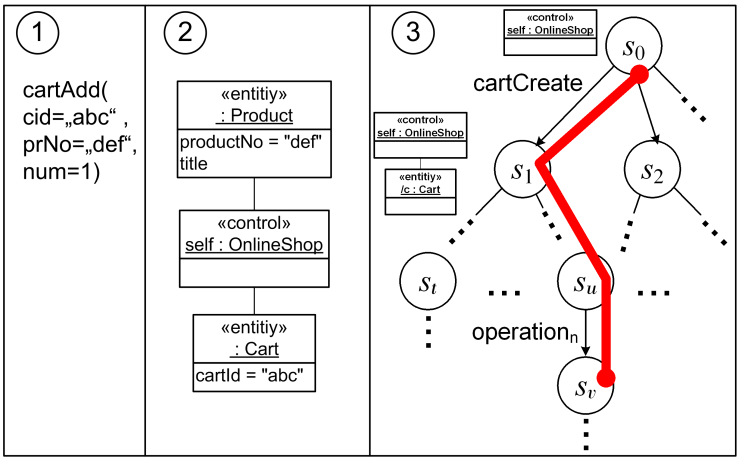


Fig. 5. Three steps of test case generation

operation call leads to a state change of the system. Thus, we have to find a sequence of operation calls that starting from the initial state lead to a sufficient system state which contains the object structure found in step two. As a visual contract describes the system state change of an operation, we can use these contracts to compute all possible states of the system. Therefore, we consider a system state as a *graph* and the visual contracts constitute *production rules* of a *graph transition system*. Figure 5 illustrates how we want to generate a transition system. Initially the system state comprises just an instance (`self`) of the controller class `OnlineShop`. Executing, e.g., the operation `cartCreate` makes the in Fig. 3 specified changes on the system state. Thus, a new object of type `Cart` is generated and linked to the control object `self`. Executing further operations brings the system to a state s_v which contains the object structure generated in step two. Knowing all visual contracts and an initial state, we can compute the graph transition system and search for a production sequence that creates a system state which contains the object structure found in step two. These computations can be done automatically with *model checking* techniques [18]. The computed production sequence directly refers to an operation sequence which brings the system state to some desired state containing the object structure computed in step two. If no sufficient production sequence is found in the graph transition system (the searched object structure cannot be constructed using the existing operations), our test case generation approach has to backtrack to step one and generate other test data.

5.2 Test Execution with Embedded Oracles

After test cases are generated, the test execution can start. Test execution comprises two main steps as shown in Fig. 6. First, the operation sequence deter-

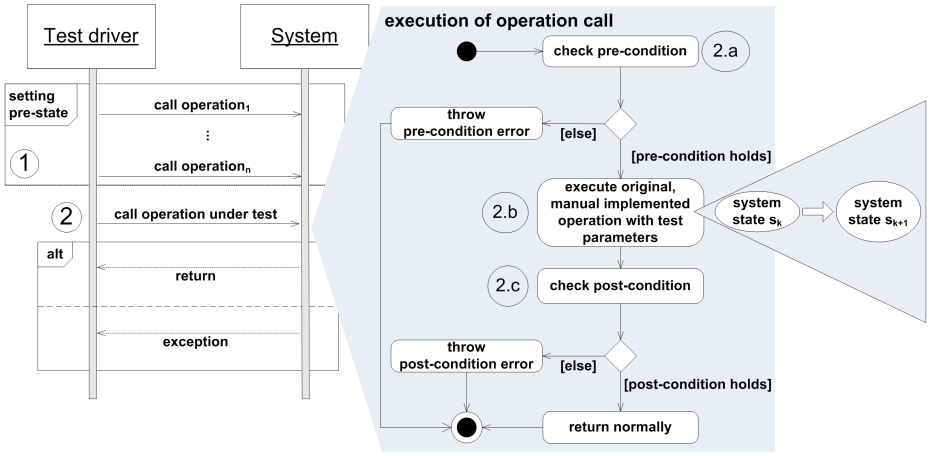


Fig. 6. Run-time behavior of test execution

mined by the test case generation must be executed in order to set the system state. Second, the operation under test is called with the test input parameters also generated by the test case generation.

The embedded assertions lead to a run-time behavior of an operation call as shown in Fig. 6. When the operation under test is called, a pre-condition check method evaluates the method's pre-condition and throws a pre-condition violation exception if it does not hold. If the pre-condition holds, then the original, manually implemented operation is invoked. After the execution of the original operation, a post-condition check method evaluates the post-condition and throws a post-condition violation exception if it does not hold. If the embedded assertions throw an exception then the implementation does not behave according to its specification. Thus, we have found an error.

6 Tool Support

Most of the steps of our approach can be supported by tools. In former publications we have reported on our Visual Contract Workbench, an integrated development environment for using visual contracts in a software development process [19]. This development environment allows software designers to model class diagrams and specify the behavior of operations by visual contracts. It further supports automatic code generation as described in Sect. 4.

The most challenging task of our test generation approach is finding an operation sequence for setting a system state as explained in Sect. 5.1. This task can be automatically solved by *model checking* tools. A candidate for our purposes is GROOVE [20], a model checker for attributed graph transition systems. For that, we consider to interpret our visual contracts and our test inputs as graph

transformation rules in GROOVE. A reachability analysis can show whether a system state containing the object structure in the test input can be reached in the graph transition system. If this is true, we expect GROOVE to supply us with a *witness path*, representing the desired operation sequence.

The test execution can be implemented by a test driver (see Fig. 6). The test driver will execute the operation sequence computed by the model checker before the actual test of the operation under test can begin. In the context of JML, we can use the JMLUnit tool [21] for this purpose. This tool combines JML with the popular unit testing tool JUnit for Java. JMLUnit views a JML assertion of an operation by pre- and post-conditions as a test oracle. A runtime assertion checker can then be used as the decision procedure for the test oracle.

7 Related Work

Using models for test generation is intensively studied by the model-driven testing community [22, 23], especially for black-box software testing. To the best of our knowledge using visual models for unit-testing is a new idea. However using contracts for testing purposes is a well-known technique (*contract-based testing* [24, 25]). We will mention here a few recent publications in this area and show the differences to our approach.

A recent work [25] explores a fully automatic testing of Eiffel programs with DbC. The authors developed a tool suite AutoTest which randomly generates test cases. A major disadvantage of the approach is that for test case generation they do not take the pre-conditions into account. However, the authors state in [25] that they are currently working on this problem.

The Korat tool [26] uses JML specifications for test case generation for Java programs and handling the oracle problem. The contribution of our approach compared with this approach is that we lift the model abstraction one level higher, where our approach supports behavioral specification in the design phase by using UML notation.

In [27] graph transformation rules are used for test case generation in the context of web services. Test cases are generated from the behavioral specifications and executed via a pre-defined testing interface to ensure the correct funtioning of the web service which is considered as a black-box system. A clear statement about setting a sufficient system state in a black-box system is missing in this work.

8 Conclusion

We have developed an approach that lifts the Design by Contract (DbC) idea, which is usually used at the code level, to the model level. Visual contracts are used as a specification technique. They are used to specify system state transformations with pre- and post-conditions. Pre- and post-conditions are modeled

by UML (composite) structure diagrams. By using UML, we build on a well-known standard that is predominantly used in today's model-driven development processes. Further, we presented how to use the visual contracts in a software development process. A translation of the visual contracts into the Java Modeling Language, a DbC extension for Java, enables the model-driven monitoring. To support our model-driven monitoring method, we provide a visual contract workbench that allows developers to coherently model class diagrams and visual contracts. Further the workbench supports automated code generation.

In this paper, we have shown how we want to extend our approach with model-driven unit testing. In our testing approach, a test case consists of parameter values and a concrete system state. The visual contracts – respectively the generated JML assertions – are viewed in our approach as test oracles to decide whether a manual implementation is correct according to its specification.

In future work we will have to concretize our model-driven unit testing approach and extend our visual contract workbench with testing facilities.

References

1. Meservy, T.O., Fenstermacher, K.D.: Transforming software development: An MDA road map. IEEE Computer **38** (2005) 52–58
2. Lohmann, M., Sauer, S., Engels, G.: Executable visual contracts. In Erwig, M., Schürr, A., eds.: 2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05). (2005) 63–70
3. Engels, G., Lohmann, M., Sauer, S., Heckel, R.: Model-driven monitoring: An application of graph transformation for design by contract. In: International Conference on Graph Transformation (ICGT) 2006. (2006) accepted for publication.
4. Heckel, R., Lohmann, M.: Model-driven development of reactive informations systems: From graph transformation rules to JML contracts. International Journal on Software Tools for Technology Transfer (STTT) (2006) accepted for publication.
5. Meyer, B.: Applying "Design by Contract". IEEE Computer **25** (1992) 40–51
6. Leavens, G., Cheon, Y.: Design by Contract with JML (2003)
7. Beck, K.: Test Driven Development: By Example. Addison-Wesley Professional (2002)
8. Beck, K.: Extreme Programming Explained. Embrace Change. The XP Series. Addison-Wesley Professional (1999)
9. Heckel, R., Lohmann, M.: Towards model-driven testing. Electr. Notes Theor. Comput. Sci. **82** (2003)
10. Antoy, S., Hamlet, D.: Automatically checking an implementation against its formal specification. IEEE Transactions on Software Engineering **26** (2000) 55–69
11. Peters, D.K., Parnas, D.L.: Using test oracles generated from program documentation. IEEE Transactions on Software Engineering **24** (1998) 161–173
12. Jacobson, I., Booch, G., Rumbaugh, J.: The Unified Software Development Process. Addison-Wesley Professional (1999)
13. OMG (Object Management Group): UML 2.0 superstructure specification - revised final adopted specification (2004)
14. Heckel, R., Ehrig, H., Wolter, U., Corradini, A.: Double-pullback transitions and coalgebraic loose semantics for graph transformation systems. APCS (Applied Categorical Structures) **9** (2001) 83–110

15. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundamenta Informaticae **26** (1996) 287–313
16. Zündorf, A.: Graph pattern matching in PROGRES. In Cuny, J., Ehrig, H., Engels, G., Rozenberg, G., eds.: 5th. Int. Workshop on Graph-Grammars and their Application to Computer Science. LNCS 1073 (1996)
17. Binder, R.V.: Testing Object-Oriented Systems. Addison-Wesley (2000)
18. Rensink, A., Schmidt, Á., Varró, D.: Model checking graph transformations: A comparison of two approaches. In: International Conference on Graph Transformation (ICGT) 2004. (2004) 226–241
19. Lohmann, M., Engels, G., Sauer, S.: Model-driven monitoring: Generating assertions from visual contracts. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE) 2006 Demonstration Session. (2006) accepted for publication.
20. Rensink, A.: The GROOVE simulator: A tool for state space generation. In: Applications of Graph Transformations with Industrial Relevance (AGTIVE) 2003. (2003) 479–485
21. Cheon, Y., Leavens, G.T.: A simple and practical approach to unit testing: The JML and JUnit way. In: European Conference on Object-Oriented Programming (ECOOP) 2002. (2002) 231–255
22. Hartman, A., Nagin, K.: The AGEDIS tools for model based testing. In: UML Satellite Activities. (2004) 277–280
23. El-Far, I., Whittaker, J.: Model-based software testing. In Marciniak, J., ed.: Encyclopedia of Software Engineering. Wiley (2001)
24. Heckel, R., Lohmann, M.: Towards contract-based testing of web services. Electr. Notes Theor. Comput. Sci. **116** (2005) 145–156
25. Ciupa, I., Leitner, A.: Automatic testing based on Design by Contract. In: Proceedings of Net.ObjectDays 2005 (6th Annual International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World). (2005) 545–557
26. Boyapati, C., Khurshid, S., Marinov, D.: Korat: automated testing based on java predicates. In: International Symposium on Software Testing and Analysis (ISSTA) 2002. (2002) 123–133
27. Heckel, R., Mariani, L.: Automatic conformance testing of web services. In: Fundamental Approaches to Software Engineering (FASE) 2005. (2005) 34–48

Automated Object's Statechart Generation and Testing from Class Method Contracts

Atul Gupta

Dept. of Computer Science and Engineering,
Indian Institute of Technology Kanpur, India 208 016.
atulkg@cse.iitk.ac.in

Abstract. The link between an object's class specifications and UML statechart is rather informal and poses consistency issues during software evolution. We address this issue by proposing a connection between class diagram and statechart in a lock-step fashion, which leads to a coherent design for better development, testing, and maintenance of the software system. In this paper, we describe an approach for generating a statechart for an object from its method contracts. We then show how the generated object statechart can be used for performing automated testing at unit level. We illustrate the approach using a simple example and discuss its effectiveness as a V&V step during software evolution.

1 Introduction

Models are necessary for performing effective V&V of software artifacts. A model is usually an abstract, partial representation of the software artifact's desired behavior. Any V&V done on this model is on the same level of abstraction as the model. The results of the V&V may not necessarily equally applicable for the software artifact and greatly affected by the difference of the two levels. Ideally, one should use the program code itself to identify and verify its behavior. But there are serious limitations in identifying and applying automated V&V to what represents a 'behavior' of the object in the code thereby requiring an abstract model for suppressing, or ignoring, inessential details while focusing on the important, or essential, details.

The *object model* is the common computational model shared by UML and object-oriented programming languages. The object model views an executing program as a collection of communicating objects where individual objects are responsible for maintaining part of a system's data and for implementing some aspect of its overall functionality. This abstraction provides us a way to deal with the objects as "black boxes" whose details of the underlying implementations are hidden to the users (consumers) of those objects, and all interactions take place through their well-defined interfaces.

An object, in its own right, represents an independent, concurrently executing, and co-existing software artifact. This makes reliability even more important in object-oriented systems than elsewhere. Meyer's "Design by Contract"

[1, 2] approach acknowledged the problem and advocated the use of suitable constraints in the design. Helm et al. [3] introduced 'Contracts' for specifying behavioral compositions and obligations on participating objects. This followed by further investigations done by Jezequel et al. [4] and development of formal approaches for 'contract' based designs [5]. Use of OCL [6] constraints on various UML diagrams is now well accepted and commonplace.

State models like finite state machines, statecharts, etc, on the other hand, have been used as representations of dynamic behavior of the objects/components and are very effective in performing verification and validation activities for them. The state models have been used for requirements validation [7] and systematically generating tests[8–10]. Briand et al., in many studies [11–13], demonstrated the effectiveness of test cases obtained using state based coverage criteria, proposed originally by Offutt et al. [14] and Binder [10].

Efforts have been made to connect other models with state diagrams. Whittle et al. [15, 16] demonstrated how a set of scenarios, along with the notion of domain theory, can be converted into objects' statecharts which helps in identifying the behavior of various components involved in the scenarios. Similar approaches have also been formulated by Makinen et al. [17, 18] and Uchitel et al. [19].

From these studies it has become clear that state models, like UML statecharts are very effective in verification and validation activities of software objects. An important issue then arises is how to obtain these state models so as verification and validation can effectively be performed, and perhaps, be automated.

The main objective of this work is to perform effective V&V by inspecting the statechart generated from an object's method contracts for any incorrect behavior and then generation of automated unit test cases. From the method contracts expressed as OCL constraints [6] and the knowledge of problem domain, we identify a set of key *domain variables* important to the current context. Given this configuration, a constrained method of a class can be viewed as representing a transition in a state space constituted by the set of *domain variables*. Starting with an initial state configuration (typically emulated by a constructor of that object), we searched for possible method invocations at each generated state iteratively and obtained a statechart. The generated statechart is inspected for incorrect behavior and then used for automated unit test generation (including test-oracles generation) to determine if the implementation produced the correct output during testing.

The paper is organized as follows: we first present elements of our model in Section 2. Then we describe our approach and algorithm for generating statecharts from annotated class diagrams in Section 3. We illustrate the statechart generation approach using an example for in Section 4 and performing V&V using the generated statechart in Section 5. We discuss the utility and limitations of our approach in Section 6 and draw conclusions in Section 7.

2 Underlying models

We first present an object's contractual model representing the object's contracts followed by our abstract state model for the object.

2.1 The Constrained Class Diagram

A contract for an object is the set of constraints typically associated with its method invocations. An item in the contract can be:

- A pre-condition to a method which is a restriction that must be true at the moment that the method is going to be executed.
- A post-condition to an method which is a restriction that must be true at the moment that the method has just ended its execution.
- An invariant which is a restriction that must be true before as well as after a method invocation.

Without the loss of generality, we assume an invariant as an integral parts of respected method's pre- and post-conditions. An object's contractual model is then its UML class diagram augmented with method contracts expressed as OCL constraints[6].

2.2 The Abstract State Model

Typically, an object's state is characterized by a specific value assignment of the object's variables in their respected domains. But even for a trivial object with just one integer or floating-point variable, the problem of state explosion makes it intractable to analyze such behavior. This problem can be effectively tackled by the notion of '*state variables*' with '*abstract states*' which facilitates such variables to take abstracted out values over their domains. This requires extra effort to identify such *state variables* and their possible assignments, but provides an opportunity to carry out an effective behavioral analysis which otherwise was not possible.

Formally, we abstract out various data types to be used as state variables, in the following manner:

- Quantifying data types like integer, float, double, etc. are to be mapped into a finite set of disjoint partitions over its valid state space, e.g., an integer state variable X is mapped to three abstract states $x < 0$, $x = 0$, and $x > 0$. A quantified state variable X, in general, can take five possible assignments: bounded range (e.g. $a <= X <= b$), unbounded-ve (e.g. $X < a$), unbounded+ve (e.g. $X > a$), and a specific value (e.g. $X = a$).
- Boolean and enumeration data types are considered inherently abstract.
- Object references x are mapped either to the abstract state x = null, or to the abstract state x instance Of C for each class C of the object referenced by x.

By abstracting out the state dependent variables, a state in the problem domain is the assignment of those state variables with abstract values. For example, a bounded list with maximum size as 'maxElement' is identified by one state variable 'size', which can take theoretically five abstract values: < 0, $= 0$, > 0 AND $< maxElement$, $= maxElement$, and $> maxElement$. Two of the five values, i.e., < 0 and $> maxElement$, are representing invalid states and hence should result in exception throws. Hence the bounded list will be represented by three valid abstract states: $= 0$, $> 0 AND < maxElement$, $= maxElement$ and two invalid states: < 0 and $> maxElement$.

Thus, the state space constituted by n-state variables $s_i(\ i \in [n])$, each with $r_i(\ i \in [n])$ abstracted-out values, can be thought of an n-dimensional hyperspace partitioned into $\Pi r_i\ (i \in [n])$ hyper-cubes, each of which represent an abstract state of the object. For the present state of an object, the constraints on the state variables at a method invocation will decide the resulting valid and invalid states of the object. We developed a rule-based search approach for possible method invocations at a given state and obtaining resulting states which is described in the following Section.

3 Generating the Statechart

Our approach uses an annotated UML class diagram in which class methods[1] have pre and post-condition constraints expressed in OCL. The class diagram annotated with method constraints may be obtained as a result of detailed design activity during software development or the required method level constraints can be attached with desired domain information. Then we choose a state vector S_d, whose elements are important *domain variables*, typically included in pre and post-conditions of the constrained object's methods. Assuming an initial S_d variables' value assignments $S_d(0)$ as the initial state configuration for the generated statechart, we search for object methods whose pre-conditions are consistent with current state configuration. These methods represent state transitions from the current state which is then combined with methods' post-conditions to obtain resulting states. The generated statechart is then inspected to find various possible anomalies in the object's behavior which are used to refine the method contracts for the object. First, we introduce some terms used in this paper.

3.1 Some Definitions

First consider some terms used in our study:

- *Domain Method Set* (M_d): is the set-union of all constrained methods of the object of the specified domain D.

[1] Typically, there are some methods in a class which are of the type getter or observer methods. We exclude them in our approach of statechart generation for better efficiency, as they do not play any role in the behavior of the specified object.

- *A Domain Variable* d_i: is a variable appearing in a pre- and/or post-conditions and/or invariant of a method in M_d.
- *Domain Variable Set* (S_D): is the set-union of d_i.
- *State Vector* (S_d): is a selected subset of the domain variable set $S_d \subseteq S_D$, obtained by identifying some key domain elements of the domain variable set S_D of the domain D of interest.
- *A State Configuration* ($S_d(k)$): is a specific assignments to the variables of state vector S_d for some abstract state k, characterizing that state.

3.2 Identification of Key Domain Variables

Our approach is based on identifying key domain variables set (S_d), each element of which typically represents an important behavior aspect in some context. For example, the variable 'size' in a bounded list plays important decision roles in various operations of that bounded list. In the most straightforward case, for the sake of arguments, S_D may be chosen as (S_d) but a wise selection of *domain variables* from S_D may increase the efficiency (i.e. reduce the efforts requirements) of the whole process.

3.3 State Generation

Having identified (S_d), we obtain an initial state configuration $S_d(0)$ typically specified in a 'constructor' method of the object. Using $S_d(0)$ as the initial state for the given object and the domain method set (M_d), we search all the methods' pre-conditions to find the set of method $M_d(0)$, which can be invoked at initial state 0, i.e. these are the methods whose pre-conditions hold in $S_d(0)$. The set $M_d(0)$ represents all possible transitions (i.e. method invocations), from initial state configuration represented by $S_d(0)$ to some other state i represented by the state configuration $S_d(i)$, obtained according to post-conditions of the methods in set $M_d(0)$. This procedure is repeated for each newly obtained state configurations $S_d(i)$ until all such states are accounted for. States with no method can be invoked at state i. Then the state i will be a final state of the system. An end state of the generated statechart is a state which does not has a transition(i.e. method invocation) to any other state of the object.

To determine, whether a method can be invoked in a given state, we match the pre-condition of that method with the current state configuration. A state configuration is inherently represented in DNF for its constituent state variables and their values assignments. After a match, we determine resulting state by combining the current state configuration with the method post-conditions. To facilitate this, we made following assumptions:

- The pre- and post-conditions are represented in DNF form, so that we can separately analyze each conjunct of the DNF and result can then easily be combined.
- The domain variables which are not state variables, are specified with their range constraints.

Criteria for Method Invocation at a State A specific state can be simply represented as a conjunction of its state variables, assigned with specific/abstracted-out values. Whether a particular method can be invoked in a given state is determined by state-pre-condition match for that method. For state-pre-condition match for a method, we match each literal of a conjunct in the method's pre-condition with corresponding state variable for an overall decision. As we shall see that some conjuncts may result in conditional matches, which form the condition part of that method call and represented as conditional transitions in the resulting state diagram.

Matching of most of state variables of the type boolean, enumerated, and object is rather straight forward, except, the case of quantified variables like integer, float and similar. Here we resolve the problem by exclusively creating a rule base for the matching. A literal, in a method's pre-condition, can take following general form:

$< expr1 > op < expr2 >$

where $< expr1 >$ can be a single quantified variable X or an arithmetic expression containing quantified variables which can be evaluated as X. $< expr2 >$ can be a constant or an arithmetic expression containing quantified variables which may or may not be state variables and op is an algebraic operator ($<, <=, =, >, >=, <>$). Assuming $< expr1 >$ as X, and $< expr2 >$ as Y, i.e., some quantified variables, the two cases for the above literal are handled as follows:

1. Y is a constant. Figure 1 shows the rule base[2] for determining whether a state-pre-condition match holds (or does not hold) in this case which is represented by T (or F) at the leaf nodes. Please note that a successful match may specify a condition to be true and therefore will appear as a conditional transition in the statechart.
2. Y is an arithmetic expression. We evaluate the expression for the constraints given and thus may obtained four possibilities for Y as stated above for bounded range (e.g. $c <= Y <= d$), unbounded-ve (e.g. $Y < c$), unbounded+ve (e.g. $Y > c$), and a specific value (e.g. $Y = c$). Corresponding rule base for this case is given in figure 2.

Criteria for obtaining Resulting State Once an affirmative decision regarding a class method invocation in a given a state configuration is made, its post conditions are evaluated and combined with the state to obtain next state configuration. For this, we determine changes in state variables' values due to the method invocation. Here we describe the mapping of a quantified state variable to its possible values assignments as rest of the variable types can trivially be mapped to their respected categorizations.

There are primarily three types of OCL post-condition expressions which typically manipulate quantified variables. These expressions are handled as given

[2] Only a part of the rule base is shown here for illustration

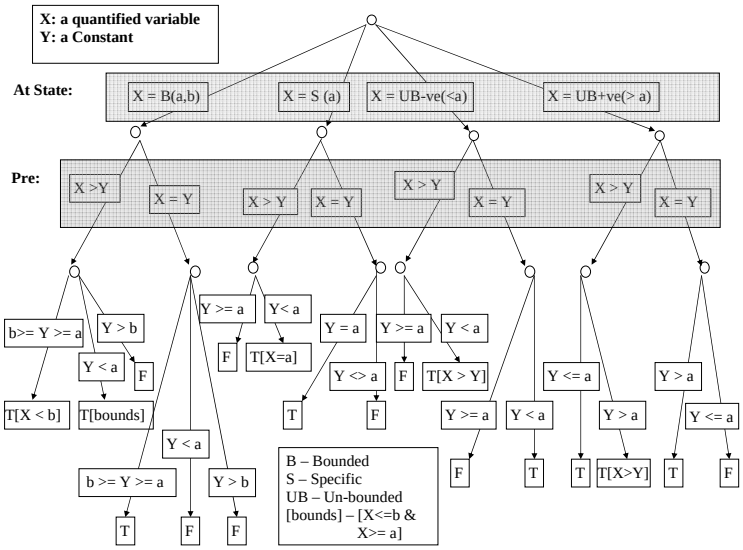


Fig. 1. Handling quantified state variables for state-pre-conditions match (Y is a constant)

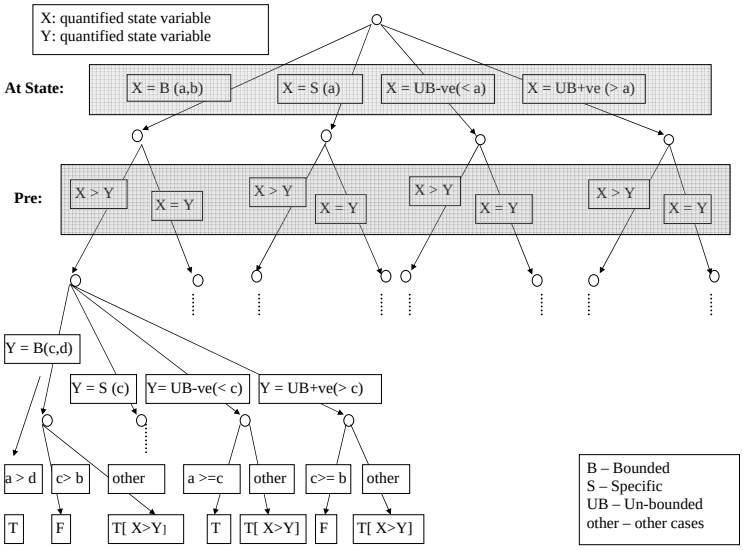


Fig. 2. Handling quantified state variables for state-pre-conditions match (Y is an arithmetic expression)

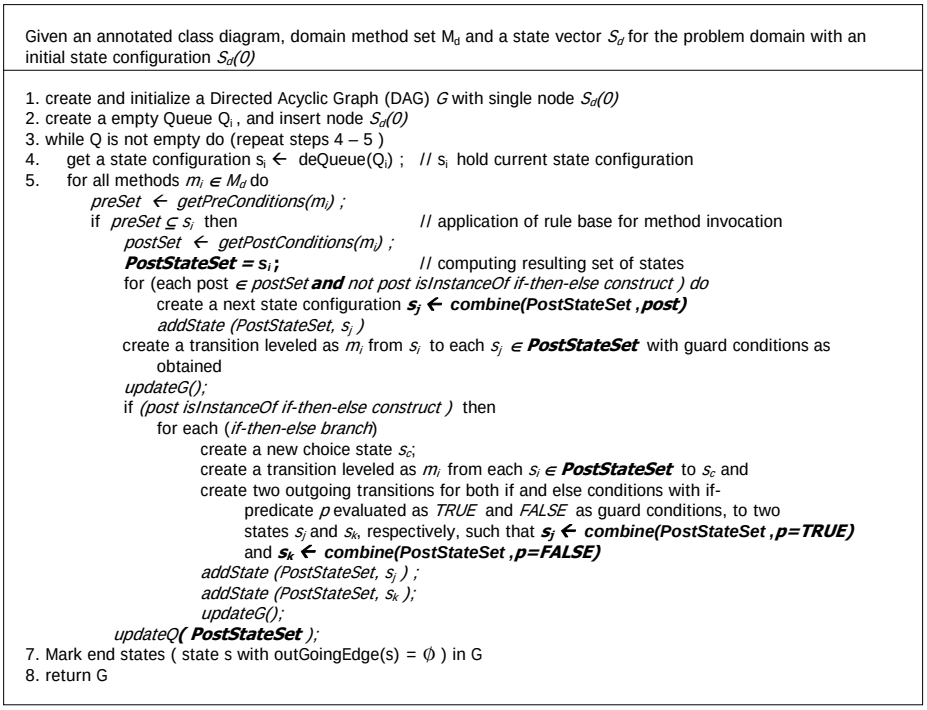


Fig. 3. Algorithm for generating object state model from annotated class diagram

below where X is a single quantified state variable (or an arithmetic expression containing quantified state variables in the 3rd case below):

1. $X \leftarrow X \bullet pre$ **op** $< expr >$ indicating whether the method is additive or subtractive for X depending upon arithmetic operator **op**. Here we follow a conservative approach where next state value for X is mapped to all possible higher abstract state value assignments of X including the current one, in the former case and, to all possible lower abstract state value assignments of X including the current one, for the later case.
2. $if - then - else$ statement will result into a separate state for each if/else block. Such method invocations are modeled by a choice state in the resulting statechart with method name appearing on incoming edge to choice state and the two outgoing transitions from the choice state having guards as $if - condition - predicate$ evaluated as $TRUE$ and $FALSE$, respectively. The conditions and resulting block expressions are similarly evaluated to obtain resulting states.
3. X **op** $< expr >$ where $< expr >$ can be evaluated determine whether it represent a specific value which can be mapped directly to resulting abstract

state value assignment for X, or a range. In either case, it represent next-state constraints on the possible assignments a state variable may take.

The algorithm for automated statechart generation from an annotated class diagram is presented in Figure 3. Please note that a generated statechart corresponds to a specific initial state configuration $S_d(0)$, which can be inferred from a 'constructor' of the object. If the object may be instantiated in multiple ways (i.e. the case with overloaded constructor), then for each possible instantiation, i.e. a different $S_d(0)$, a separate statechart is generated. In that case, an overall statechart for the object is obtained by:

- combining the generated constructor-specific statecharts using a pseudo-start state having a separate transition for each constructor-method leading to the constructor-specific initial state configuration $S_d(0)$.
- combining the common states among the generated constructor-specific statecharts.

3.4 Guard-Conditions for Statechart Transitions

There are two instances where a statechart transition, i.e. a method invocation at a particular state, is to be augmented with a desired guard-condition.

- Firstly, when we explore the possibility of the method to be invoked at that state (ref to Section 3.3), the rule base may return a condition for the transition.
- The other instance is constituted by a method invocation resulting in generation of more than one state 3.3, each of which requires an appropriate guard-condition for the relevant transition in the statechart. These guard-conditions can easily be obtained by the partitioning information of the state variables.

A method involved in both the two instances will have an over-all guard-condition which is a logical *AND* of the two separately obtained guard-conditions.

4 An Illustrated Example: Class *CoinBox*

In this section, we demonstrate our approach using a class *CoinBox* in a Vending Machine domain. The *CoinBox* class is a simple vending machine which takes minimum two quarters to deliver a drink. Some quantity of drinks is to be inserted when the vending machine is empty. It also allows user to get her money back without delivering the drink. The class specifications with OCL constraints are given in Figure 4.

We generate statechart for the class *CoinBox* using our algorithm. First, identify a state configuration S_d as represented by three variables with possible assignments as given below

```
Class CoinBox {
int curQtr, quantity, totalQtrs
boolean allowVend

addQtr( )  // adding a quarter in the machine
  pre   : quantity > 0;
  post  : curQtr  ← curQtr@pre +1
          if (curQtr >= 2) then
          allowVend ← TRUE
retQtrs( )  // returning quarters back to the user
  pre   : curQtr > 0;
  post  : curQtr  ← 0
          allowVend ← FALSE
vend( )  // deliver a drink
  pre   : allowVend = TRUE &&
          quantity > 0;
  post  : curQtr  ← 0
          allowVend ← FALSE
          quantity ← quantity@pre – 1
          totalQtrs ← totalQtrs@pre + curQtr
addDrink(m )  // add  m unit of drink in the
        //machine
  pre   : quantity = 0;
  post  : quantity ← quantity@pre + m
```

Fig. 4. Class *CoinBox* example: Specifications

$s_1 \equiv curQtr \quad := \{0, 1, >= 2\}$
$s_2 \equiv allowVend := \{TRUE, FALSE\}$
$s_3 \equiv quantity \quad := \{= 0, > 0\}$

and an initial state vector $S_d(0)$ as
$S_d(0) := \|0, FALSE, 0\|$

Only one method $addDrink(m)$ can be called at the current state, which will change the object state to
$S_d(1) := \|0, FALSE, > 0\|$

Now, in this state, only method $addQtr()$ can be called, which, results in change of object state changed to
$S_d(2) := \|1, FALSE, > 0\|$

At $S_d(2)$, two methods, $addQtr()$ and $retQtrs()$ may be invoked, resulting $S_d(3) := \|2, TRUE, > 0\|$ and $S_d(1)$,
respectively.

Methods $addQtr()$, $retQtrs()$ and $vend()$ may be invoked at the newly generated state $S_d(3)$. Note that the method $vend()$'s post-conditions include $quantity \leftarrow quantity - 1$. As described in Section 3.3(case -1), the method invocation leads to two different states, $S_d(1)$ and $S_d(0)$ with guard conditions on transitions as

[*quantity* > 1] and [*quantity* = 1], respectively. The complete statechart for the class *CoinBox* is shown in Figure 5 with states named alphabetically.

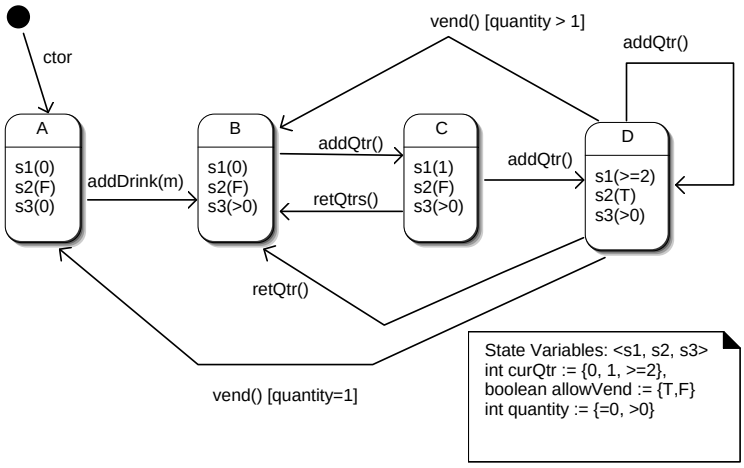


Fig. 5. Generated statechart for the class *CoinBox*

Note that for a different state configuration, one may obtain a different statechart, thereby providing flexibility to the user to experiment and come up with a more desirable statechart for further analysis. For example, the statechart for the class *CoinBox* with slightly different state configuration, is shown in Figure 6.

5 Performing V&V using Generated Statechart

The generated statechart can be carefully inspected to identify various discrepancies in the method contracts. Following discrepancies/errors can be identified as a result of the inspection:

- Incorrect method invocations (transitions)
- Incorrect resulting states.
- Dead states
- Incorrect end states.
- Un-reachable states and transitions.

All these discrepancies can be tracked down to either incorrectly or incompletely specified pre- or post-conditions for object's methods.

The above approach, as any other static verification method, is complementary to software testing and supports it to a greater extent. As we have inspected

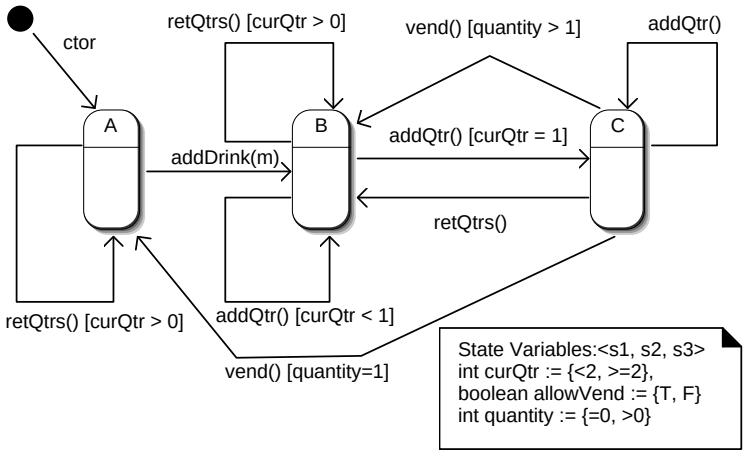


Fig. 6. Generated statechart for the class *CoinBox* (for a different state configuration)

the obtained statechart from a class diagram with pre and post-condition constraints attached with class methods, the actual run-time behavior of the configuration, like proper objects' instantiations, methods actual outputs cannot be verified. The obtained statecharts can be further used for systematically generating automated unit tests to meet these objectives.

Performing Automated Testing

The statechart obtained from the object's class diagram can be considered as an abstract representation of the behavior of that object at source-code level. Therefore, the automated test suite generated from the model can be directly executed against the object's code under test. Due to the state representations (specific assignments of state variables-values) used in our approach, the automated generation of test cases not only has automated generated test sequences, but, unlike other automated-approaches generating tests from statecharts, test-oracles will also be generated automatically. Moreover, the pre/post-conditions and invariant-checks provide additional test-oracles if required.

Various coverage adequacy criteria for statechart based test generation have been defined [14, 10] which include All-Transition coverage(AT), All-Transition-Pair coverage(ATP), Full-Predicate coverage(FP), Transition-Tree coverage (TT). Due to the nature of the constraints involved in the pre-conditions of the method invocations, we may also generate test-inputs automatically, but in general this problem is un-decidable for an arbitrary path in the statechart. In any case, we can easily obtain the test-harness for automated unit testing which includes systematically generated unit tests for these coverage criteria.

As an example of automatically generated JUnit [20] test cases from the generated statechart for the *CoinBox* example coded in Java, we consider the

Figure 5 and use All-Transition (AT) coverage criteria for test generation. An automatically generated test case *testAddDrink*() covering the transition from state A to state B in the figure will look like as follows:

```
public class CoinBoxTest extends TestCase {
CoinBox cbox;


    public CoinBoxTest(String name) {
        super(name);
        cbox = new CoinBox();
    }


    public void testAddDrinkAtA(){                    // testing addDrink(m) in state A
    try{
        assertEquals(0, cbox.getCurrectQtrs());
        assertFalse(cbox.isAllowVend());
        assertEquals(0, cbox.getCurrectQty());
        cbox.addDrink(2);
        assertEquals(0, cbox.getCurrectQtrs());
        assertFalse(cbox.isAllowVend());
        assertTrue(cbox.getCurrectQty()> 0)
    }catch (Exception e){fail("Unwanted exception is raised")}
    }
}
```

The test case oracle before and after the call to *addDrink*(m) method are generated automatically as they represents the state configurations before and after the method call.

The generated statechart is a representation of the explicit behavior of the object, i.e. the methods which can be invoked at any given state. For all other methods, which cannot be invoked, i.e. not shown in the graph, the most straightforward implicit behavior is that it should results in some kind of exception to be raised. For example, invoking method *addQtr*() at start state will raise an exception, which will be ensured by following test case:

```
    public void testaddQtrRaiseExceptionAtA(){
    try{
        assertEquals(0, cbox.getCurrectQtrs());
        assertFalse(cbox.isAllowVend());
        assertEquals(0, cbox.getCurrectQty());
        cbox.addQtr();
        fail("Exception should be raised")      }catch (Exception e){}
    }}
```

6 Discussions

A model is usually an abstract, partial representation of the software artifact's desired behavior. Any V&V done on this model are on the same level of abstraction as the model. The results of the V&V may not necessarily equally applicable for the software artifact and greatly affected by the difference of the two levels. The statechart obtained from the object's class diagram is an abstract representation of the behavior of that object at source-code level. Therefore, the automated test suite generated from the model can be directly executed against the object's implementation.

An important requirement of the proposed approach is the identification of the set of proper *domain variables* to be used as state variables. In the most straightforward case, S_D may be chosen as (S_d). A wise selection of *domain variables* from S_D, however, may increase the efficiency (i.e. reduce the efforts requirements) of the whole process. For instance, the variable *totalQtrs* in our *CoinBox* class, incremented by the *curQtr* in the post-conditions of method *vend*(), can be left out from the considerations of state variables. For the given problem, the variable *totalQtrs* does not play any significant role in the behavior of the *Coinbox* object.

An important outcome of this exercise is that it allows a modeler to specify object's dynamic requirements declaratively on an abstract level, at one place on a structural diagram (i.e. class diagram) without referring to its operational dynamic diagrams like statecharts as they can be generated on the fly. The approach is inherently suited to a typical evolutionary software development where software objects gradually evolve incrementally. Other benefits and limitations of the approach are discussed in the following Sections.

6.1 Effective V&V for Software Re-Use

Starting from initial state, by exploring the possibilities of a method invocation in each of the state next generated, we virtually worked out for all possible simple (no repetition) execution-sequences of object's methods with respect to the statechart generated. Our approach, at the same time, rules out all possible simple invalid-sequences. Allowing method-repetitions (i.e. loops in the statechart) in the test sequences may further improve the testing results. This is highly desirable for software re-use. Moreover, the objects with method contracts so obtained are easier to integrate and the assembly is expected to be more reliable.

6.2 Change Management

The approach strengthens the link between an object's constrained class diagram and its UML statechart by generating the statechart from the constrained class diagram. The changes made on the obtained statechart can be mapped to the method constraints and vice-versa. This lock-step of the two specifications will be highly desirable during software evolution and change management. The *CoinBox* example validates the point made here.

6.3 Limitations of the Proposed Approach

Simplicity of our approach lies in the fact that state generation depends only in identifying the set of domain variables of interest, i.e., S_d of that configuration and their abstracted-out partitions for the domain values. This requirement, however, seems to add to the variability in the effectiveness of our approach applied by two developers on the same problem as the two may choose different sets of domain variables of interest and/or their partitions. Clearly, the effectiveness of this approach depend on the size of the state space as constituted by global state vector S_d, and results of this analysis are to be interpreted accordingly. Arguably, this approach provides more flexibility to the designers, for the amount of efforts they are willing to make for verification at that level.

Automated testing is possible but at times, it may have to be supplemented with the test inputs for some paths for which test data may not be automatically generated.

7 Conclusions and Future Work

Automation is the future trend of software V&V in order to reduce its cost. In the past decades, a great amount of research effort has been spent on automatic test case generation, automatic test oracles, etc. However, the current practice of software test automation is still mostly based on recording manual testing activities and replaying recorded test scripts for regression testing.

In this work, we presented a simple and practical approach to automatically generate a UML statechart for an object from its annotated structural representation, namely, UML class diagrams. Our approach make use of OCL pre and post-condition constraints specified at method level and a state vector S_d consists of important domain variables involved in the constraint specifications. From an initial value assignment of S_d as start state, we incrementally generate remaining of the statechart by searching applicable methods at current state and obtaining resulting states until all the generated states are explored.

The generated statechart can be used for performing effective automated testing of the implicit and explicit behavior of the object. The resulting statechart and class specifications are in a lock-steps which facilitates effective incremental development and change management including efficient regression testing of the objects.

Identifying the set of key run-time domain variables is possible as we are dealing at near-code level, but informal selection of these variables for our analysis may require some non-trivial efforts. There is a need of developing formal approaches to simplify this task. Testing may not be fully automated as test inputs for some paths may not be automatically generated. We did not make any effort estimation for the proposed approached; therefore, make no claim about the efficiency of the approach. This should be treated as future work. We also plan to undertake some case studies as to investigate the effectiveness of our approach on real life problems.

References

1. Meyer, B.: Object-Oriented Software Construction. Prentice Hall, Englewood Cliffs N.J. (1988)
2. Meyer, B.: Design by contracts. IEEE Computer **25** (1992) 40–52
3. Helm, R., Holland, I.M., Gangopadhyay, D.: Contracts: specifying behavioral compositions in object-oriented systems. In ACM SIGPLAN (ECOOP/OOPSLA '90 Proceedings) **25** (1990) 169–180
4. Jezequel, J.M.: Design by contract: The lessons of ariane. IEEE Computer **30** (1997) 129–130
5. Andrade, L.F., Fiadeiro, J.L.: Interconnecting objects via contracts. In: In Proc. Second Int'l Conf. the Unified Modeling Language (UML '99). (1999) 566–586
6. : Ocl 2.0 specifications. (http://www.omg.org/docs/ptc/05-06-06.pdf)
7. Chow, T.: Testing software design modeled by finite-state machines. IEEE Trans. on Software Engineeing **4** (1978) 178–187
8. Kim, Y., Hong, H., Bae, D., Cha, S.: Test cases generation from uml state diagrams. IEE Proceedings on Software **146** (1999) 187–192
9. Hyoung, S., Young, G., Sung, D., Doo, H., Hasan, U.: A test sequence selection method for statecharts. Jour. Software Testing, Verification and Reliability **10** (2001) 203–227
10. Binder, R.: Testing Object-Oriented Systems—Models, Patterns, and Tools, Object Technology. Addison-Wesley (1999)
11. Antoniol, G., Briand, L., Penta, M., Y., L.: A case study using the round-trip stretegy for state-based class testing. In: in Proc of the 13th Int'l Symp. On Reliability ISSRE'02. (2002)
12. Briand, L., Labiche, Y., Wang, Y.: Using simulation to empirically investigate test coverage criteria based on statechart. In: In Proc of the 26th Int'l Conf. on Software Engineering ICSE. (2004)
13. Briand, L., Penta, M.D., Labiche, Y.: Assessing and improving state-based class testing - a series of experiments. IEEE Trans. on Software Engineeing **30** (2004) 1–37
14. Offutt, A., Abdurazik, A.: Generating tests from uml specifications. In: In Proc. Second Int'l Conf. the Unified Modeling Language (UML '99). (1999) 416–429
15. Whittle, J., Schumann, J.: Generating statechart designs from scenarios. In: In Proc of the 22nd Int'l Conf on Software Engineering ICSE. (2000)
16. Whittle, J., Saboo, J., Kwan, R.: From scenarios to code: An air traffic control case study. In: In Proc of the 25th Int'l Conf on Software Engineering ICSE. (2003)
17. Makinen, E., Systa, T.: Mas - an interactive synthesizer to support behavioral modeling in uml. In: In Proc of the 23rd Int'l Conf on Software Engineering ICSE. (2001)
18. Makinen, E., Systa, T.: An interactive approach for synthesizing uml statechart diagrams from sequence diagrams. In: OOPSLA, Minneapolis, USA (2000)
19. Uchitel, S., Kramer, J.: A workbench for synthesising behaviour models from scenarios. In: In Proc of the 23rd Int'l Conf on Software Engineering ICSE. (2001)
20. : Junit home page. (http://www.junit.org)

Using B to verify UML Transformations

Kevin Lano

Dept. of Computer Science, King's College London, Strand, London, WC2R 2LS, UK

Abstract. This paper describes the use of the B formal method to verify semantic properties of UML graphical models, and the correctness of transformations on these models.

1 Introduction

UML is a large and complex notation, in which many aspects of the semantics remain incomplete or are only expressed in an operational manner, ill-suited for analysis using proof tools. Specific problems include:

1. Complex use of undefined and null values within OCL, and missing/inexpressible axioms of OCL. For example the $s \rightarrow at(i)$ operation does not have a formal semantics in UML 2.0 [1].
2. Lack of semantic consistency properties for individual models and between models of the same system [2]. For example, it is necessary that the state invariants of a state machine defining the classifier behaviour of a particular class should be consistent with the invariants of that class.
3. Lack of grounded interpretation for UML concepts, independent of UML [3].

Many of these problems are due to the lack of an objective semantics [4] for UML, and the desire by the UML community to leave certain semantic aspects open, to support a wider use of the notation across different domains. At the same time, an objective semantics is essential to support reuse (if we don't know what a diagram means, how can we reuse it and its corresponding developed code?) and to support verification (the diagrams should have a mathematical semantics because they are abstract descriptions of mathematically precise artifacts – programs).

We solve some of these problems by defining a subset, UML-RSDS (Reactive system development support), of UML, which has a precise semantics based on ZF set theory and classical predicate calculus, which is also the foundation for the Z [5] and B [6] specification languages. This semantics is entirely independent of UML. B provides a notation in which the semantics of models can be expressed and used for verification and validation of the models. B provides an integration of class diagram and state machine models in single components (B machines).

Figure 1 shows the overall development process supported by UML-RSDS and its accompanying toolset. A developer can construct PIM or PSM class diagrams and state machines using the tool, transform models to improve their

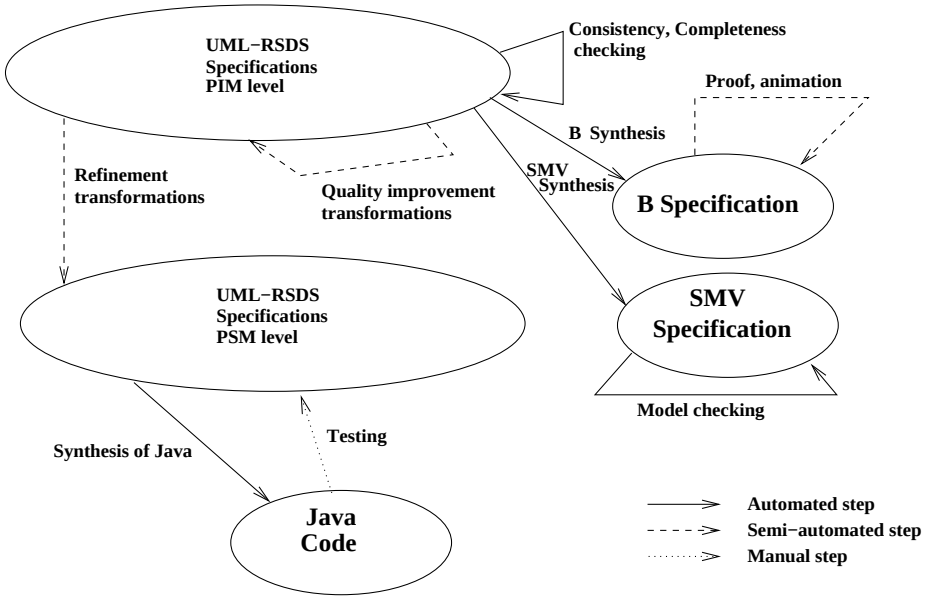


Fig. 1. UML-RSDS Development Steps

quality or refine them, translate to B [6] or SMV [7] for semantic analysis, and generate Java code from a Java PSM.

The translation from UML to B closely corresponds to the semantics of UML-RSDS defined in [8]: both the mathematical elements used to interpret classes, associations, etc, and the structuring/interrelationships of machines parallels the formalisation of models as theories, and the relations of inclusion/extension between these theories. However there are some differences, such as the absence of real numbers in B, and the absence of internal concurrency in B.

Section 2 defines the UML-RSDS notation, Section 3 describes the UML-RSDS tools. Section 4 describes the translation from UML-RSDS specifications to B. Section 5 shows how UML transformations can be verified using this translation. Section 6 gives a comparison with related work.

2 Specification in UML-RSDS

UML-RSDS specifications consist of:

1. A UML class diagram, including constraints attached to operations, classes and (sets of) associations;
2. A use-case diagram, defining the operations of the system;
3. State machine models attached to classes or operations in the class diagram, or to use cases.

Class attributes can be stereotyped as *input*, *internal*, *derived* or *output*: Derived attributes are prefixed by / as usual. The prefix ? indicates an input attribute and ! an output. These stereotypes are applicable for many different kinds of system, for example, an input field on a web page, or a sensor in a process control system, could both be represented as input attributes.

2.1 Specification Example

An example of a UML-RSDS specification, of part of a robot control system from the production cell case study [9], is shown in Figure 2.

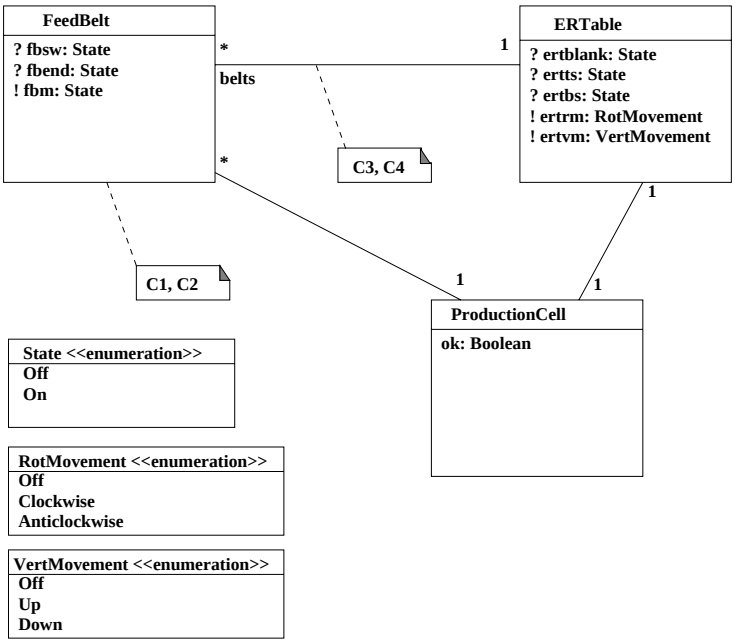


Fig. 2. Class Diagram of Production Cell System

The *FeedBelt* class represents feed belts, which move work pieces (such as car bodies) into the robot production cell. These have a motor *fbm* to move the belt, a switch *fbsw* to switch the belt on and off, and a sensor *fbend* to detect if a piece has reached the end of the belt, ready for unloading into the next component of the cell.

One such component is an 'elevating rotating table', represented by the *ERTable* class. These tables have two motors *ertvm* for vertical movement and *ertrm* for rotary movement, and two sensors *ertts* and *ertbs* to detect if the table is at its top or bottom position, respectively. The sensor *ertblank* detects if there is a work piece in the table.

Normally one or more feed belts may feed blanks into a given table. *belts* gives, for each table, the set of belts that feed that table.

Some example constraints in this system are:

- $C1$ "If the belt switch is off, the motor is off":

$$fbsw = Off \Rightarrow fbm = Off$$

- $C2$ "If there is no blank at the end, the belt keeps moving":

$$fbsw = On \ \& \ fbend = Off \Rightarrow fbm = On$$

These are local invariants of the *FeedBelt* class.

A constraint which links the feed belt and table classes is:

- $C3$ "If a belt is ready to unload, and its table is ready to receive a blank, then unloading may proceed":

$$fbsw = On \ \& \ fbend = On \ \&$$
$$ertblank = Off \ \& \ ertts = On \Rightarrow fbm = On$$

$C3$ is a constraint on the association between *FeedBelt* and *ERTable*: it specifies that, for any pair of feed belt and table objects linked by this association, that the given invariant must hold true. In this system the association represents the physical connection between the robot system components: that the belt is positioned to feed blanks to the table.

2.2 UML-RSDS Constraints

One significant feature of UML-RSDS class diagrams is that constraints may be attached to associations, these represent an implicit universal quantification over all the objects linked by the associations.

Table 1 shows the syntax of constraints currently accepted in UML-RSDS constraints, within the UML-RSDS tools. A *valueseq* is a comma-separated sequence of values. A factor level operator $op1$ can be: $+$, $-$, $*$, $/$, div, mod, $\backslash/$, $/\backslash$ (also written as $\cup$ and $\cap$), or $\frown$. A comparator operator $op2$ is one of $=$, $/=$, $<$, $>$, $<=$, $>=$, $:$, $<:$, $/:$, $/<:$. A logical operator $op3$ is one of $\&$, or. Identifiers are either class names, function names, class features (attribute, operation or role names), elements of enumerated types, or represent variables or constants (if in upper case). Variables are implicitly universally quantified over the entire formula. Operations can also be written with parameters as $op(p_1, ..., p_n)$, etc.

The notation $objs \mid (predicate)$ denotes the select operator, and evaluates to the set of elements of *objs* which satisfy *predicate*.

< *value* >	::= < *ident* > |	Variable expression.
	< *number* > | < *string* > |	Primitive literal
	< *boolean* >	expressions.
< *objectref* >	::= < *ident* > |	
	< *objectref* >.< *ident* > |	Navigation call expression.
	< *objectref* > `|(` < *expression* > `)`	Select expression.
< *arrayref* >	::= < *objectref* > |	
	< *objectref* >`[`< *value* >`]`	At expression.
< *factor* >	::= < *value* > |	
	{ < *valueseq* > } |	Collection literal
	`Sequence{` < *valueseq* > } |	expressions.
	< *arrayref* > |	
	< *factor* > op1 < *factor* >	Infix binary operation call (1)
< *expression*1 >	::= < *factor* > op2 < *factor* >	Infix binary operation call (2)
< *expression* >	::= < *expression*1 > |	
	`(` < *expression* > `)` |	
	< *expression*1 > op3 < *expression* >	Infix binary operation call (3)
< *invariant* >	::= < *expression* > |	
	< *expression* > `=>` < *expression* >	

Table 1. UML-RSDS Constraint Syntax

3 The UML-RSDS Tools

A large toolset has been developed to support UML-RSDS. The tool facilities include:

1. Diagram creation and editing for class diagrams and state machines.
2. Syntactic and semantic checks on diagram correctness, including consistency and completeness of constraints.
3. Transformations on UML models.
4. Automated translations from UML-RSDS specifications into SMV, the B notation, and Java.

The translation and diagram checking operations are fully automated. Transformations are also automatically applied, but must be selected manually by the tool user.

In addition, there are facilities for the creation of web applications.

4 Translation from UML-RSDS to B

To semantically analyse UML models, and to animate (test using symbolic execution) models, we use a translation to the B notation [6]. B is an established formal method which has been extensively used in industry, particularly in the

European railway industry [10]. It has comprehensive tool support, the B Toolkit [11], Atelier B [12] and B4Free.

The translation from UML-RSDS into B essentially represents the axiomatic UML-RSDS semantics [13, 14, 8] of models in the B language. Each class E is represented by a variable es (the set of instances of E currently existing) and a type E_OBJ with $es \subseteq E\_OBJ$. Each instance attribute att of type T is represented by a map

$$att : es \rightarrow T'$$

where T' is the representation of T in B. Associations are also represented as maps, Table 2 shows the most common cases.

Association	*B role type*	*B invariants*
$A_{*}{}^{r}_{*}B$	$r : as \rightarrow \mathbb{F}(bs)$	
$A_{0..1}{}^{r}_{*}B$	$r : as \rightarrow \mathbb{F}(bs)$	$\forall a.(a \in as \Rightarrow r(a) \cap union(r[as - \{a\}]) = \varnothing)$
$A_{1}{}^{r}_{*}B$	$r : as \rightarrow \mathbb{F}(bs)$	$\forall a.(a \in as \Rightarrow r(a) \cap union(r[as - \{a\}]) = \varnothing)$
		$union(r[as]) = bs$
$A_{*}{}^{r}_{1}B$	$r : as \rightarrow bs$	
$A_{0..1}{}^{r}_{1}B$	$r : as \rightarrowtail bs$	

Table 2. Representation of Associations in B

Ordered associations are represented in a similar manner, except that the range type of the B representation is $seq(bs)$ instead of $\mathbb{F}(bs)$. Table 3 shows the interpretation of some basic expressions.

OCL	*Semantic Representation in B*
Variable or constant x, primitive or string value x	x
Attribute of single-valued expression $obj.att$	$att(obj)$
Attribute of set-valued expression $s.att$	$att[s]$
Role of single-valued expression $obj.role$	$role(obj)$
Multiplicity ONE role of set-valued expression $s.role$	$role[s]$
Non-ONE role of set-valued expression $s.role$	$union(role[s])$

Table 3. The Interpretation of OCL expressions in B

The B Toolkit can then be used to check if a UML specification has a model, non-trivial models, or to animate the specification. It can also be used to compare two models to verify that one is a refinement of another, ie, that all functional properties of one model are also true in a proposed refining model.

Proof obligations for internal consistency of a module in B are:

1. That there is some state which satisfies the module constraints and the typing constraints.

2. That all the constraints are true in the initial state.
3. That each operation, if executed within its precondition, maintains the truth of each constraint.

These correspond directly to similar properties of the UML-RSDS class or subsystem from which the B module was derived. Together they ensure that the constraints are always true, for each object of the class, at time points where no operation is executing on the object provided that operations are only executed within their preconditions (the latter becomes a proof obligation for callers of the operations). Condition 3 ensures that each transition into a state of a state machine attached to a class establishes the invariant of that state.

Animation can be used to check that state invariants of a class are consistent with the class invariants: it should be possible to enter each state of the state machine while satisfying the class invariants.

In the translation to B, the effect of an input event is made explicit: the changes to all objects affected by the event are defined in the B operation which represents the event. The semantics of inheritance, state machines and dynamic binding are also made explicit in the B translation.

The translation to B uses a pragmatic approach which attempts to make the resulting B specification as modular as possible, to enable a close correspondence between the B and UML, and to improve the feasibility of proof. Classes A and B are translated to separate B machines A and B unless:

1. A and B are linked by inheritance: all descendents of a class E without ancestors are grouped into a single machine E.
   If instead we require A and B to be represented by separate machines, the transformation 'replace inheritance by association' can be applied before translation to B.
2. A and B are members of a cycle of (directed) associations: an association $E \rightarrow_r F$ is represented as a variable r of E which refers to a variable of F, so that machine E USES machine F. Cycles are not permitted in the USES relationship, so if there are dependencies in both directions the machines for A and B must be extended by a third machine S representing the complete subsystem of A, B and their linking associations together. The variables representing the associations and the operations on these are placed in S.

Figure 3 shows an example of how UML structures are represented in B.

For the production cell there are no inheritances or cyclic dependencies of classes, so all classes can be specified in separate B machines. A *SystemTypes* machine encapsulates the type definitions of the system:

```
MACHINE SystemTypes
SETS State = {Off, On}; FeedBelt_OBJ; ERTable_OBJ
END
```

The *FeedBelt* machine gives the semantic representation of the *FeedBelt* class, with attributes expressed as maps from the set of existing *FeedBelt* objects (*feedbelts*) to their type sets, and operations synthesised to maintain the class invariants:

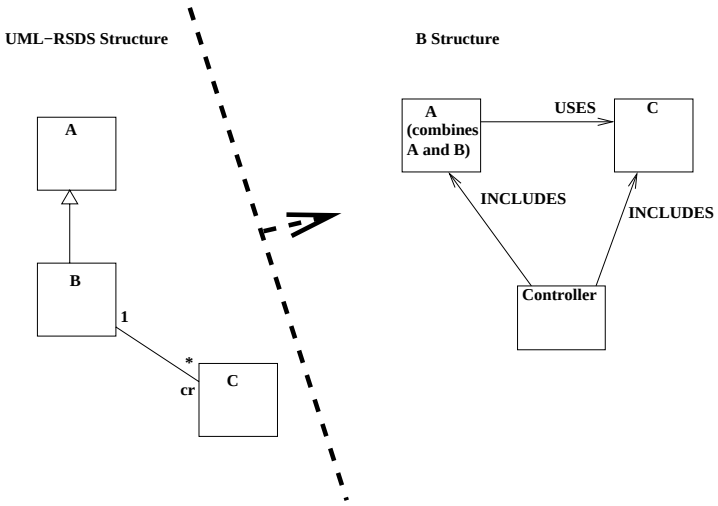


Fig. 3. Structure of B Translation of UML

```
MACHINE FeedBelt
SEES SystemTypes
VARIABLES feedbelts, fbsw, fbend, fbm
INVARIANT (feedbelts <: FeedBelt_OBJ) &
  (fbsw : feedbelts --> State) &
  (fbend : feedbelts --> State) &
  (fbm : feedbelts --> State) &
  /* C1: */
  (!feedbeltx.(feedbeltx : feedbelts =>
      ( fbsw(feedbeltx) = Off => fbm(feedbeltx) = Off ))) &
  /* C2: */
  (!feedbeltx.(feedbeltx : feedbelts =>
      ( fbsw(feedbeltx) = On &
        fbend(feedbeltx) = Off => fbm(feedbeltx) = On )))
INITIALISATION feedbelts := {} ||
  fbsw := {} || fbend := {} || fbm := {}
OPERATIONS
    oo <-- new_FeedBelt(fbswx,fbendx,fbmx) =
    PRE feedbelts /= FeedBelt_OBJ & fbswx : State &
      fbendx : State & fbmx : State &
      ( ( fbswx = Off => fbmx = Off ) &
        ( fbswx = On & fbendx = Off => fbmx = On ) )
    THEN
      ANY feedbeltx
      WHERE feedbeltx : FeedBelt_OBJ - feedbelts
      THEN feedbelts := feedbelts \/ {feedbeltx} ||
        fbsw(feedbeltx) := fbswx ||
```

```
        fbend(feedbeltx) := fbendx ||
        fbm(feedbeltx) := fbmx ||
        oo := feedbeltx
      END
    END;

    setfbsw(feedbeltx,fbswxx) =
    PRE feedbeltx : feedbelts & fbswxx : State
    THEN fbsw(feedbeltx) := fbswxx ||
      IF fbswxx = Off
      THEN fbm(feedbeltx) := Off   /* Derived from C1 */
      ELSE IF fbswxx = On & fbend(feedbeltx) = Off
      THEN fbm(feedbeltx) := On   /* Derived from C2 */
      END
      END
    END;

  ...
END
```

The controller machine defines all externally-available operations of the system, and manages the global invariants.

For each operation such as *setfbsw* there is both a local version *setfbsw* of the operation, defined in the machine representing the owning class of the operation, and a global version *set_fbsw*, defined in the *Controller* machine. The local version carries out those updates of local features due to the operation, whilst the global version carries out updates of non-local features.

The specifications of B operations are generated from UML-RSDS constraints. These specifications are derived from three parts of the UML-RSDS models:

1. The invariant constraints of the system.
2. The pre and post constraints of the operation, together with its declaration in its owning class.
3. The statemachine of the owning class.

Operations which update features may affect the truth of invariant constraints, both local and global. Therefore it may be necessary to define additional effects for the operation, to maintain these constraints.

In general there are five stages in deriving operation code from invariant constraints:

– For each individual constraint:
  1. Identify if the operation can invalidate the constraint, and therefore if new code needs to be added to the operation to ensure that the constraint is not invalidated.
  2. Identify what set of objects throughout the system can be affected by the operation.

3. Identify what updates are required on each affected object to maintain constraints.
4. Convert the update and the conditions under which it applies into B notation.

– Integrate the B derived from each individual constraint into an overall effect for the operation.

For example, in the case of *setfbsw*(*fbswxx*), this operation can violate both $C1$ and $C2$. The new updates which need to be added are:

```
fbswxx = Off  =>  AX(fbm = Off)
```

in the first case, and

```
fbswxx = On & fbend = Off  =>  AX(fbm = On)
```

in the second. No further objects in the system are affected by these actions, so the required updates are purely local to the feedbelt *feedbeltx*. The updates become

```
IF fbswxx = OFF THEN fbm(feedbeltx) := Off END
```

and

```
IF fbswxx = On & fbend(feedbeltx) = Off
THEN fbm(feedbeltx) := On
END
```

in B notation. These can be integrated into a single operation using an IF THEN ELSE structure as their conditions are mutually exclusive.

Operation postconditions can modify local features of a class. These updates are specified in the same manner as in constraint succedents, with the addition that the value of a modified attribute *att* at the start of the operation can be referred to as *att*@*pre*.

Underspecified postcondition constraints can be formalised using the ANY construct of B.

Behavioural statemachines can be attached to a class C, to define how the operations of that class change the state of the class. The transitions of the statemachine can modify local features of the class and also invoke operations of supplier objects. In the translation to B, the local updates are carried out in the local version of the operation, and the non-local are carried out in the *Controller* version.

The correctness of the translation can itself be verified by providing a common semantics for B and UML, and demonstrating that the B translation $T(e)$ of any UML element e has the same semantics as e [15].

The close correspondence between the UML and the B translation permits analysis on the generated B to be interpreted directly in terms of the model it is derived from.

5 Verification of Model Transformations

Transformations on UML models include:

1. Quality improvements, such as removing redundant classes or associations
2. PIM to PSM transformations, such as the replacement of many-many associations by many-one associations (for implementation of a data model in a relational database).
3. Introduction of detailed design elements, such as a design pattern.

A large number of UML model transformations are known in the modelling community, and some, such as transformations of class diagrams to relational database ER diagrams, have been embedded in commercial tools. We also provide a wide range of transformations in the UML-RSDS tools.

However, developers may need to apply variations of known model transformations, or devise new transformations, and the correctness of these should be shown, so that properties of the original system are preserved in the transformed system.

The translation from UML to B described in the previous section can be used for such verification, by using the B concept of formal refinement. Figure 4 shows the approach adopted. The models on the LHS can be combinations of

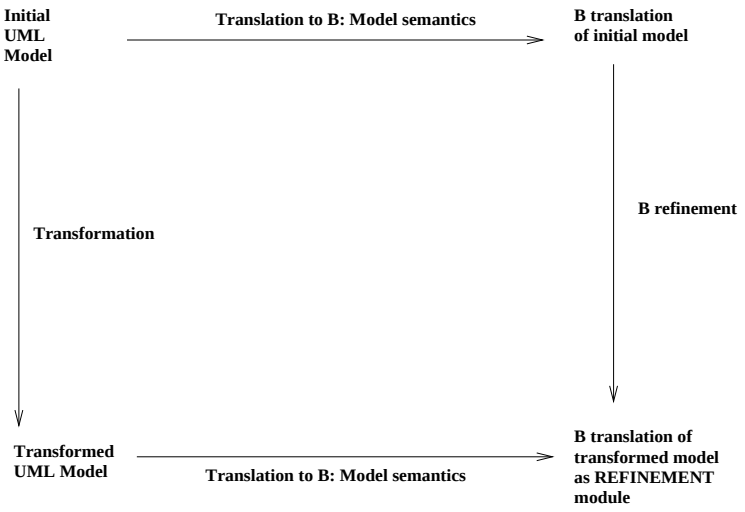


Fig. 4. Transformation Verification

class diagrams and state machines, as in transformations which introduce the State pattern. Transformations between different modelling languages could also be proved correct, provided both languages have a semantics expressible using B.

A model transformation to be verified is expressed in a general form, and both the original model and the transformed model are translated to B, which expresses their semantics (the translation to B is performed automatically by the UML-RSDS tool). The transformed model is defined in a B module which is declared as a REFINEMENT of the B module which expresses the semantics of the original model. The B proof obligations for refinement can then be generated using a tool for B. These obligations are:

1. That the static invariants of the original module remain true (under the data transformation) in the refined module.
2. That the possible initialisations of the refined module correspond to possible initialisations of the original module.
3. That for each operation *op* of the refined module, its behaviour as defined in the refined module is consistent with its behaviour as defined in the original module. More precisely, each possible execution of the refined version of *op* corresponds under the data transformation to a possible execution of the original version.

The refinement proof in B establishes that all pre-post properties of operations and that all invariant properties of the original UML model are also valid in the transformed model.

Each such proof verifies a general family of model transformations. For example, consider the transformation 'replace many-many association by two many-one associations' shown in Figure 5.

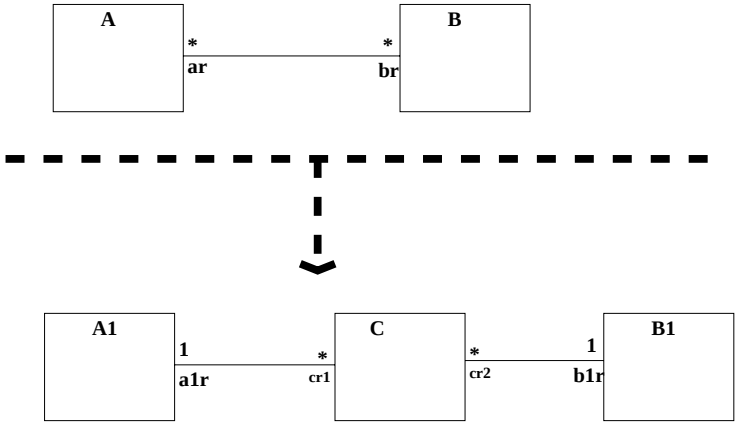


Fig. 5. 'Replace many-many association' Transformation

The B module representing the semantics of the original model is:

```
MACHINE Model1
```

```
SETS A_OBJ; B_OBJ
VARIABLES as, bs, ar, br
INVARIANT (as <: A_OBJ) & (bs <: B_OBJ) &
  (ar: bs --> FIN(as)) & (br: as --> FIN(bs)) &
  !ax.(ax : as =>
       !bx.(bx : bs & bx : br(ax) => ax : ar(bx))) &
  !ax.(ax : as =>
       !bx.(bx : bs & ax : ar(bx) => bx : br(ax)))
INITIALISATION
  as := {} || bs := {} || br := {} || ar := {}
OPERATIONS
  addbr(ax,bx) =
    PRE ax: as & bx: bs
    THEN
      br(ax) := br(ax) \/ {bx} ||
      ar(bx) := ar(bx) \/ {ax}
    END:
  ...
END
```

There are also operations to create A and B instances, and to remove elements from the association, etc.

The model of the new system has the formalisation:

```
REFINEMENT Model2
REFINES Model1
SETS C_OBJ
VARIABLES a1s, b1s, a1r, b1r
INVARIANT (a1s <: A_OBJ) & (b1s <: B_OBJ) & (cx <: C_OBJ) &
  (a1r: cs --> a1s) & (b1r: cs --> b1s) &
  (cr1: as --> FIN(cs)) & (cr2: bs --> FIN(cs)) &
  !ax.(ax : a1s =>
       !cx.(cx : cs & cx : cr1(ax) => ax = a1r(cx))) &
  !ax.(ax : a1s =>
       !cx.(cx : cs & ax = a1r(cx) => cx : cr1(ax))) &
  !bx.(bx : b1s =>
       !cx.(cx : cs & cx : cr2(bx) => bx = b1r(cx))) &
  !bx.(bx : b1s =>
       !cx.(cx : cs & bx = b1r(cx) => cx : cr2(bx))) &

  a1s = as & b1s = bs &
  !ax.(ax : a1s => br(ax) = b1r[cr1(ax)]) &
  !bx.(bx : b1s => ar(bx) = a1r[cr2(bx)])
INITIALISATION
  a1s := {} || b1s := {} || cs := {} ||
  b1r := {} || a1r := {} || cr1 := {} || cr2 := {}
OPERATIONS
  addbr(ax,bx) =
    PRE ax: a1s & bx: b1s
    THEN
```

```
      IF bx : b1r[cr1(ax)]
      THEN skip
      ELSE
        ANY cx WHERE cx : C_OBJ - cs
        THEN
          cs := cs \/ {cx} ||
          a1r(cx) := ax ||
          b1r(cx) := bx ||
          cr1(ax) := cr1(ax) \/ {cx} ||
          cr2(bx) := cr2(bx) \/ {cx}
        END
      END
    END;
  ...
END
```

The last four invariant conjuncts describe the refinement relation corresponding to the data transformation, and they define how the data of the original model is interpreted in terms of the new model.

The invariants of the original model must be proved correct for these interpretations, for example the property that ar and br are inverse roles:

```
!ax.(ax : as =>
     !bx.(bx : bs & bx : br(ax) => ax : ar(bx)))
```

must hold in the form:

```
!ax.(ax : a1s =>
     !bx.(bx : b1s & bx : b1r[cr1(ax)] => ax : a1r[cr2(bx)]))
```

This is proved by using the corresponding properties of the pairs of inverse roles $a1r$ and $cr1$ and $b1r$ and $cr2$.

For each operation, each execution of the operation according to the *Model*2 definition must satisfy the *Model*1 specification of the operation, under the interpretation of *Model*1 data in *Model*2. Informally this is clear for *addbr*, since if there is not already a cx with

$$ax = a1r(cx) \;\&\; bx = b1r(cx)$$

then such a cx is created and results in bx being added to $b1r[cr1(ax)]$, and ax to $a1r[cr2(bx)]$ as required. Formal proof of the transformation requires precise assumptions (which might be neglected in informal definitions of UML transformations). In this case we require that no memory problems occur, and that it is always possible to allocate a new cx object as required in the new definition of *addbr*. We ensure this by fixing C_OBJ as isomorphic to $A\_OBJ * B\_OBJ$, and only permitting at most one cx object to be linked to a particular pair (ax, bx) of A and B elements.

6 Related Work

Related work on UML is the U2B tool of Butler [16] as part of the RODIN project [17], and translations [18] from UML to Object-Z.

Constraints which need to be manually specified in U2B are provided automatically for the developer by UML-RSDS, such as preconditions for *addrole* operations on injective associations [19].

Verification of UML transformations is also treated in [20], using algebraic interpretations, however this has limitations (simple patterns such as Value Object cannot be treated, for example) which our approach avoids. Modelling transformations in OCL is another alternative [21], however there are no proof tools available for OCL comparable to the tools available for B. Likewise, the approaches of [22] and [23], using abstract machines (ASMs) and graph transformations, respectively, are limited by the lack of proof support for these representations. B is a more semantically transparent (closer to ZF set theory) representation than ASM. We also provide direct support for models enhanced with OCL constraints, which these last two approaches do not.

Our approach to UML development is similar to that of [24], which carries out performance analysis of a system specified in UML, by means of a translation to a process algebra and analysis tools for this algebra. However this translation is manual, which increases the cost and the risk of introducing errors, compared to automated translations.

References

1. OMG: UML OCL 2.0 specification, ptc/2005-06-06 (2005) http://www.omg.org/uml/.
2. Glinz, M.: Problems and deficiencies of UML as a requirements specification language. In: Proceedings of 10th International Workshop on Software Specification and Design (IWSSD-10). (2000) 11–22
3. Naumenko, A., Wegmann, A.: Triune continuum paradigm and problems of UML semantics (2003)
4. Cook, S.: (UML semantics) MSDN Weblog, December 2004.
5. Spivey, J.: The Z Notation. Prentice Hall (1990)
6. Abrial, J.R.: The B Book: Assigning Programs to Meanings. Cambridge University Press (1996)
7. Burch, J., Clarke, E., McMillan, K., Dill, D., Hwang, J.: Symbolic model checking: 10^{20} states and beyond. In: Proceedings of the Fifth Annual Symposium on Logic in Computer Science. (1990)
8. Lano, K.: (A compositional semantics of UML-RSDS) submitted to SoSyM, 2006.
9. Lewerentz, C., Lindner, T.: Formal Development of Reactive Systems. Case Study Production Cell. LNCS Vol. 891. Springer-Verlag (1995)
10. Behm, P., Benoit, P., Faivre, A., Meynadier, J.M.: Meteor: A successful application of B in a large project. In: Proceedings of FM'99: World Congress on Formal Methods in the Development of Computing Systems. (1999)
11. B-Core (UK) Limited: B-toolkit manuals (1999) http://www.b-core.com.

12. Engineering, C.S.: (Atelier B) http://www.atelierb.societe.com/, 2004.
13. Bicarregui, J., Lano, K., Maibaum, T.: Objects, associations and subsystems: a hierarchical approach to encapsulation. In: ECOOP 97, Springer-Verlag LNCS (1997)
14. Lano, K., Clark, D., Androutsopoulos, K.: Formalising inter-model consistency of the UML. In: UML 2002, Workshop on Consistency Analysis of UML. (2002)
15. Androutsopoulos, K.: (Verification of reactive system specifications using model checking) PhD thesis, King's College, 2004.
16. Snook, C., Butler, M.: U2B – a tool for translating UML-B models into B. In J., M., ed.: UML-B Specification for Proven Embedded Systems Design. Springer-Verlag (2004)
17. project, R.F.I.: (2006) http://rodin.cs.ncl.ac.uk.
18. Kim, S., Carrington, D.: A formal mapping between UML models and Object-Z specifications. In: ZB2000, Springer-Verlag LNCS Vol. 1878 (2000)
19. Butler, B., Leuschel, M., Snook, C.: Tools for system validation with B abstract machines. In: ASM 2005 - International Workshop on Abstract State Machines, Paris. (2005)
20. Kosiuczenko, P.: (Redesign of UML class diagrams: a formal approach) Munich University, 2003.
21. T., C., A., E., M., G., P., S., J., W.: Modelling language transformations. L'Objet **9** (2003) 31–51
22. Chen, K., Sztipanovits, J., Abdelwalhed, S., Jackson, E.: Semantic anchoring with model transformations. In: ECMDA. (2005)
23. Varro, D., Pataricza, A.: Automated formal verification of model transformations. In: CSDUML. (2003)
24. Bennett, A., Field, A.: Performance engineering with the UML profile for schedulability, performance and time: A case study. In: Proc. IEEE/ACM International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS). (2004)

Validation of Model Transformations - First Experiences using a White Box Approach

Jochen M. Küster[1] and Mohamed Abd-El-Razik[23⋆]

[1] IBM Zurich Research Laboratory, Säumerstr. 4, 8803 Rüschlikon, Switzerland
jku@zurich.ibm.com
[2] IBM Cairo Technology Development Center, El-Ahram, Giza, Egypt
[3] Department of Computer Science, American University in Cairo, Egypt
mohrazik@aucegypt.edu

Abstract. Validation of model transformations is important for ensuring their quality. Successful validation must take into account the characteristics of model transformations and develop a suitable fault model on which test case generation can be based. In this paper, we report our experiences in validating a number of model transformations and propose three techniques that can be used for constructing test cases.

1 Introduction

The success of model-driven engineering generates a strong need for techniques and methodologies for developing model transformations. How to express model transformations and build appropriate tool support is a widely discussed research topic and has led to a number of model transformation languages and tool environments.

For practical use in model-driven engineering, the quality of model transformations is a key issue. If models are supposed to be semi-automatically derived using model transformations, then the quality of these models will depend on the quality of model transformations. Proving correctness of model transformations formally is difficult and requires formal verification techniques. An alternative approach widely applied in the industry is validation by testing. Today, it is common practice to apply large-scale testing for object-oriented programs using tools such as JUnit.

Model transformations can either be implemented as programs (e.g. in Java) or using one of the available transformation languages (e.g. [1–4]). In both cases, they require a special treatment within testing. One of the key challenges for testing model transformations is the construction of 'interesting' test cases, i.e. those test cases that show the presence of errors. For black box testing of model transformations, the meta model of the input language of the transformation can be used to systematically generate a large set of test cases [5, 6]. If the result of the model transformation is supposed to be executable, a possible testing approach is to take the output of a model transformation and to test whether it is executable [7]. By contrast, a white box approach to testing takes into

⋆ Part of this research was conducted while at the IBM Zurich Research Lab.

account design and implementation of the model transformation for constructing test cases. Compared with the extensive work on model-based testing of reactive systems (see Utting et al. [8] for a taxonomy and tool overview), testing of model transformations can still be considered to be in its early stages.

In this paper, we present our first experiences with a white box model-based approach to testing of model transformations. Our techniques have been developed while implementing a set of five model transformations for business-driven development [9, 10] which are used in a model-driven engineering approach for business process modeling. We propose three techniques for constructing test cases and show how we have used them to find errors in our model transformations.

The paper is structured as follows: We first introduce the idea of business-driven development and discuss the motivation for designing our model transformations in Section 2. Then we elaborate on our approach to design and implementation of these transformations in Section 3. In Section 4, we introduce three techniques for constructing test cases and explain how we apply them to validate our transformations. We conclude with a discussion of related work and conclusions drawn from our experience.

2 Model Transformations for Business Process Models

The field of business process modeling has a long standing tradition. Business-driven development is a methodology for developing IT solutions that directly satisfy business requirements. The idea includes that business process models are iteratively refined and transformed using model transformations, to move semi-automatically from a higher to a lower abstraction level.

We present business process models in the notation of IBM's WebSphere Business Modeler [11], which is based on UML 2.0 activity diagrams [12]. The language supported by the WebSphere Business Modeler makes some extensions to UML and can be considered as a domain-specific language for business process modeling. In these models, we distinguish *task* and *subprocess* elements. While tasks capture the atomic, not further dividable activities in the business process models, subprocesses can be further refined into more subprocesses and tasks. *Control and data flow* edges connect tasks and subprocesses. The control and data flow can be split or merged using *control actions* such as *decision*, *fork*, *merge*, and *join*. Process start and end points are depicted by *start* and *end nodes*. In addition, the language also contains a number of specific actions such as *broadcast* for broadcasting signals and *accept signal* for receiving signals or *maps* for mapping input data to output data.

In the language supported by the WebSphere Business Modeler, pin sets (based on parameter sets in UML2) are used for expressing implicit forks and joins as well as decisions and merges. Although these constructs leave a lot of freedom to the developer, they are problematic for complex transformations. As a consequence, we distinguish between models that only use control actions

and those that only use pin sets. A model in the Control Action Normal Form (CANF) requires that an action only has at most one pin set with exactly one pin in it [13]. A model in the Pinset Normal Form (PNF) requires that all forks, joins, decisions and merges are expressed implicitly using pin sets [13].

To support the idea of business-driven development for business process models, we have designed and implemented a number of model transformations for business process models (see Koehler et al. [10] for a detailed overview). The goal of these transformations is to enable a model-driven approach within business process modeling:

- the *Control Flow Extraction* transformation transforms a business process model with control and data flow into a process model with control flow only,
- the *Data Container Assignment* transformation transforms a business process model without data flow into a process model with data flow,
- the *Cycle Removal* transformation transforms a business process model with unstructured cycles into a process model with structured cycles only [14],
- the *Control Action to Pinset* transformation [13] transforms a business process model into the Pinset Normal Form, and
- the *Pinset to Control Action* transformation [13] transforms a business process model into the Control Action Normal Form.

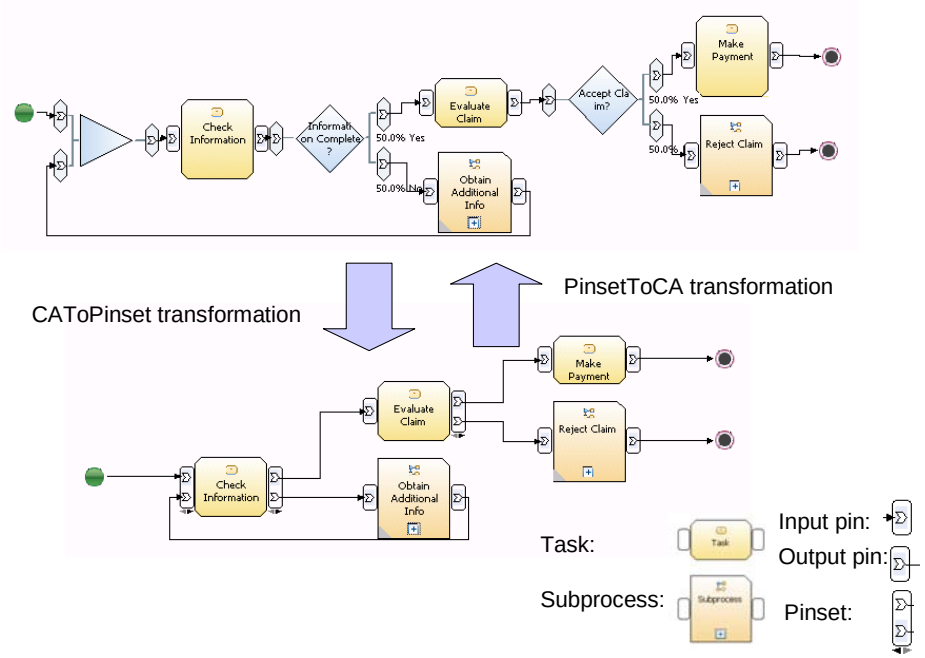


Fig. 1. Example of a process model in both forms

Figure 1 shows an example of a process model in the Pinset Normal Form (lower model) and in the Control Action Normal Form (upper model). All pre-

viously mentioned transformations have been implemented as Eclipse plugins to the WebSphere Business Modeler. In the following, we will concentrate on the Control Action to Pinset (CAToPinset) transformation, although we have applied similar techniques to other transformations. First, we will discuss our approach to design and implementation, and then elaborate on testing the transformations.

3 Design and Implementation of Model Transformations

For design and implementation of the model transformations introduced, we apply an iterative approach [15] that consists of producing a high-level design which is then used as a basis for the implementation.

The high-level design of a model transformation aims at producing a semi-formal description of a transformation, abstracting from its details such as all possible cases to be supported. As such, it can be considered as an early design in the elaboration phase, following the terms of the Rational Unified Process [16]. This high-level design provides an incomplete description and is not executable. The main objective of this activity is to capture the fundamentals of the transformation graphically to produce a description that can then be used for discussions among the developers.

A model transformation within high-level design is specified with a set of *conceptual transformation rules* $r : L \rightarrow R$, each consisting of a left and right side. The left side L and right side R show subsets of the source and target models for the transformation respectively. Concrete syntax of the underlying modeling languages is used, depicting how a part of the source model resembling the left side L is replaced by the part of the model described by R. In addition to elements from concrete syntax, those elements that are considered to be abstract are represented using additional abstract elements. These elements will typically be refined in later design phases or during implementation.

In Figure 2, rules of the CAToPinset transformation are shown. In addition to concrete syntax elements such as the *fork*, abstract elements are used, such as an abstraction for the node type. Overall, the rules abstract from the details such as the number of pins in a pin set, the number of outgoing or incoming edges, the type of the nodes and the type of the edge (control or data flow). Nevertheless, the main idea of each transformation rule is captured. For example, rule r_1 removes a fork, creates a new pin within the pin set of A, and connects the edges outgoing from the fork directly to the pins of A.

In general, different ways of implementing a model transformation exist. A pure model-driven approach consists of using one of the existing transformation engines, e.g. supporting the language QVT [17]. This has the advantage that the developer is given an environment that allows a definition of the transformation in a transformation language. In our case, we decided to implement the transformations directly in Java. This target implementation was then packaged as an Eclipse plugin and executed in the WebSphere Business Modeler.

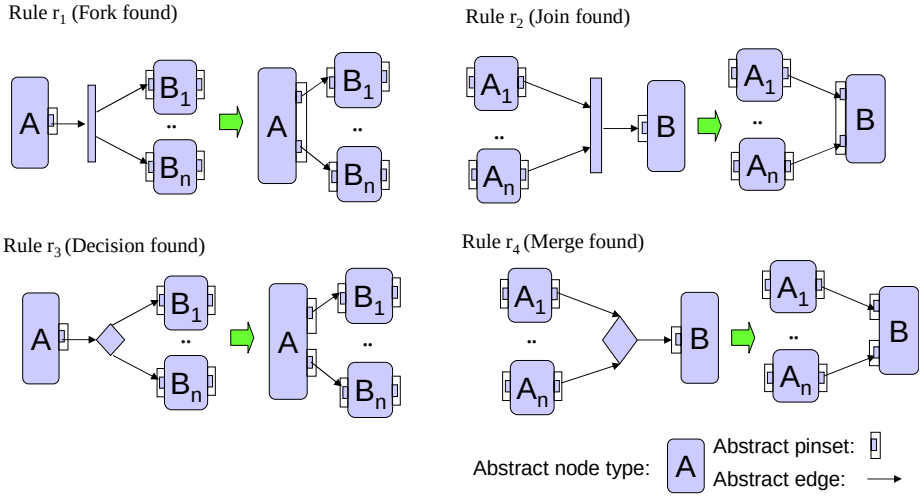


Fig. 2. Rules of the Control Action to Pinset transformation

In both cases, the conceptual rules of the transformation have to be refined by identifying the different cases they have abstracted from and defining how to handle them. For example, rule r_1 in Figure 2 has to be specified in more detail to take into account the possibility of data flow along the edges, the possibility of having multiple edges and special cases where parts of the fork are unconnected. In addition, the rule has to be refined with regards to the different possible node types for A and B_1 to B_n. In our case, possible node types include *start* and *end nodes*, *task*, *subprocess*, loop nodes such as a *while* loop, all control action nodes, and a number of specific action nodes such as the *broadcast* node. It is because of this number of model elements together with attached constraints that the transformation, which might look trivial at the conceptual level, requires some effort during implementation as well as thorough testing.

4 Systematic Testing of Transformations

Along the line of general principles of software engineering [18], we can distinguish between testing in the small and in the large. Testing in the small applied to model transformations can be considered as testing each transformation rule whereas testing in the large requires testing of each transformation.

For both types of testing, challenges of testing specialized to model transformations can be expressed as follows (adapted from [19]):

- the generation of test cases from model transformation specifications according to a given coverage criterion,
- the generation of test oracles to determine the expected result of a test, and
- the execution of tests in suitable test environments.

In our approach to model transformation development, the third challenge is easy to overcome because we can execute tests directly in our development

environment. The main challenges are the first and second ones because the model transformation specification in our case is based on the informal conceptual rules introduced above. In the following, we will show how we can partially overcome these challenges. First, we will discuss common types of errors that we have encountered when implementing the transformations. Then we discuss three techniques for test case generation and discuss cases where the test oracle problem is easy to overcome.

4.1 Fault model for model transformations

A fault model describes the assumptions where errors are likely to be found [20]. Given our approach to model transformation development, we can apply a model-based testing approach that takes into account the conceptual transformation rules as models. Based on our experience, the following errors can occur when coding a conceptual transformation rule:

1. *Meta model coverage*: the conceptual transformation rule has been coded without complete coverage of the meta model elements, leading to the problem that some input models cannot be transformed (e.g. the rule only works for certain node types, only for control flow edges, or only for one edge between two tasks but not for two edges).
2. *Creation of syntactically incorrect models*: the updating part of the transformation rule has not been implemented correctly. This can lead to models that do not conform to the meta model or that violate constraints specified in the meta model of the modeling language.
3. *Creation of semantically incorrect models*: the transformation rule has been applied to a source model for which it is not suitable, i.e. the result model is syntactically correct but it is not a semantically correct transformation of the source model.
4. *Confluence*: The transformation produces different outputs on the same model because the transformation is not confluent. This also includes the possibility that the transformation leads to intermediate models that cannot be transformed any further because non-confluence of the transformation has not been detected and treated.
5. *Correctness of transformation semantics*: the transformation does not preserve a desired property that has been specified for the transformation. Possible properties include syntactic and semantic correctness (see above) but also refinement or behavioral properties such as deadlock freedom.
6. *Errors due to incorrect coding*: there are also errors possible that cannot be directly related to one of the other categories. These errors can be classical coding errors.

Often, there is an interplay between meta model coverage and syntactic correctness. A meta model coverage error can lead to a syntactically incorrect model. The challenge in all cases is how to systematically generate test cases and how to

create the appropriate test oracles. Errors due to incorrect coding are indirectly found when testing for the first four types of errors. In addition, further techniques such as code walk-throughs can be applied. In the following, we introduce three techniques and discuss how they can be applied to find different types of errors. The last two types of errors are not explicitly dealt with in this paper and are left to future work.

4.2 Meta model coverage testing

In our approach to model transformation development, a given conceptual rule can be transformed into a *meta model template*. The idea of a meta model template is to be able to create automatically template instances that represent suitable test cases.

In the transition from a conceptual rule to a meta model template, abstract elements must either be made concrete or must be resolved by parameters together with a parameter set. To identify for each parameter in the conceptual rule the possible parameter values, the meta model of the underlying modeling language must be taken into account.

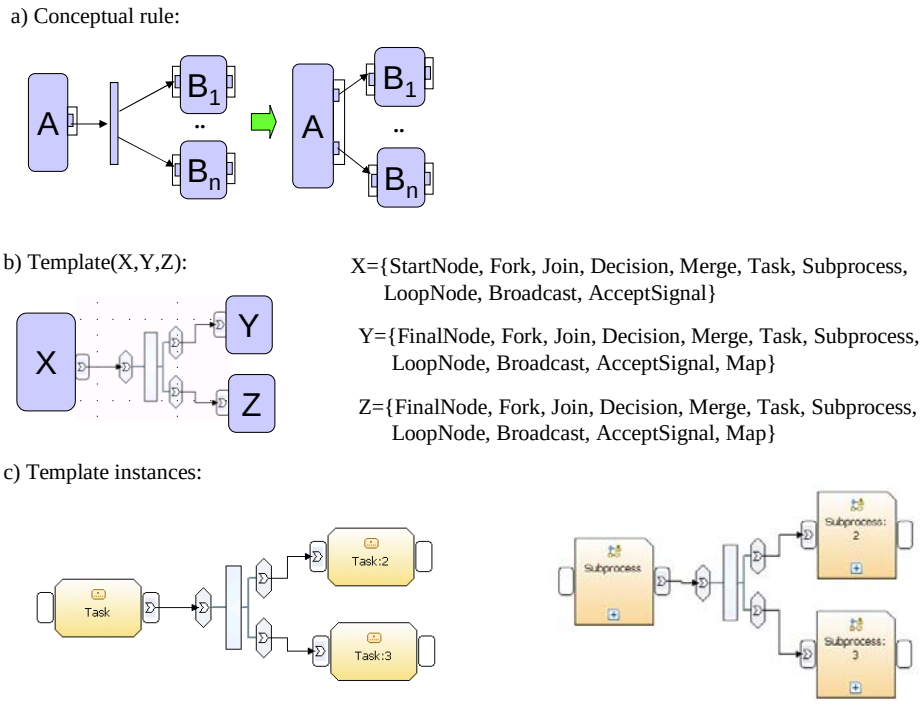


Fig. 3. Conceptual rule, meta model template and possible instances

Figure 3 b) shows a meta model template derived from rule r_1 shown in Figure 3 a). We make concrete the number of available nodes B_1, .. ,B_n and fix it to be $n = 2$. Further, we also fix the pin set structure of the nodes.

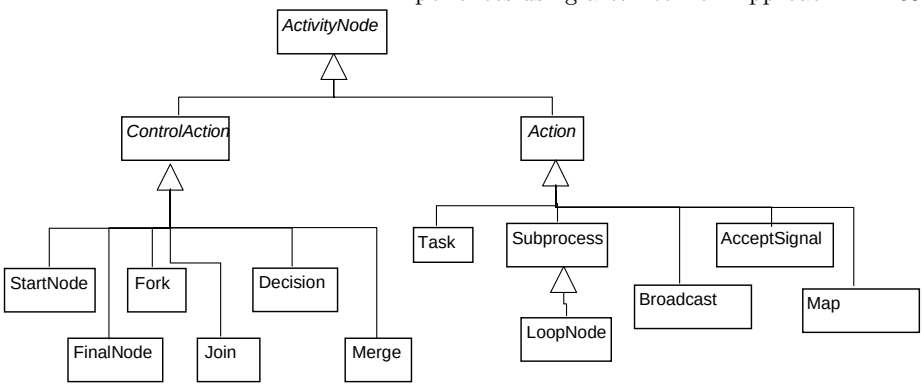


Fig. 4. Abridged metamodel extract

The remaining abstraction of the nodes is parameterized by the possible meta model classes. These can be identified when looking at the meta model (shown in Figure 4) and must be captured for each parameter in the meta model template. Figure 3 c) shows two template instances derived from the template.

Due to the abstraction process, one conceptual transformation rule can give rise to a number of different meta model templates. Figure 5 shows another template and possible instances. Here, we assume a different edge structure between the nodes. Note that when specifying the parameters for X, Y, Z one has to take into account well-formedness constraints of the language which e.g. do not allow that X includes the *StartNode*.

It is important to realize that meta model coverage testing is a classical case where white box testing is very powerful. This is because from each rule a number of templates can be derived that together can ensure a high degree of meta model coverage (per rule). If we obtain meta model coverage for each rule, we can deduce meta model coverage for the entire transformation.

After meta model templates have been defined, automatic generation of template instances yields a set of test cases for the transformation rule for which the template has been defined. Both the systematic instantiation of the templates and the testing can be automated. In the context of our work, a straightforward generation of templates has been implemented [13] that requires specification of the template and the suitable parameters. Based on this, a number of test cases is then generated automatically.

Beyond finding meta model coverage errors, meta model coverage testing can also be applied to find both syntactic and semantic correctness errors as well as errors due to incorrect coding. For syntactic correctness, the test oracle is the tool environment which in our case can detect whether the transformation result is syntactically correct. With regards to semantic correctness, each result must be manually compared and evaluated.

A remaining question is whether each rule needs to have complete coverage of the meta model or whether sometimes partial coverage can be tolerated if it is stated by suitable precondition constraints. An extension of the plain meta

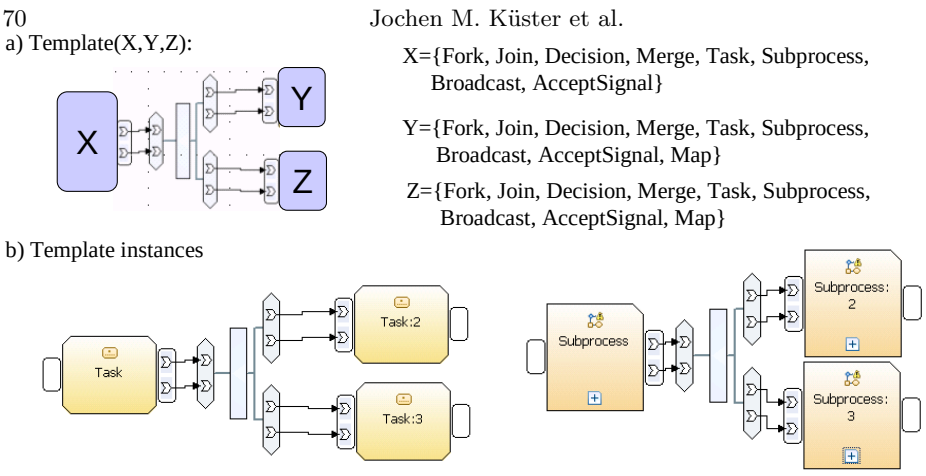


Fig. 5. Meta model template and possible instances

model coverage approach can take precondition constraints into account and give rise only to those test cases that fulfill the precondition constraints.

Meta model coverage testing is a powerful mechanism and can also be used for partially ensuring that constraints hold for the model that is created by the transformation. However, as test cases for meta model coverage are derived directly from a transformation rule, this technique has its limitations for those cases in which constraints are formulated for a number of model elements: If these model elements are not part of a certain rule, no test case generated using meta model coverage testing will be a suitable test case. This is why in the next section we present a technique that, given a constraint, aims at construction of test cases for this particular constraint.

4.3 Using constraints for construction of test cases

Typically, the meta model of a language also specifies well-formedness constraints. These constraints can be expressed using the Object Constraint Language (OCL) or in natural language. Violations of constraints give rise to syntactic correctness errors. As constraints can be violated by the interplay of several transformation rules, they cannot be completely detected by meta model coverage testing.

As a consequence, we believe that existing constraints specified in the language specification should be used to construct interesting test cases that aim at discovering errors due to the violation of constraints. As a transformation changes model elements, it needs to be tested that all constraints that may be violated due to the change hold after applying a transformation. We can test constraints both on the rule and transformation level.

After identification of the changed model elements, we take those constraints into consideration that are dependent on the model elements changed. A constraint is independent of a model element if the existence or value of the model

element instance does not influence the value of the constraint, otherwise it is dependent.

The idea to construct test cases to ensure constraints after application of the transformation is then as follows:

- Identify model elements changed by the transformation.
- Identify constraints that are dependent on these model elements.
- For each constraint, construct a test case that checks validity of the constraint under the transformation.

The test oracle for these tests is again the execution environment which in our case checks the constraints after application of the transformation.

An important issue is how we can detect which model elements are changed by the transformation, in the absence of a complete specification of the transformation rules. Partially, these elements can be detected when regarding the conceptual rule. At the same time, one can also obtain this information directly from the programmer.

With regards to the CAToPinset transformation, the model elements changed by r_1 are the pin set of A, because r_1 extends the pin set by adding an additional pin. Furthermore, edges are affected because r_1 changes their source or target nodes. In a similar way, we can find model elements changed by the other rules.

In our example of business process models, some of the constraints that are dependent on the changed model elements are:

- C1: A final node has one incoming edge.
- C2: An initial node has one outgoing edge of type control flow.
- C3: A Loopnode has one regular output pin set.
- C4: A Map has at least one output object pin.

All constraints are concerned with edges or with pin sets and are thus dependent on the changed model elements.

Given a constraint, we construct a test case for it as follows: Constraints can be divided into positive constraints requiring the existence of model elements and negative ones requiring the non-existence of model elements. In both cases, we try to create test cases that, after the transformation has been applied, can result into a violation of the constraint.

For example, with regards to constraint C_1, which requires that a final node has one incoming edge, we try to create a test case that after transformation results in the situation that the final node has two incoming edges. Figure 6 a) shows such a test case. An incorrect implementation will simply remove the *join* node and try to reconnect the incoming edges to the final node, which of course results into a syntactically incorrect model. Figure 6 b) shows a test case for C_2 (removal of the *join* node can lead to the creation of a *String* data flow edge from the start node, if incorrectly coded). In Figure 6 c)d) we present similar test cases for the constraints C3 and C4. All of these test cases have revealed errors in the implementation of the model transformation CAToPinset.

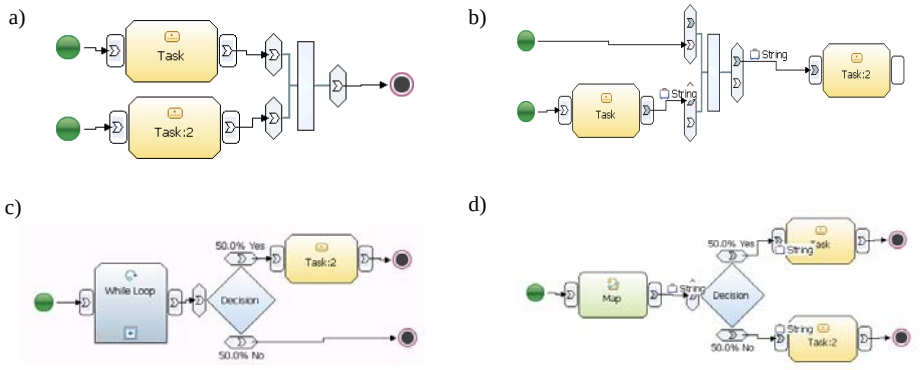


Fig. 6. Test cases for constraints

4.4 Using rule pairs for testing

Another source of errors arises from the interplay of rules: The application of one rule at some model element in the model might inhibit the application of another rule at the same model element. The property of confluence requires that the application of transformation rules on the same or an equivalent model yields the same result. As stated in [21], confluence of transformations need not always be ensured. However, it is important to detect whether the overall transformation is confluent because this can cause very subtle errors that are difficult to detect and reproduce. Confluence errors can give rise to syntactic as well as semantic errors.

In theory, the concept of parallel independence [22] of two rules has been developed which requires that all possible applications of the two rules do not inhibit each other i.e. it is always the case that if one rule r_1 was applicable before applying r_2 it is also applicable afterwards.

If two rules are not parallel independent, they might give rise to confluence errors. To detect such errors at design time, we have discussed in [21] a set of criteria which are based on the construction of critical pairs. The idea of a critical pair is to capture the conflicting transformation steps in a minimal context and analyze whether a common successor model can be derived. For exact calculation of critical pairs, a complete specification of the rules is required, e.g. in one of the model transformation languages.

In testing, the challenge is to construct test cases systematically that lead to the detection of confluence errors. In our approach, a complete specification of the transformation rules is not available. We can still use the conceptual rules for construction of test cases as follows: Based on the idea of critical pairs, we argue that it is useful to construct systematically all possible overlapping models of two rules. These overlapping models can represent a critical pair and can thus be used to *test* for the existence of a confluence error.

The overlapping models can be constructed systematically. The idea is to take the left sides of two rules and then calculate all possible overlaps of model elements. Based on an overlap, a model is constructed which joins the two models

at the overlapping model elements. If the overlapping model is syntactically incorrect, it is discarded. Otherwise, it is taken as a test case.

For example, for rules r_1 and r_3 in Figure 2 one possible overlap is to identify the node B_1 of r_1 with node A of rule r_3. The result is shown in Figure 7 a), assuming $n = 2$, a *task* node type for all nodes and a simple edge structure. Figure 7 c) shows another test case constructed from overlapping the rules. This test case gave rise to a confluence error because removing the *fork* leads to the construction of a pin set with two pins at the *decision* which is invalid and leads to an execution error because in our environment the construction of invalid intermediate models is not possible. If the fork is removed first, then no invalid model is constructed. Note that in a different execution environment supporting invalid intermediate models, the test case would not lead to an execution error.

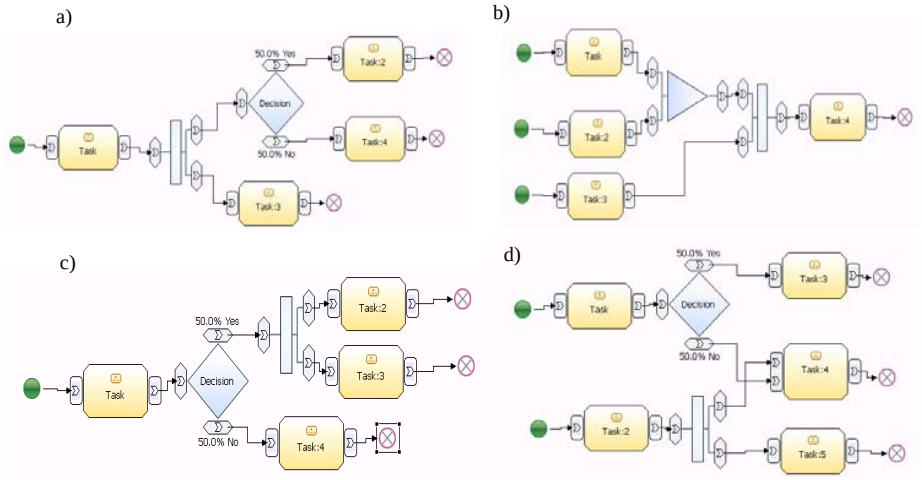


Fig. 7. Test cases for confluence (adapted from [13])

Figure 7 b) and d) show further test cases constructed from overlapping rules r_2 and r_4, and r_1 and r_3, respectively.

The idea of templates introduced above can also be used for rule pairs: If instead of two rules two template rules are used for constructing the overlapping rule, then the overlapping rule will be a template and can be instantiated automatically. This leads to an increased number of test cases that can be constructed automatically. To detect confluence errors automatically, the execution of test cases can require human intervention if no ability to compare results of test case execution automatically is available.

5 Related Work

With the advance of model-driven engineering, the idea of applying a model-driven approach to testing has also received increasing attention. Heckel and

Lohmann [19] describe how Web applications can be tested in a model-driven approach. One key idea is to use models as the test oracle for specifying the output of a test. A similar approach could be applicable for model transformations: If each transformation rule is completely specified in a transformation language, testing the implementation of the transformation can use the specification as a test oracle.

Mottu et al. [23] describe mutation analysis testing for model transformations. They identify four abstract operations within model transformations: Navigation within the model, filtering of elements, output model creation and input model modification. Based on these operations, they define mutation operators. For example, a possible mutation operator for navigation changes the association used within the navigation. A mutation operator for model creation is to replace a creation of an object with a parent class. Their mutation analysis can be used to ensure the quality of the test case set and has therefore a different focus compared to our work.

Fleurey et al. [5] describe an approach to generate test models. They first calculate the effective meta model for the transformation and then determine a coverage criterion based on this effective meta model which is similar to our concept of a meta model template for a rule. The coverage criterion is used for generating test models. Overall, their approach can be considered a black box approach for model transformation testing because they do not explicitly take different rules into account. Such an approach can be seen as complementing our work which is based on white box testing. We believe that if white box testing has succeeded (e.g. applying the techniques presented in this paper), it can be followed by large-scale black box testing.

Markovic and Baar [24] study how common refactoring operations on class diagrams such as moving an attribute or an operation can be expressed in the transformation language QVT and how OCL constraints are influenced by these operations. They describe a means how OCL constraints can be automatically refactored in these circumstances. On the contrary to their work, we use OCL constraints to construct test cases.

Recent work by Baudry et al. [25] summarizes model transformation testing challenges. They first discuss current limitations of black box testing by generating arbitrary input models. They also introduce the idea of testing the output of a transformation e.g. for a UML to Java transformation the Java output program can be checked by executing it. Another idea is to use patterns for specifying input or output of a transformation. This last idea is somewhat related to our approach, where the conceptual rule abstracts from the details of a transformation.

In the area of graph transformation which can be used as a formal basis for model transformation, there have been several approaches that deal with verification of graph transformations. With regards to testing, Darabos et al. [26] describe a way to generate test cases for graph pattern matching. They first extract logical criteria for matching a rule in form of a Boolean expression and then transform the Boolean expression into a combinatorial circuit. Using fault

injection into the circuit, mutations of the left side of the rule are created and form a set of test graphs. Our work can be seen as complementary to their work because their fault model is different from ours and we do not assume a graph-transformation-based implementation.

6 Conclusions

Validation of model transformations is a key issue to ensure their quality and thereby enables the vision of model-driven architecture become reality. In the context of business-driven development, model transformations are used for transforming more abstract models into more concrete ones and to move between different representations of models. In this paper, we have reported our first experiences with testing a set of model transformations for business process models systematically.

We have proposed three techniques which follow a white box testing approach. Using this approach, we have been able to significantly improve the quality of the model transformations under development. Both the meta model coverage technique as well as the construction of test cases driven by constraints has shown the existence of a number of errors. Rule pairs have indicated fewer errors, possibly due to the low number of rules.

If model transformation rules are completely specified already at the model level, using one of the model transformation languages, our techniques can also be applied but may require modifications. With regards to meta model coverage testing, one could manually abstract from a set of related rules and construct a conceptual rule which can be transformed into a template. The technique for using constraints for construction of test cases can make use of the transformation rules for automatically identifying the elements changed and can then be applied in the same way as described above. Further, a complete specification of rules also enhances the ability to use rule pairs for construction of test cases for confluence.

In our environment, we make the assumption that intermediate models must be correct with regards to the language specification. Sometimes, it is rather the case that either the model is correct only after application of the entire transformation or at certain checkpoints during the transformation. In such a case, the meta model coverage technique must apply the entire transformation to a generated test case.

There remain further challenges that we have not been able to address yet, for example, the automation of constructing test cases from OCL constraints. Here we see two possible improvements, firstly the automatic detection of constraints that could be violated by providing an algorithm that, given a meta model element, finds all attached constraints. Secondly, the automatic conversion of such a constraint into a possible test case. Future work also includes the elaboration of tool support in order to fully automate testing of transformations.

References

1. Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., Varró, D.: VIATRA: Visual Automated Transformations for Formal Verification and Validation of UML Models . In: Proceedings 17th IEEE International Conference on Automated Software Engineering (ASE 2002), Edinburgh, UK (2002) 267–270
2. Jouault, F., Kurtev, I.: Transforming Models with ATL. In Bruel, J.M., ed.: Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium, Montego Bay, Jamaica, October 2-7, 2005, Revised Selected Papers. Volume 3844 of LNCS. (2005) 128–138
3. Braun, P., Marschall, F.: BOTL - The Bidirectional Objekt Oriented Transformation Language. Technical report, Fakultät für Informatik, Technische Universität München, Technical Report TUM-I0307 (2003)
4. Karsai, G., Agrawal, A., Shi, F., Sprinkle, J.: On the Use of Graph Transformation in the Formal Specification of Model Interpreters. Journal of Universal Computer Science **9** (2003) 1296–1321
5. Fleurey, F., Steel, J., Baudry, B.: Model-Driven Engineering and Validation: Testing model transformations. In: Proceedings SIVOES-MoDeVa Workshop. (2004)
6. Ehrig, K., Küster, J.M., Taentzer, G., Winkelmann, J.: Generating Instance Models from Meta Models. Volume 4037 of LNCS., Springer (2006) 156–170
7. Dinh-Trong, T., Kawane, N., Ghosh, S., France, R., Andrews, A.: A Tool-Supported Approach to Testing UML Design Models. In: Proceedings of ICECCS'05, Shanghai, China. (2005)
8. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing. Technical report, Department of Computer Science, The University of Waikato (New Zealand), Technical Report 04/2006 (2006)
9. Mitra, T.: Business-driven development. IBM developerWorks article, http://www.ibm.com/developerworks/webservices/library/ws-bdd, IBM (2005)
10. Koehler, J., Hauser, R., Küster, J., Ryndina, K., Vanhatalo, J., Wahler, M.: The Role of Visual Modeleling and Model Transformations in Business-Driven Development. In: Proceedings of the 5th International Workshop on Graph Transformations and Visual Modeling Techniques. (2006) 1–12
11. : IBM WebSphere Business Modeler. (http:///www-306.ibm.com/software/integration/ wbimodeler/)
12. Object Management Group (OMG): UML 2.0 Superstructure Final Adopted Specification. OMG document pts/03-08-02. (2003)
13. Abd-El-Razik, M.: Business Process Normalization using Model Transformation. Master thesis, The American University in Cairo, in collaboration with IBM (2006) In preparation.
14. Hauser, R., Koehler, J.: Compiling Process Graphs into Executable Code. In: Proceedings of the 3rd International Conference on Generative Programming and Component Engineering,. Volume 3286 of LNCS., Springer-Verlag (2004) 317–336
15. Küster, J.M., Ryndina, K., Hauser, R.: A Systematic Approach to Designing Model Transformations. Technical report, IBM Research, Research Report RZ 3621 (2005)
16. Kruchten, P.: The Rational Unified Process. Addison Wesley (2003)
17. Object Management Group (OMG): Meta Object Facility (MOF) 2.0 Query/View/Transformation. Final Adopted Specification. OMG document ad/2005-11-01. (2005)

18. Ghezzi, C., Jazayeri, M., Mandrioli, D.: Fundamentals of Software Engineering. Prentice-Hall (1991)
19. Heckel, R., Lohmann, M.: Towards Model-Driven Testing. In: Proceedings of the International Workshop on Test and Analysis of Component-Based Systems (TACoS'03). Volume 82. (2003)
20. Binder, R.: Testing Object-Oriented System Models. Addison Wesley (1999)
21. Küster, J.M.: Definition and validation of model transformations. Software and Systems Modeling (2006) DOI: 10.1007/s10270-006-0018-8, to appear.
22. 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, Volume 1: Foundations. World Scientific (1997) 163–245
23. Mottu, J.M., Baudry, B., Traon, Y.L.: Mutation Analysis Testing for Model Transformation. In Rensink, A., Warmer, J., eds.: Model Driven Architecture - Foundations and Applications, Second European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10-13, 2006, Proceedings. Volume 4066 of LNCS., Springer (2006)
24. Markovic, S., Baar, T.: Refactoring OCL Annotated UML Class Diagrams. In Briand, L.C., Williams, C., eds.: Model Driven Engineering Languages and Systems, 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, Proceedings. Volume 3713 of LNCS., Springer (2005) 280–294
25. Baudry, B., Dinh-Trong, T., Mottu, J.M., Simmonds, D., France, R., Ghosh, S., Fleurey, F., Traon, Y.L.: Model Transformation Testing Challenges. In: Proceedings of IMDT workshop in conjunction with ECMDA'06, Bilbao, Spain. (2006)
26. Darabos, A., Pataricza, A., Varró, D.: Towards Testing the Implementation of Graph Transformations. In: Proceedings of the 5th International Workshop on Graph Transformations and Visual Modeling Techniques. (2006) 69–80

Towards Verified Model Transformations

Holger Giese[1], Sabine Glesner[2], Johannes Leitner[3],
Wilhelm Schäfer[1], and Robert Wagner[1]

[1] Department of Computer Science
University of Paderborn, D-33098 Paderborn, Germany
[hg|wilhelm|wagner]@uni-paderborn.de
[2] Faculty IV – Electrical Engineering and Computer Science
Technical University of Berlin, D-10587 Berlin, Germany
glesner@cs.tu-berlin.de
[3] Department for Computer and Information Science
University of Konstanz, D-78457 Konstanz
leitner@inf.uni-konstanz.de

Abstract. Model-driven software development (MDD) is seen as a promising approach to improve software quality and reduce production costs significantly. However, one of the problems in using MDD especially in the area of safety-critical systems is the lack of verified transformations. The verification of crucial safety properties on the model level is only really useful, if the automatic code generation is also guaranteed to be correct, i.e., the verified properties are guaranteed to hold also for the generated code. This particularly means to check semantic equivalence, at least to a certain extent between the model specification and the generated code. This paper addresses the problem of verifying that a given transformation ensures semantic equivalence between an arbitrary model in a given model specification language and the resulting programming language code. While the presented approach ensures that the transformation algorithm is correct, existing related work is restricted on verifying only the correctness of a particular transformation result.

1 Introduction

Model-driven software development (MDD) is seen as a promising approach to improve software quality and reduce production costs significantly. A major basis of such an approach is a usually domain-oriented modeling language which enables to abstract from implementation specific details and thus makes models (much) easier to develop and analyze than the final implementation. A significant additional benefit in terms of improved quality and reduced costs could be gained by the fully automatic transformation of a model-based system specification into executable code, if at all possible.

In developing safety-critical systems this approach is getting increasing attention, as model analysis has advantages over pure testing of implemented systems. Important required safety properties of a system under development could be

verified on the model level rather than trying to systematically test the absence of failures. Prominent failures in the past illustrate that testing often fails to detect malfunctioning by overlooking particular scenarios.

However, one of the problems in using MDD especially in the area of safety-critical systems is the lack of verified transformations. The verification of crucial safety properties on the model level is only really useful, if the automatic code generation is also guaranteed to be correct, i.e., the verified properties are guaranteed to hold also for the generated code. This particularly means to check semantic equivalence, at least to a certain extent between the model specification and the generated code.

While testing model transformations [1] and in particular approaches which exploit the specification of the code generator to derive critical test cases [2] are a valuable aid to ensure the quality of the transformation, they can only check a finite number of cases and thus fail to ensure the required semantic equivalence.

This paper addresses the problem of formally verifying that a given transformation ensures semantic equivalence between any model of the given model specification language and the resulting programming language code.

In compiler construction several approaches exist which check correctness of a transformation algorithm in particular or the correctness of the implementation when going from the code to lower level code or executables, see [3] for an overview. As a detailed example, on the level of source code transformations, Java and its transformation in Java byte code have been extensively investigated [4, 5]. The approach of proof-carrying code [6] is also weaker than what we intend to provide, because it concentrates only on the verification of necessary but not sufficient correctness criteria. The approach of program checking has been proposed by the Verifix project [7] and has also become known as translation validation [8, 9], recently also for loop transformations [10]. For an overview and for results on program checking in optimizing backend transformations cf. [11].

In contrast to these approaches for compiler construction, model to code transformations are characterized by rules and pattern matching like activation schemes of these rules, and thus the techniques employed in compiler construction are not directly applicable for their formal verification.

Although there exist many approaches for the specification and execution of model transformations, to the best of our knowledge, the only approaches addressing the problem of semantic equivalence in the above sense, at least to a limited extent, consider specific model instances and their translation results. Either specific correctness conditions are checked for both the original model and its transformation [12] or the semantic equivalence between both models is guaranteed by a bisimulation check [13].

Our approach realized in the Fujaba Tool Suite[4] is based on a formal specification technique, namely triple graph grammars (TGG) to specify a model to code transformation. The correctness of this specification and consequently the code generator is shown using the theorem prover Isabelle/HOL.

[4] www.fujaba.de

The next section will illustrate the use of a domain-oriented modelling language based on the example of a production line and its fairly complex control software. This realistic example of an existing industrial system will also be used to present the TGG and their application in building model to code transformations in Section 3. Section 4 describes the use of Isabelle/HOL to verify the transformation. Section 5 gives an account of the status of our work by listing required next steps and remaining open questions. Finally, Section 6 summarizes the work.

2 Modeling Approach

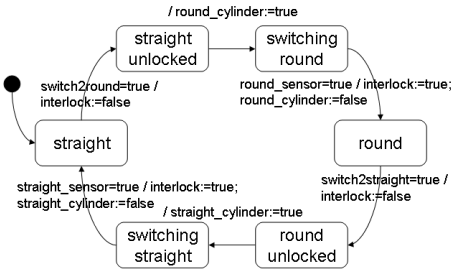
In this section, we provide a brief overview of the employed modeling approach [14] using a simple case study from the area of flexible production systems. It exemplifies the need for dependable model transformations and serves as a running example for explaining the specification of model transformations and their verification.

The substantial components of this modular system are working stations, straight and curved monorail tracks, as well as transfer gates (switches). For the transportation of materials and goods between the working stations self-propelled transportation units (shuttles) moving along the tracks are employed. The transportation units circulate on the main loop of the material flow system and can be stopped at stations or before curves and transfer gates only.

The decentralized production control system consists of PCs on the supervisory control level and Programmable Logic Controllers (PLC) on the cell level. The components' actuators and sensors are connected to the PLC via an Actuator Sensor Interface (ASI). The communication among PCs and PLCs is implemented by a multi-point interface (MPI, Siemens AG). Higher-level tasks, e.g., planning, order assignment, and coordination of local activities of all controllers are done at the supervisory control level. The PLCs on the cell level are responsible for the control of local components such as stations or transfer gates.

For the specification of the control software, we combined subsets of the Specification and Description Language (SDL) [15] and the Unified Modeling Language (UML) [16] into an executable graphical language [17]. In this language, a block diagram is used to specify the overall static communication structure where processes and blocks are connected to each other by channels and signal routes. For implementation purposes, the block diagram is automatically transformed to an initial class diagram. This class diagram can be refined and extended to an executable specification. For example, we can assign an automaton to model the reactive behavior of the control software. Fig. 1(a) presents a simple automaton for the control of the transfer gate used in our case study.

The specified automaton switches the transfer gate between the straight and the round direction. Initially, the transfer gate is switched to the straight direction and fixed by a mechanical interlock. When the condition *switch2round=true* becomes true, the interlock is disengaged by the action *interlock:=false*, and state



(a) Transfer gate automaton

```
VAR
  state : INT := 1; /* set to 'straight' */
END_VAR;

CASE state OF

 1: /* current state is 'straight' */
  IF switch2round=true THEN
   interlock:=false;
   state:= 2; /* set to 'straight unlocked' */
  END_IF;

 2: /* current state is 'straight unlocked' */
  round_cylinder:=true;
  state:=3; /* set to 'switching round' */

 ...

END_CASE;
```

(b) Generated PLC-code

Fig. 1. Automaton and generated PLC-code controlling a transfer gate

straight unlocked is entered. Thereafter, the triggerless transition fires and the appropriate action *round_cylinder:=true* is executed. This action activates the pneumatic cylinder responsible for turning the transfer gate into the round direction. The state *switching round* is left if the proximity sensor announces that the switching process completed. If the switching process was successful, the state *round* is entered and the interlock re-engaged. Now, the transfer gate can be switched back to the straight direction which is performed analogous to the described switching process for the round direction.

For the specification of the entire system, further controller automata are needed, e.g., automata to control the stopping and starting of shuttles at stations or before transfer gates. The sheer number of these automata and their interaction makes it hard to check manually whether the system functionality is defined correctly. As an example consider a requirement like "a shuttle never enters a transfer gate if the transfer gate is currently switching its direction". Our approach enables automated verification of such kind of safety-critical requirements using model checking [18]. After a successful verification, the controller automata need to be implemented. In our approach, the defined precise semantics of the automata model allows us to generate the PLC-code automatically.

PLCs are microprocessor systems that are widely used in industrial automation. The reason for their popularity is that they are robust and reliable. A PLC is connected to sensors and actuators: the former provide information on the state of the controlled component while the latter perform the actions prescribed by the control software. PLCs behave in a cyclic manner where each cycle follows three phases: (1) poll all inputs and store read values, (2) compute new output values, and (3) update all outputs. The repeated execution of this cycle is managed by the built-in real-time operating system. Thus, the control software has to compute the output values based on the read input values only.

For the automatic generation of PLC-code out of an object-oriented specification, we adapted our code generation mechanisms to produce Structured Text

(ST). Structured Text is a notation similar to PASCAL. It provides constructs such as if-then-else-conditionals and while-loops. As typical object-oriented concepts like inheritance or polymorphism are not supported, we implement the behavior of an automaton by simple switch-case constructs. The piece of code in Fig. 1(b) is an excerpt of the generated PLC-code for the automaton shown in Fig. 1(a) and gives a short impression on the translation in Structured Text for the states *straight* and *straight unlock*.

First, each state of the automaton is assigned a unique integer value. Then, we declare an integer variable *state* to keep the current state of the automaton which is handled in a case-statement. For our example automaton, the current state variable *state* is set to the initial state *straight* represented by the assigned integer value.

The outgoing transitions are encoded as if-statements with the transition guard as condition. For triggerless transitions, the if-statement is omitted if it is the one and only outgoing transition from that state. If they are more outgoing transitions with a guard, the triggerless transition is embedded in a if-else statement. Multiple triggerless transitions from one state are forbidden. The actions are realized as simple variable assignments. These variables, together with the variables from the conditions, are mapped by the compiler to the real addresses of the hardware. Note that the presented program is executed once in each cycle of the PLC. Thus, it is the body of an implicit loop-forever statement.

3 Model Transformations

To realize the modeling approach outlined in the previous section, we need a transformation which translates the given automata into executable PLC-code. Since code can be also viewed as a more detailed model of the software, we employ for this translation a model transformation technique based on triple graph grammars [19]. In this section, we give an overview of our model transformation approach and introduce the basics of the employed model transformation technique using our example from the previous section.

Fig. 2 gives an overview of our model transformation approach. The model transformation is specified by a number of transformation rules. The transformation rules are specified w.r.t. the metamodels of the source, the target, and an additional correspondence metamodel. From these rule specifications, a transformation engine is generated. The automatically derived engine transforms a source model into a target model yielding an additional correspondence model. This correspondence model enables a clear distinction between the source and the target model and holds additional traceability information about the applied mappings between the involved model elements. This information is used for further incremental updates if one of the models changes [20]. Hence, after an initial transformation the correspondence model serves as an additional input for following update transformations. In addition, since the employed transformation technique is bidirectional in nature, the source and target models can

change their roles and a reverse transformation, i.e., form the target to the source model, will be also possible. However, to keep things simple, in this paper we consider only transformations in the forward direction, i.e., from the source to the target model.

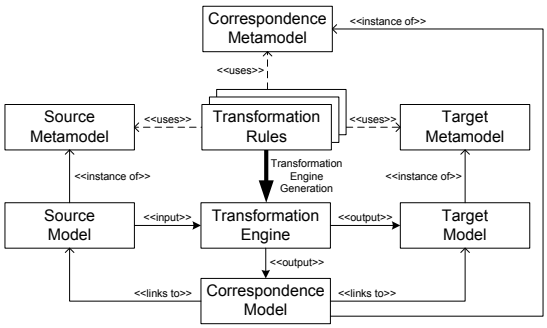


Fig. 2. Overview of the model transformation approach

In order to explain the specification technique of triple graph grammars for model transformation, we have to take a closer look at the involved metamodels. A metamodel defines the abstract syntax and static semantics of a modeling language. In Fig. 3, the automata metamodel, the metamodel defining the abstract syntax tree of the Structured Text programming language for PLCs, and the correspondence metamodel are shown.

In the automata metamodel shown in the upper left of Fig. 3, an *Automaton* consists of *States* and *Transitions*. A *Transition* connects *States* by its *outgoing* and *incoming* associations and has a *Trigger* as well as an ordered sequence of *Actions*. Some special states are the classes *InitialState* and *FinalState*. An automaton can have only one *InitialState* referenced by the directed association *initialState* but many *FinalStates* though they are rarly used for the specification of reactive systems.

For the specification of a triple graph grammar, we need an additional correspondence metamodel. It is shown in the upper right of Fig. 3. The metamodel defines the mapping between a source and a target metamodel by the classes *TGGNode* and *Object* and its associations *sources* and *targets*. Since all classes inherit implicitly from the *Object* class (not shown here), the correspondence model stores the traceability information needed to preserve the consistency between two models. In addition, the class *TGGNode* has a self-association *succ* which connects the correspondence nodes with their successor correspondence nodes. This extra link is used by our transformation algorithm.

The two described classes and their associations are essential for our transformation algorithm. However, further correspondence classes and refined associations can be added. In our example, we have added two additional correspondence classes, including the correspondence class *CorrNode* used in our example

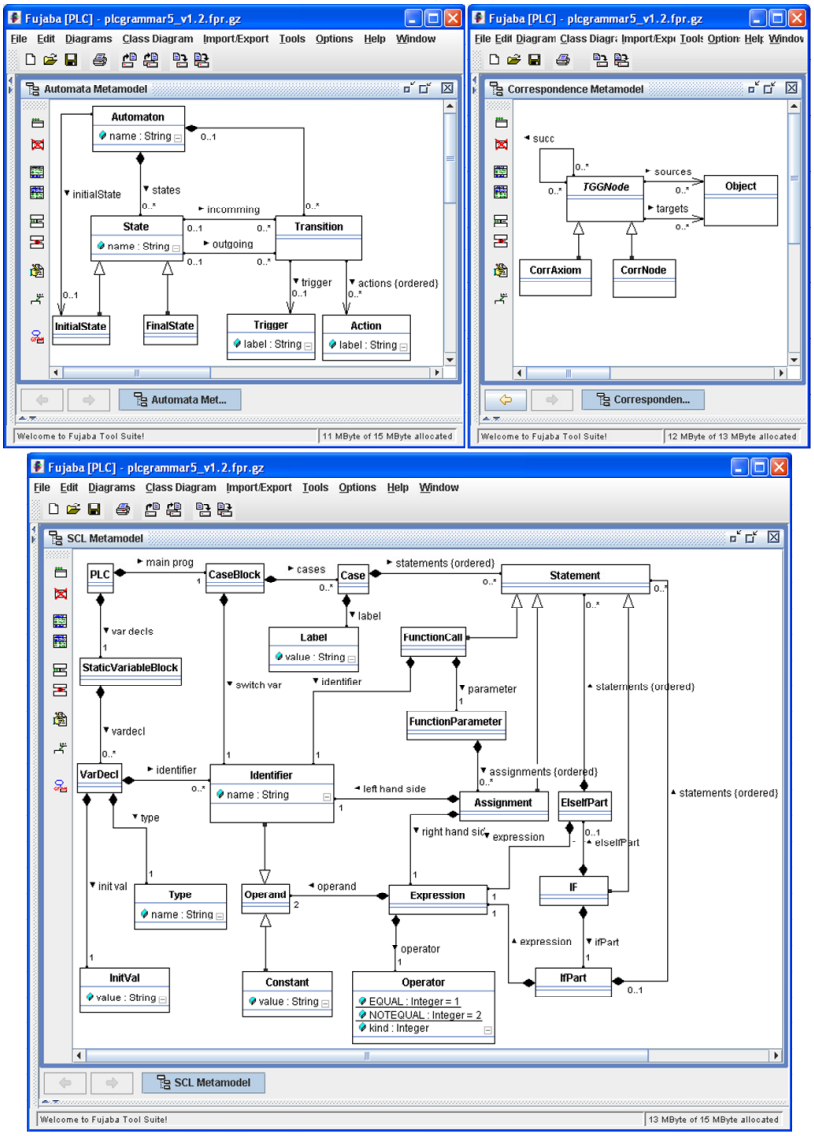


Fig. 3. Metamodels of the source, correspondence, and target model

rule (cf. Fig. 4). The additional correspondence classes increase the performance of our transformation algorithm but have no impact on the carried out formal verification.

The abstract syntax tree for Structured Text is defined by the metamodel shown in the lower part of Fig. 3. In fact, we are using only a subset of the language that is needed for the code generated out of automata. This subset was

extracted from the Structured Text grammar definition and comprises basically case-switch statements, if-then-else statements, assignment statements as well as expressions.

A program is represented by the class *PLC* that consists of one *StaticVariableBlock* and a *CaseBlock*. The class *StaticVariableBlock* has a to-many composition association to the class *VarDecl* which represents a variable declaration. A variable declaration comprises a *Type*, an *Identifier*, and an *InitVal* class representing the initial value of the identifier. The *CaseBlock* relates to an *Identifier* and is associated to many *Cases* that are represented by a *Label*. Each *Case* comprises a sequence of ordered *Statements*. A *Statement* is either a *FunctionCall* with a *FunctionParameter* whose result is assigned to an *Identifier*, an *Assignment* with a left-hand side *Identifier* and a right-hand side *Expression*, or an *IF* condition block. The *Expression* is defined by two *Operands* and an *Operator*. Up to now, only two kinds of operators are supported: equality and inequality. The *Operands* can be also represented by an *Identifier* or a *Constant*. An *IF* condition consists of an *IfPart* and an optional *ElseIfPart* which both have an *Expression* and embody an ordered sequence of *Statements*.

Given these three metamodels, a triple graph grammar for our example model transformation can be specified. In the following, we will explain the basic concepts with the help of our example and refer to [19] for a formal definition.

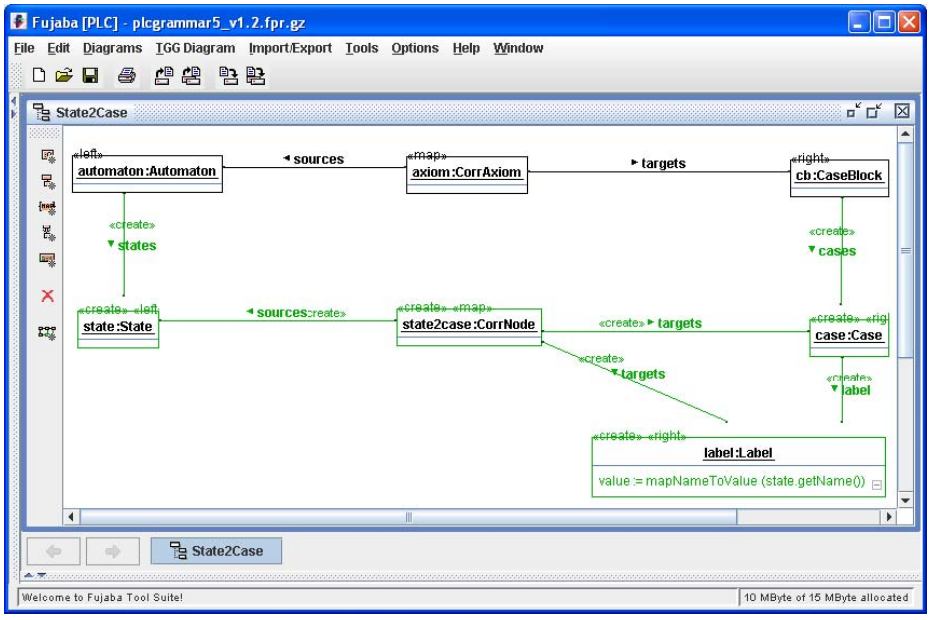


Fig. 4. A triple graph grammar rule mapping states to case statements

A triple graph grammar specification is a declarative definition of a bidirectional model transformation. In Fig. 4, a triple graph grammar rule in the Fujaba-notation is depicted. The rule specifies a consistent correspondence mapping between the objects of the source and the target model. In particular, the presented rule defines a mapping between a state and a corresponding case statement. The objects of the automaton are drawn on the left and the objects of the programming language are drawn on the right. They are marked with the ≪left≫ and ≪right≫ stereotypes respectively. The correspondence objects in the middle of the rule are tagged with the ≪map≫ stereotype.

The rule is separated into a triple of productions (source production, correspondence production, and target production), where each production is regarded as a context-sensitive graph grammar rule. A graph grammar rule consists of a left-hand side and a right-hand side. All objects which are not marked with the ≪create≫ stereotype belong to the left-hand side and to the right-hand side; the objects which are tagged with the ≪create≫ stereotype occur on the right-hand side only. In fact, these tags make up a production in Fujaba's graph grammar notation.

The source production on the left shows the generation of a new state and linking it to an automaton. The target production on the right shows the addition of a new case statement and its linking to the case block. In addition, the case block is equipped with a label to identify the state in the program. Since states in the program are encoded as integer values, a mapping function is used to translate the name of the state to a unique integer value. The correspondence production in the middle shows the relations between a state and the objects representing the case statement.

A graph grammar rule is applied by substituting the left-hand side with the right-hand side if the pattern of the left-hand side can be matched to a graph, i.e., if the left-hand side is matched all objects tagged with the ≪create≫ stereotype will be created. Hence, our example rule, in combination with additional rules covering other elements, can generate an automaton with the corresponding representation in the programming language by applying the production triples simultaneously. However, the transformation will not be executed this way. To execute a transformation, conceptually, we can assume that whenever a state is added to the automaton, a case statement with a corresponding label will be generated in the program. This way, the triple graph grammar rules define a transformation between automata and their representation in the programming language Structured Text.

The briefly described model transformation approach was realized in the Fujaba Tool Suite. For the visual specification of a triple graph grammar rule we use the TGGEditor (cf. Fig. 4) which is realized as a plug-in. This editor ensures conformance to the source, the correspondence, and the target metamodels. For this purpose, the required metamodels have to be specified in Fujaba as class diagrams (cf. Fig. 3).

The execution of a model transformation is done by the MoTE plug-in. MoTE is the abbreviation for Model Transformation Engine. It is the core li-

brary for the execution of triple graph grammars and can be also used without Fujaba. In order to execute a model transformation, we generate from each triple graph grammar rule Java code using Fujaba's code generation facilities. This code is compiled to executable transformation rules which are bundled into a single Jar archive file. The archive represents the catalog of transformation rules defining the model transformation specified by a triple graph grammar. Once the catalog is available, the transformation engine is complete and model transformations can be carried out.

As mentioned in Section 2, our approach verifies the specified automata w.r.t. crucial safety requirements. However, the proven properties can only be guaranteed to hold also for the implementation, if we can ensure that the implementation realizes the same behavior as the specified automata. Therefore, we have to ensure that the employed model transformation from the automaton model to the code model is correct (the implementation model must be semantically equivalent to the already verified automata model).

4 Verification

In this section, we describe our approach for the verification of triple graph grammar transformations in Isabelle/HOL. We show in more detail how to derive Isabelle/HOL representations from structures in Fujaba and outline the basic proof scheme.

In essence, we prove that the relation of semantic equivalence is a congruence with respect to an appropriate representation of the transformation rules. Fig. 5 extends the modeling overview from Fig. 2 with an illustration of our general proof scheme.

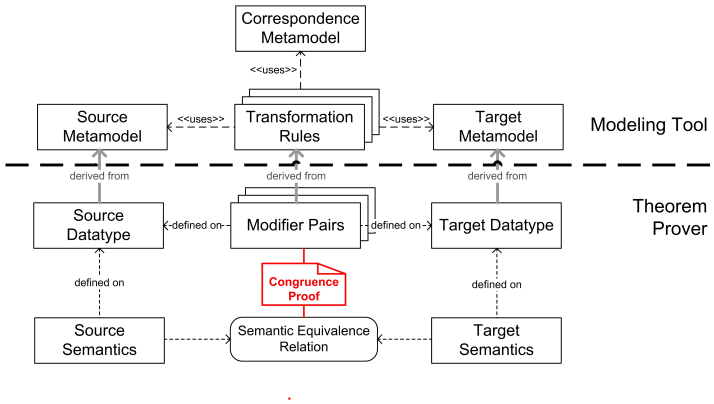


Fig. 5. Overview of the verified model transformation approach

Note that the model instances (as well as the transformation engine) shown in Fig. 2 are omitted here. Since we verify the correctness of transformation rules applied to *any* model of the specified type, these instances are irrelevant for the proof.

Model transformations are often formalized as instances of graph transformations. While this approach is intuitive and shifts the problem into an extensive and well-known theory, its realization in Isabelle/HOL poses a number of difficulties. Problems already arise when trying to formalize models as instances of metamodels in HOL. A metamodel entails a number of structural constraints on its instances, while a graph just consists of arbitrarily connected nodes. Metamodel constraints have to be expressed as additional axioms about the structure of the graph, like "Nodes with a type of `State` can only be directly connected to nodes of type `Transition`". Even small metamodels will result in graph types encompassed by long lists of such axioms. Defining semantics and conducting proofs on these structures is tedious and, more importantly, the derivation of axioms from metamodels is not straight-forward, has to be done manually and is thus error-prone.

For these reasons, we chose a different formalization of metamodels that comprises all the structural information directly in a type definition. This makes proofs simpler and significantly more compact. At first, we create a modified version of the metamodel with an ordered, tree-like structure. This structure can always be achieved by, for example, converting circular compositions to reference attributes. Fig. 6 shows the result for a part of the metamodel of the employed automata presented in Fig. 3.

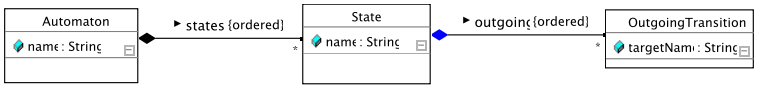


Fig. 6. Part of the modified metamodel for automata

This kind of "flattened" model can be mapped to a type in Isabelle/HOL using only constructs like records, lists and other primitive data types. The nature of this mapping is straight-forward and might be implemented as an automatic procedure in the future. The result for the above metamodel is:

```
record State =                          record OutgoingTransition =
  Identity   :: BaseType                  Target  :: BaseType
  Outgoing   :: OutgoingTransition list   Actions :: ActionType list
  FinalState :: bool                      Trigger :: TriggerType option
```

Primitive (i.e. algebraic) types are a core concept of Isabelle/HOL. On these types, we can easily define an operational semantics as a recursive function over the structure of the model. On this semantics, we define a bisimulation $\approx$, formalizing the notion of *semantic equivalence*. In many cases, semantic equivalence is just defined as (statewise) equality; however, different semantic

domains of source and target model might require a more abstract comparison. Our definition of semantics and semantic equivalence is shown in detail in [21, 22].

We view rules of a triple graph grammar not as specifications of transformations on a single graph, but as *pairs* of productions describing a way that two models are simultaneously modified. This allows for an easy and elegant formalization in Isabelle/HOL. Fig. 7 shows the pair of graph productions corresponding to the triple graph grammar rule that maps states to case statements (shown in Fig. 4).

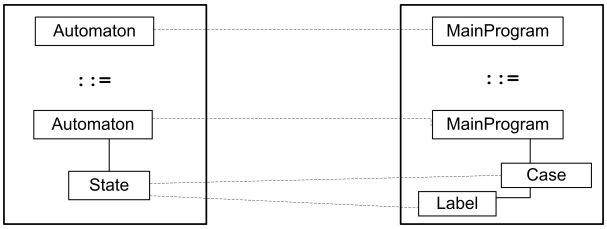


Fig. 7. View of the triple graph grammar rule in Fig. 4 as a pair of productions

This interpretation captures the bidirectional nature of TGGs by interpreting them as a grammar for the parallel evolution of source and target model. For each of these productions we can now define an operator, called *modifier*, on source and target model. We formalize the application of a transformation rule as a parallel application of the corresponding modifiers to both models. For example, for the productions above, we define modifiers for automata and PLC programs that add a state or a case statement, respectively:[5]

$$A \oplus s \equiv A(\!|\, \text{States} := (\text{States}\ A) \cdot s \,|\!)$$
$$P \oplus c \equiv P(\!|\, \text{MainProgram} := c \cdot (\text{MainProgram}\ P) \,|\!)$$

For the correctness of each transformation rule it then suffices to show that the application of the modifiers will not destroy semantical equivalence of models. For example, if we add a state to an automaton and a corresponding case block to a semantically equivalent PLC program in the Structured Text programming language, automaton and program remain equivalent:

$$A \approx P \implies (\ A \oplus s\ ) \approx (\ P \oplus \texttt{State2Case}(s)\ ) \tag{1}$$

[5] Here, the Isabelle/HOL operator $(\!|... := ...|\!)$ is used to update the specified member of a record. The operator $\cdot$ appends elements to lists.

In effect, we show that semantical equivalence is a congruence with respect to every transformation rule. The proof for the above lemma is straight-forward, since neither the new state nor the new code segment will be reachable. However, more complex rules require more elaborate proofs. For example, the proof for one variant of the TGG rule `Action2Code` that adds actions to transitions (and inserts PLC code in the appropriate location in the program) requires over 100 lines of proof code[6] in ISAR notation [23] and makes use of 15 additional helper lemmas. In total, the proof of the correctness of the transformation from automata to a PLC language contains approximately 1500 lines of proof code. Table 1 shows the distribution of lines of proof code for the different parts of the proof.

Table 1. Lines of ISAR proof code for different parts of the proof

Mapping		
	Automata formalization	170
	SCL formalization	259
	TGG rule formalization	302
Rule Correctness Proofs		
	Definition of semantic equivalence	42
	The `Axiom` rule	22
	The `State2Case` rule	81
	Two variants of `Transition2Code`	328
	Two variants of `Action2Code`	276

5 Next Steps

To realize the vision of MDD by means of verified model transformations, the presented results are a first step. We discuss in this section required next steps we plan to address.

Rule correctness lemmas of the simple form as presented in Section 4 will not always be provable. TGG rules are not applicable on arbitrary patterns in the source and target graph, but rely on the correspondence graph created by previous rules. For example, adding a transition to a state of an automaton and a corresponding code segment to a case-block of a PLC program will only preserve semantic equivalence if the state and the case-block themselves correspond to each other, i.e., were created by a pair of modifiers. In fact, in the example lemma (1) the correspondence is hidden in the Function `State2Case`, which creates a case block with the same identifier as the state s. This results in additional preconditions, which we call *correspondence preconditions*, for the rule

[6] Note that proofs in ISABELLE/HOL cannot be done automatically and that the vast majority of proof steps needs manual interaction.

correctness lemmas. At the moment, we tackle this problem by introducing the correspondence preconditions manually. We aim to show that these preconditions indeed result from previous rule applications. A possible solution makes use of the set of all models introduced by successive application of all the rules in a TGG grammar, which would enable us to conduct proofs about the relationships between rule applications.

In addition to a proof technique, also the methodological aspects of verifying model transformation have to be addressed. We therefore plan to elaborate the design and verification process for model transformations and develop automated or semi-automated tool support for the required activities where possible.

A first planned extension is to automatically derive the formalization of the metamodels and TGG rules in Isabelle/HOL which accounts for nearly 50% of the lines of proof code. We also want to explore how we can combine the interactive theorem proving with available automated verification approaches for finite and infinite graph transformations already present in Fujaba [24] in order to reduce the effort for the verification of a model transformation.

6 Conclusions

Model-driven software development, especially with domain-specific languages, is increasingly important to automatically develop software that adheres to its specification. In this paper, we have shown how model-driven software development is applied in the context of flexible production systems. These systems and their transformations are specified within the Fujaba Tool Suite using triple graph grammars (TGGs). TGGs are a special form of graph grammars that allow us to specify the parallel evolution of systems, namely of the source (or model) system and the target system (its implementation). We have presented results of ongoing work how such transformations can be formalized and verified in the Isabelle/HOL theorem prover. This is an important step towards fully verified model transformations, which are necessary to guarantee correctness of the generated implementations of the specified models.

References

1. Fleurey, F., Steel, J., Baudry, B.: Validation in model-driven engineering: Testing model transformations. In: Proc. of the First International Workshop on Model, Design and Validation, Rennes, November 2004. (2004) 29–40
2. Stuermer, I.: A contribution of graph grammar techniques to the specification, verification and certification of code generation tools. Electronic Notes in Theoretical Computer Science **72** (2002) 10
3. Glesner, S., Goos, G., Zimmermann, W.: Verifix: Konstruktion und Architektur verifizierender Übersetzer (Verifix: Construction and Architecture of Verifying Compilers). it - Information Technology **46** (2004) 265–276 Print ISSN: 1611-2776.
4. Klein, G., Nipkow, T.: Verified Bytecode Verifiers. Theoretical Computer Science **298** (2003) 583–626

5. Klein, G., Strecker, M.: Verified Bytecode Verification and Type-Certifying Compilation. The Journal of Logic and Algebraic Programming **58** (2004) 27–60
6. Necula, G.C.: Proof-Carrying Code. In: Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97), Paris, France (1997) 106–119
7. Goerigk, W., Dold, A., Gaul, T., Goos, G., Heberle, A., von Henke, F., Hoffmann, U., Langmaack, H., Pfeifer, H., Ruess, H., Zimmermann, W.: Compiler Correctness and Implementation Verification: The Verifix Approach. In Fritzson, P., ed.: Poster Session of CC'96, IDA Technical Report LiTH-IDA-R-96-12, Linkoeping, Sweden (1996)
8. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In Steffen, B., ed.: Proceedings of Tools and Algorithms for the Construction and Analysis of Systems, Lisbon, Portugal, Springer Verlag, Lecture Notes in Computer Science, Vol. 1384 (1998) 151–166
9. Necula, G.C.: Translation Validation for an Optimizing Compiler. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'00), Vancouver, British Columbia, Canada (2000) 83–94
10. Goldberg, B., Zuck, L., Barrett, C.: Into the Loops: Practical Issues in Translation Validation for Optimizing Compilers. In: Proceedings of the Workshop Compiler Optimization meets Compiler Verification (COCV 2004), 7th European Conferences on Theory and Practice of Software (ETAPS 2004), Barcelona, Spain, Elsevier, Electronic Notes in Theoretical Computer Science (ENTCS) (2004)
11. Glesner, S.: Using Program Checking to Ensure the Correctness of Compiler Implementations. Journal of Universal Computer Science (J.UCS) **9** (2003) 191–222
12. Varró, D., Pataricza, A.: Automated formal verification of model transformations. In Jürjens, J., Rumpe, B., France, R., Fernandez, E.B., eds.: CSDUML 2003: Critical Systems Development in UML; Proceedings of the UML'03 Workshop. Number TUM-I0323 in Technical Report, Technische Universität München (2003) 63–78
13. Narayanan, A., Karsai, G.: Towards verifying model transformations. In: 5th International Workshop on Graph Transformations and Visual Modeling Techniques, Vienna, 2006. Electronic Notes in Theoretical Computer Sience (2006) 185–194
14. Schäfer, W., Wagner, R., Gausemeier, J., Eckes, R.: An engineer's workstation to support integrated development of flexible production control systems. In Ehrig, H., Damm, W., Desel, J., Große-Rhode, M., Reif, W., Schnieder, E., Westkämper, E., eds.: Integration of Software Specification Techniques for Applications in Engineering. Volume 3147 of Lecture Notes in Computer Science (LNCS). Springer Verlag (2004) 48–68
15. International Telecommunication Union (ITU), Geneva: ITU-T Recommendation Z.100: Specification and Description Language (SDL). (1994 + Addendum 1996)
16. OMG 250 First Avenue, Needham, MA 02494, USA: Unified Modeling Language Specification Version 1.5. (2005)
17. Nickel, U., Schäfer, W., Zündorf, A.: Integrative specification of distributed production control systems for flexible automated manufacturing. In Nagl, M., Westfechtel, B., eds.: DFG Workshop: Modelle, Werkzeuge und Infrastrukturen zur Unterstützung von Entwicklungsprozessen, Wiley-VCH Verlag GmbH and Co. KGaA (2003) 179–195
18. Giese, H., Kardos, M., Nickel, U.: Integrating Verification in a Design Process for Distributed Production Control Systems. In: Proceedings of the 2nd International Workshop on Integration of Specification Techniques for Applications in Engineering (INT2002), Grenoble, France. (2002)

19. Schürr, A.: Specification of graph translators with triple graph grammars. In Mayr, E.W., Schmidt, G., Tinhofer, G., eds.: Graph-Theoretic Concepts in Computer Science, 20^{th} International Workshop, WG '94. Volume 903 of LNCS., Herrsching, Germany (1994) 151–163
20. Giese, H., Wagner, R.: Incremental model synchronization with triple graph grammars. In Nierstrasz, O., Whittle, J., Harel, D., Reggio, G., eds.: Proc. of the 9th International Conference on Model Driven Engineering Languages and Systems (MoDELS), Genoa, Italy. Lecture Notes in Computer Science (LNCS), Springer Verlag (2006)
21. Leitner, J.: Verifikation von Modelltransformationen basierend auf Triple Graph Grammatiken. Master's Thesis (*Diplomarbeit*), University of Karlsruhe (2006)
22. Blech, J.O., Glesner, S., Leitner, J.: Formal Verification of Java Code Generation from UML Models. In: Proceedings of the 3rd International Fujaba Days 2005: MDD in Practice, Technical Report, University of Paderborn (2005)
23. Nipkow, T.: Structured Proofs in Isar/HOL. In: Types for Proofs and Programs (TYPES 2002), Springer Verlag, Lecture Notes in Computer Science, Vol. 2646 (2003) 259–278
24. Becker, B., Beyer, D., Giese, H., Klein, F., Schilling, D.: Symbolic Invariant Verification for Systems with Dynamic Structural Adaptation. In: Proc. of the 28^{th} International Conference on Software Engineering (ICSE), Shanghai, China, ACM Press (2006)

Model Checking Dynamic and Hierarchical UML State Machines

Toni Jussila[1], Jori Dubrovin[2], Tommi Junttila[2], Timo Latvala[3], and Ivan Porres[4]

[1] Johannes Kepler Universität Linz
[2] Helsinki University of Technology
[3] University of Illinois at Urbana-Champaign
[4] Åbo Akademi University

Abstract. This paper presents a technique to model check UML specifications by translating UML models to the model checker SPIN. Our models consist of active UML classes, whose behavior is defined by hierarchical state machines. The intended application is to find errors in protocols communicating using asynchronous message passing. Compared to previous efforts using a similar approach, our novel points are the following. First, we consider a subset of UML that in our opinion is expressive enough for protocol models but allows a simpler translation to SPIN than existing work. Preliminary analysis of simple industrial models support our conclusions on the expressivity of our UML subset. Second, we present a powerful action language that is still amenable to automatic analysis. The action language is used to specify the effects of transitions, which may include dynamic creation of new objects. Finally, we discuss an even simpler SPIN translation for flattened UML state machines and compare it to the translation that supports hierarchy.

1 Introduction

Model-based approaches for system design have been studied for a long time. Advantages associated with model-based approaches are several. Models give designs a restricted implementation independence, and they also provide a convenient form of documentation. However, arguably the most important benefit is that the level of abstraction of the design is raised. This has many implications. Abstract models allow efficient communication of the design, since unnecessary details are hidden. They also facilitate testing and verification of the design at an early stage, the topic of this paper. The widely acknowledged benefit of this is that it is much cheaper to detect and correct software errors early in the design process.

We report preliminary results from a project, where the goal is to find errors in protocol designs using *model checking*. In our approach, the protocols are modeled using UML class diagrams and state machines. The Unified Modeling Language (UML) is a standardized graphical notation for modeling and

documenting object-oriented software and business processes. UML is also the most widespread software modeling language, and it is accepted in the industry as the standard language for software analysis and design. UML state machine models can fairly naturally capture protocol designs, where the communication is asynchronous and data can be abstracted with the class subtyping mechanism.

Our approach to model checking UML designs is based on using the state-of-the-art model checking tool, SPIN [1]. Models with assert specifications are automatically translated to SPIN's input language, and counterexamples can be simulated. More advanced properties can be specified in the SPIN model as temporal logic formulae.

Our work improves or differs from previous work in the following ways. The subset of UML we support has specifically been chosen to be expressive enough for our intended application, modelling protocols, yet it allows a precise and fairly simple formal semantics. We present an action language, which is used to specify effects of transitions, that is powerful and amenable to automatic analysis. Supported features include dynamic creation of objects, the usual flow constructs, and arithmetic expressions. We also discuss a simpler SPIN translation for flattened state machines and compare it to the translation that supports hierarchy.

2 Related Work

The idea of applying model checking on UML-type state machine models is not new. Latella et al. [2] present a translation from UML state machines to PROMELA, the input language of SPIN. They only allow a model to contain a single state machine. In another translation by Mikk [3], the input language is not UML but statecharts, which is a similar formalism with different semantics. Perhaps the closest work to ours is [4] which presents a tool called vUML that translates UML to PROMELA. vUML supports a larger subset of UML than our approach. This, however, has the effect that the translation of a UML state machine is more complex; it requires a code block that chooses the transitions to be fired and another block that models the effects of transitions. All these works have the limitation that no data attributes can be associated with objects or state machines. Consequently, there is no action language, and the only possible effect of a transition is to send signals with no parameters.

The Hugo project [5] also supports SPIN as a back-end to verify UML models. Their initial PROMELA translation was only feasible for very small models, and the current version of the tool follows ideas similar to those in vUML. To our knowledge, the translation is undocumented. In [6], SPIN is used to generate test cases from abstract state machines. The OMEGA project [7] has created a tool set focusing on real-time properties. Their approach is based on translating UML to the IF intermediate language that has several model checking back ends. The Rhapsody UML verification environment [8] supports model checking of UML state machines by translating the models to the input language of the

VIS symbolic model checker. Most UML constructs are supported, and the action language is a restricted subset of C++. Features that are not supported include deferred events and do activities.

3 UML Subset and Semantics

In our framework, a UML model contains classes, state machines, and deployment diagrams. Classes may contain instance attributes, i.e. data values associated with objects, and there may be associations between classes. Operation calls are not supported. Also, a class cannot be a subtype of another class, but we are planning to incorporate support for subtype relationships. We assume that all classes are active classes whose behavior is defined by behavioral state machines, discussed below. At run-time, objects communicate with each other asynchronously using signals. Signals may have associated parameter values. A deployment diagram is used to specify the initial configuration of objects.

3.1 A Well-Defined UML State Machine Language

Compared to standard UML state machines, we consider a subset. This is motivated by the need of a precise behavioral design language that can be verified efficiently.

UML offers two mechanisms for modeling concurrency: active objects communicating with signals, and orthogonal regions in the state machine of an object. We argue that the first approach is often preferable in an object oriented setting, where it is more natural to put emphasis on communicating objects (which can be dynamically created) instead of concurrent behavior within a single object. Indeed, the commercial UML tool Telelogic Tau [9] does not even allow orthogonal regions in state machines.

Our subset allows orthogonality, but we propose a restriction that no two transitions in orthogonal regions can be enabled by the same event. We argue that this violates the idea of orthogonality and creates complicated dependencies that are hard to understand and analyze. Together with execution semantics in which at most one completion transition (transition without an explicit trigger) is fired at a time, the restriction guarantees that at most one region changes its state in each step. This corresponds to interleaving execution semantics suitable for the SPIN model checker, and results in a simple implementation of the transition selection algorithm.

UML allows continuous do activities in states, but we omit them because the model checker executes models in discrete steps. Furthermore, we do not currently support history pseudostates, fork or join pseudostates, or entry or exit activities in states. These are advanced modeling concepts that could later be incorporated to our framework.

In the following, the structure of state machines is formalized.

3.2 Structure of State Machines

A state machine contains *states* and *transitions* between them. A *composite state* contains one or more orthogonal *regions*, which in turn contain substates. During execution, one or more states are *active*. If a composite state is active, then exactly one direct substate in each region is active. Whenever a substate is active, its containing composite state is also active.

Let Σ be a finite set of states consisting of simple states Σ_{simple}, initial pseudostates $\Sigma_{initial}$, choice pseudostates Σ_{choice}, final states Σ_{final}, and composite states $\Sigma_{composite}$, and let $\mathcal{R}$ be a finite set of regions. We define a *child relation* $\searrow$ such that if a region r of a composite state c directly contains a state s, then $c \searrow r$ and $r \searrow s$.

Definition 1. *A tuple* $H = \langle \Sigma_{simple}, \Sigma_{initial}, \Sigma_{choice}, \Sigma_{final}, \Sigma_{composite}, \mathcal{R}, top, \searrow \rangle$ *is a* state hierarchy *iff* $\searrow \subseteq \Sigma_{composite} \times \mathcal{R} \cup \mathcal{R} \times \Sigma$, *and* $\langle \Sigma \cup \mathcal{R}, \searrow \rangle$ *is a tree rooted at* $top \in \mathcal{R}$ *such that the set of leaves of the tree is* $\Sigma \setminus \Sigma_{composite}$.

The inverse relation of $\searrow$ is the function $parent : \Sigma \cup \mathcal{R} \setminus \{top\} \to \Sigma_{composite} \cup \mathcal{R}$, giving the parent region of a state or the parent composite state of a non-top region. Thus, the direct substates of a composite state c are the states s such that $parent^2(s) = c$. The set of proper descendants of a state or region v is defined by $descendants(v) = \{v' \in \Sigma \cup \mathcal{R} \mid v \searrow^+ v'\}$, where $\searrow^+$ denotes the irreflexive transitive closure of $\searrow$.

Two states $s_1, s_2 \in \Sigma$ are *orthogonal*, denoted $s_1 \perp s_2$, iff there are regions $r_1, r_2 \in \mathcal{R}$ such that $r_1 \neq r_2$, $parent(r_1) = parent(r_2)$, $s_1 \in descendants(r_1)$, and $s_2 \in descendants(r_2)$. A set $S \subseteq \Sigma$ is *consistent* iff for any two distinct states $s_1, s_2 \in S$ either $s_1 \perp s_2$, $s_1 \in descendants(s_2)$, or $s_2 \in descendants(s_1)$. In particular, the set of active states in a state machine is always a maximal consistent set, i.e. a consistent set that is not a proper subset of any other consistent set.

Consider the state machine diagram shown in Fig. 1. States A and Loop are orthogonal, thus they can be active at the same time. In this situation, the entire set of active states would be $\{\text{Main}, \text{A}, \text{Loop}\}$. Notice that regions are not explicitly drawn in diagrams. Instead, the regions of a composite state are separated by a dashed line, and the top region contains the entire diagram.

A state machine transition is defined as follows. First, we assume the existence of a finite set of signals E, an expression language $\mathcal{L}_g$ for expressing guards, and an action language $\mathcal{L}_a$ for expressing effects. The languages are discussed in Section 3.5.

Definition 2. *A* transition *over a given state hierarchy is a tuple* $t = \langle s, e, g, a, S' \rangle \in (\Sigma \setminus \Sigma_{final}) \times (E \cup \{\tau\}) \times \mathcal{L}_g \times \mathcal{L}_a \times 2^{\Sigma}$ *such that there exists a region* $r \in \mathcal{R}$ *for which* S' *is a maximal consistent subset of* $descendants(r)$ *and* $s \in descendants(r)$, *and if* $s \in \Sigma_{initial} \cup \Sigma_{choice}$ *then* $e = \tau$. *We define* $source(t) = s$, $trigger(t) = e$, $guard(t) = g$, $effect(t) = a$, $targets(t) = S'$, *and* $container(t) = r$.

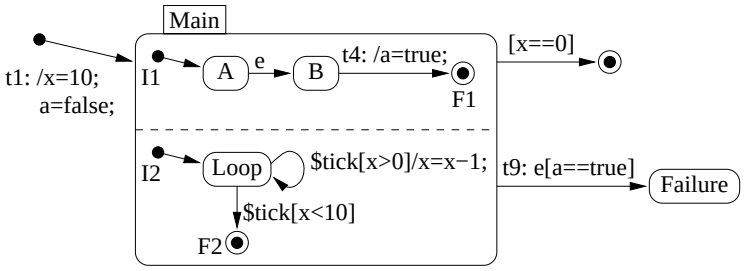


Fig. 1. Running example.

In the graphical UML notation, a transition is shown as an arrow from the source state to the main target state. A transition has a text label *name*: *trigger* [*guard*] /*effect*. Any of the four parts may be omitted.

A transition t has a source state $source(t)$ and a set of target states $targets(t)$. In the simple case, $targets(t)$ is a singleton set, but if the main target is a composite state, then we assume that $targets(t)$ also contains the initial states entered by the transition. For example, $source(\text{t4}) = \text{B}$ in Fig. 1, $targets(\text{t4}) = \{\text{F1}\}$, $source(\text{t9}) = \text{Main}$, and $targets(\text{t1}) = \{\text{Main}, \text{I1}, \text{I2}\}$. The intuition behind $container(t)$ is that it is the smallest region containing all the states exited or entered by the transition.

A transition can be fired only if an occurrence of its triggering event is dispatched. The trigger $trigger(t)$ is a signal name if the transition is triggered by the reception of a signal, or the special symbol τ if the transition is a completion transition. The symbol τ is omitted in diagrams. A transition also has an associated guard $guard(t)$, which defaults to `true` if it is omitted. The guard is a side-effect free Boolean expression giving a precondition for firing the transition. A transition may have an effect $effect(t)$ that is executed upon firing the transition. The default effect is to do nothing.

If an event occurrence is dispatched but it does not cause any transitions to be fired, the event can be *deferred* and dispatched again later. This happens if the event is designated deferrable by one of the active states. Thus, we define a mapping $deferrable$ from states to subsets of the set of signals E.

Definition 3. *A* UML state machine *is a tuple* $\langle H, \Phi, deferrable \rangle$*, where* H *is a state hierarchy,* Φ *is a set of transitions over* H*, and* $deferrable : \Sigma \rightarrow 2^E$*.*

3.3 State Configurations

A completion transition is triggered by an implicit completion event that is generated when the source state finishes all internal activity. For a pseudostate or simple state, this happens (in our UML subset) immediately when the state becomes active. For a composite state, a completion event is generated when all regions have reached their final states.

Instead of maintaining a queue of completion transitions, we associate equivalent information with each active state. We mark an active state *busy* if a completion event for the state has not yet been dispatched. An active state is *quiescent* iff it is not busy, i.e., iff a completion event has been dispatched without firing any completion transitions.

Definition 4. *A* state configuration *over a state hierarchy is a pair* $\langle A, Q \rangle$*, where the set of active states* A *is a maximal consistent subset of* Σ *and* $Q \subseteq A$ *is the set of quiescent states.*

A busy state is *completed* if, conceptually, a completion event for the state has been generated but not yet dispatched. Thus, a busy state is completed iff it is a non-composite state, or a composite state whose active substates are all quiescent final states.

Definition 5. *The set of* completed *states in a state configuration* $C = \langle A, Q \rangle$ *is*

$$completed(C) = \{s \in A \setminus Q \mid descendants(s) \cap A \subseteq \Sigma_{final} \cap Q\}.$$

When a composite state c becomes active during execution, it is first busy and not completed. If all regions of the state reach their final states, then c becomes busy and completed and, conceptually, a completion event is generated. After that, it is possible to either (i) fire a completion transition whose source is c and whose guard is `true`, or, if there is no such transition, (ii) consume the completion event and make c quiescent. If c has become quiescent, no completion transitions from c will be fired because the completion event has already been consumed. This behavior fulfills the requirement of UML that the guards of completion transitions leaving a state are evaluated only once after the state has become completed.

If c above is replaced by a non-composite state s, the behavior is similar except that the phase of s being busy and not completed is skipped.

The initial state configuration of a state machine is $\langle \{s\}, \{\} \rangle$, where $s \in \Sigma_{initial}$ and $parent(s) = top$. The model is ill-formed unless there is exactly one such s.

3.4 Transition Firing Dynamics

In general, a UML state machine instance moves from a state configuration to another by firing a maximal conflict-free set of enabled transitions. However, we have chosen to make the restriction, as discussed in Sect. 3.1, that orthogonal regions may not contain transitions triggered with the same signal. Formally, if $t_1, t_2 \in \Phi$ and $trigger(t_1) = trigger(t_2) \neq \tau$, then $\neg\ (source(t_1) \perp source(t_2))$. It follows that the maximal conflict-free set contains at most one transition.

A transition t is *enabled* by an event occurrence iff the event matches the trigger of t, the source state of t is active, and the guard of t evaluates to `true`.

If an enabled transition t is fired, the resulting state configuration is obtained by first removing $source(t)$ and any other descendant states of $container(t)$ from the sets of active states and quiescent states, and then adding $targets(t)$ to active states. Nothing is added to the set of quiescent states because all new active states are busy.

Definition 6. *Let $C = \langle A, Q \rangle$ be a state configuration and let $t \in \Phi$ be a transition such that $source(t) \in A$. The* next state configuration *associated with t and C is*

$$nextstateconf(t, C) = \langle (A \setminus S) \cup targets(t), Q \setminus S \rangle,$$

where $S = descendants(container(t))$.

A completion transition is enabled iff its source state is active, busy, and completed, and the guard condition is true. If there is an active, busy, and completed state s that is not the source state of any enabled completion transition, we say that a *quiescing step* QUIESCE(s) is enabled. The only effect of firing the quiescing step is to make s quiescent, which corresponds to implicit consumption of the completion event for s. If there are any enabled completion transitions or quiescing steps, one of them is chosen nondeterministically for firing and (according to UML) only if there are none, signal-triggered transitions are considered.

Given a state configuration and an event occurrence, it is possible that several transitions are enabled. However, some of these are ruled out based on UML semantics, which states that transitions deeper in the hierarchy have priority. Priority also applies to deferral of events, but not to completion transitions because no busy, completed state can be a descendant of another one.

Definition 7. *Let T be the set of transitions enabled by an event e in a state configuration $\langle A, Q \rangle$ and let $S = \{source(t') \mid t' \in T\} \cup \{s \in A \mid e \in deferrable(s)\}$. The set of* prioritized *transitions is*

$$prioritized(T, A) = \{t \in T \mid descendants(source(t)) \cap S = \varnothing\}.$$

Because A is a consistent set of active states and the restriction on triggers in orthogonal regions holds, every $t \in prioritized(T, A)$ has in fact the same source state. One transition in the set is chosen nondeterministically for firing.

3.5 Action Language

In our subset of UML state machines, an action language is used in two roles, namely to specify the guard constraints and the effects of transitions.

The choice of an action language is connected to the level of support for various UML model elements. The minimal level is to allow sending signals to objects. Our action language supports more than this, e.g. attributes of objects and dynamic creation of new objects. The action language supported by our

PROMELA translation is a subset of the Jumbala action language [10]. Jumbala is an object-oriented language that could be characterized as simplified Java tailored to the UML framework. The language is strongly typed with `int` (32-bit signed integer) and `boolean` primitive types and object reference types.

In state machines, the effects of transitions are lists of Jumbala statements, and the transition guards are `boolean` expressions. Below is a list of the kinds of statements supported by the PROMELA translation. The syntax and semantics follow the conventions of the Java programming language, with an added `send` statement.

- Assignments of the form '*lhs* `=` *rhs*`;`'.
- If statements of the form '`if (`*condition*`) {` *truestmt* `} else {` *falsestmt* `}`', where *condition* is a `boolean` expression. The `else` part may be omitted.
- Iteration statements of the form '`while (`*condition*`) {` *stmt* `}`'.
- Send statements of the form '`send` *signalname*`(`*paramvalues*`) to` *object*`;`'. A send statement places a signal event *signalname* in the input queue of *object*. Values for signal parameters are given as a comma-separated list.
- Assertions of the form '`assert` *condition*`;`'.

The following kinds of Jumbala expressions are supported. Below we assume that *obj* is the object in whose context the guard or action is evaluated.

- 32-bit decimal integer literals.
- The `boolean` literals `true` and `false`.
- The expression `this`, which is a reference to *obj*.
- Names of the form *identifier* or *identifier.identifier*. A name can resolve to either an object reachable from *obj* by following links (association instances), or an attribute.
- Infix expressions of the form *leftexpr op rightexpr*. The binary operator *op* can be one of `+`, `-`, `*`, `/`, `%`, `&`, `^`, `|`, `>`, `<`, `>=`, `<=`, `==`, `!=`, `<<`, or `>>`. The semantics of operators is the same as in PROMELA.
- Instance creation expressions of the form `new` *classname*`()`. The state machine of the newly created object begins executing automatically.

3.6 Execution of Models

At a given moment in time, the system is in a state that, as a whole, conforms to the UML model. We call this state the *global configuration* of the system. A global configuration consists of a set of objects, where each object *obj* contains the following information.

1. The values of the instance attributes of *obj*.
2. The links that are navigable from *obj*, pointing to other objects.
3. The state configuration of the state machine of *obj*, denoted $obj.stateconf$.
4. The input queue of *obj*, which we denote by $obj.inputqueue$.

while true:
  pick an object *obj*
  $\langle A, Q \rangle := obj.stateconf$
  $sources := completed(obj.stateconf)$
  $compl := \{t \in \Phi \mid source(t) \in sources \wedge trigger(t) = \tau \wedge evalguard(obj, t, \langle\rangle)\}$
  $enabled := compl \cup \{\text{QUIESCE}(s) \mid s \in sources \wedge \nexists t \in compl$ such that $s = source(t)\}$
  if $enabled \neq \varnothing$:
    if $enabled \cap \Phi_{pseudostate} \neq \varnothing$:
      $enabled := enabled \cap \Phi_{pseudostate}$
    pick $t \in enabled$
    if $t = \text{QUIESCE}(s)$ for some s:
      $obj.stateconf := \langle A, Q \cup \{s\} \rangle$
    else:
      execute $effect(t)$ in the context *obj*
      $obj.stateconf := nextstateconf(t, obj.stateconf)$
  else if *obj.inputqueue* is not empty:
    remove the first element $\langle e, params \rangle$ from *obj.inputqueue*
    $enabled := \{t \in \Phi \mid source(t) \in A \wedge trigger(t) = e \wedge evalguard(obj, t, params)\}$
    if $prioritized(enabled, A) \neq \varnothing$:
      pick $t \in prioritized(enabled, A)$
      $assign(obj, t, params)$
      execute $effect(t)$ in the context *obj*
      $obj.stateconf := nextstateconf(t, obj.stateconf)$
      push $obj.deferredqueue$ in front of *obj.inputqueue*
      $obj.deferredqueue :=$ empty
    else if $\exists s \in A$ such that $e \in deferrable(s)$:
      append $\langle e, params \rangle$ to $obj.deferredqueue$

Fig. 2. Execution Algorithm for UML Models

5. The deferred queue of *obj*, denoted $obj.deferredqueue$.

The last two elements are FIFO queues whose elements are signal event occurrences represented as pairs $\langle e, params \rangle$, where $e \in E$ is a signal name and *params* is a tuple of signal parameter values.

The algorithm in Fig. 2 illustrates the execution of a model. In one step of execution, one object is nondeterministically chosen. If any completion transitions or quiescing steps are enabled, then one of them is fired. Otherwise, an event occurrence is removed from the input queue and a prioritized enabled transition is fired. If no such transition exists and the event is not deferrable, the event occurrence is implicitly consumed. In transition selection, transitions whose source state is an initial or choice pseudostate (the set $\Phi_{pseudostate}$) are preferred to other completion transitions.

In order to handle signal parameters, the algorithm uses the following auxiliary procedures.

- $assign(obj, t, params)$ modifies the global configuration by assigning the values in the tuple $params$ to the instance attributes of obj. The attributes receiving the new values are named in the trigger of the transition t.
- $evalguard(obj, t, params)$ evaluates $guard(t)$ in the context of object obj. The values of $params$ are assigned as if $assign(obj, t, params)$ had been executed, for the duration of guard evaluation. After evaluation, the original attributes are restored and a truth value is returned.

The execution algorithm is such that in one step a single transition in any object is fired, and in the next step a transition in another object might be fired. We call this *transition segment granularity*. Alternatively, it would be relatively straightforward to modify the algorithm to use *compound transition granularity*, so that one step of execution would correspond to a compound transition, i.e. a sequence of transition segments with only pseudostates between them. A third possibility would be *run-to-completion granularity*, where the firing of a signal-triggered transition in an object is followed by as many completion transitions or quiescing steps as possible in the same object before executing transitions in other objects.

4 Translation to PROMELA

This section presents our main contribution, the translation to PROMELA. It translates the active classes and their state machines to corresponding PROMELA processes, and uses the deployment diagram to infer the initially active objects and how their associations to other active objects are set. The resulting PROMELA program can be checked for deadlocks or assertion violations in the model. If an error is found, the error trace can be simulated in the UML model by using a separate model simulator.

Our translation requires the user to supply information about the model that is otherwise hard to infer: (i) the size of the input and deferred queues (QSIZE), and (ii) the maximum number of instances of a each class (MAXIDS). It is relatively easy to augment the PROMELA translation to check whether these limits are exceeded, and give an error requesting the user to increase them if this happens.

4.1 SPIN and PROMELA: Brief Introduction

PROMELA (PROcess MEta LAnguage) is the input language of the tool SPIN [1] initially developed in Bell Labs by Gerard Holzmann et al. The language allows for the dynamic creation of processes and both synchronous (rendezvous) and asynchronous communication through communication channels.

The PROMELA language is rather rich, however our translation does not need most of the features. The elements that we use are briefly presented. The queue (asynchronous channel) operations are as follows. The send command

q!v1,...,vn appends the queue q with the message comprised of the values v1,...,vn. Similarly, the receive command q?v1,...,vn reads the first message from the queue (or blocks if the queue is empty). A central part of our translation is the if... fi compound statement that can for instance be as follows:

```
if
  :: (a == 1) -> a = a + 1;
  :: (a == 0) -> a = 1;
  :: else -> a = 0;
fi
```

The statement above defines a simple selection construct with three option sequences, each starting with a double-colon. Each option sequence starts with a guard (in our example, the first guard is (a == 1)). This guard must evaluate to true so that this option sequence can be executed. If several guards evaluate to true, then one of them is chosen nondeterministically and the associated commands are executed. The else sequence (if present) is chosen iff the guards of every other sequence evaluate to false.

Normally, the scheduled object in SPIN can change after each command. However, if this is not desired, it can be prevented by enclosing several commands inside an atomic block (with the keyword atomic). We also use this feature.

4.2 Global Variables and Initialization Block

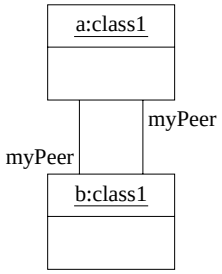
The global variables of the PROMELA specification are the input queues and deferred queues for each class. These are arrays with MAXIDS slots. The associations for each class are also stored in a similar table. Finally, for each class, there is an integer that stores the process number of the last created instance of that class.

The initialization of the active processes (the structure of the PROMELA init block) is as follows. Each active object (declared in the deployment diagram) is processed in turn. First, its associations are configured by setting the values in the global association table and then the object is started with the command run with its process id as the argument. In order to maintain scheduling in the init-process, these commands are enclosed inside an atomic block. For instance, the left-hand side of Fig. 3 shows a deployment diagram describing an initial configuration of a model. It has two objects, a and b, that are instances of the active class "class1" and the initializations of the association "myPeer" from "class1" to itself. The right hand-side of Fig. 3 gives the PROMELA translation of the diagram.

4.3 Translation of State Machines

Figure 5 gives a skeletal translation of the state machine in Fig. 1.

Each class is translated to a PROMELA process (a proctype declaration, like line 4 in Fig. 5). This process has one argument, the instance number of




```
#define MAXIDS 2
int class1_myPeer[MAXIDS];
int class1_pid = 0;
init {
  atomic {
    class1_myPeer[0] = 1;
    run class1(0); /* Start active object a */
    class1_pid = class1_pid + 1;
    class1_myPeer[1] = 0;
    run class1(1); /* Start active object b */
    class1_pid = class1_pid + 1;
  }
}
```

Fig. 3. An initial configuration and its PROMELA translation.

the created object. The first instance gets the number zero and the maximum number of instances is user specified. The instance attributes of the class and integer variables encoding the state configuration of each region in state machine of the class are declared first.

The main loop encoding the state machine is divided into two parts, evalcompletions and evaltriggers for completion and signal-triggered transitions, respectively (lines 11 and 34 in Fig. 5). This division is due to the fact that according to UML semantics, completion transitions have priority over signal-triggered transitions. Therefore the PROMELA code follows the following idea. First, fire completion transitions as long as possible. Then, consume a signal event from the event queue (or wait until the queue becomes non-empty), fire a signal-triggered transition, and go back to trying to fire completion transitions.

In order to handle completion transitions correctly without actually generating completion events and having a queue (with priority over the normal signal event queue) for them, we use the concept of busy and quiescent states introduced in Sect. 3.3. That is, for each simple and composite state that has outgoing completion transitions, we have two possible values in the PROMELA state vector (e.g. s_Top_Main_busy and s_Top_Main in the code in Fig. 5). Completion transitions are only evaluated if the state is marked busy. As pseudostates only have outgoing completion transitions, there is no need for such additional information for them.

The general structure for the block evaluating whether a completion transition can be fired is fairly simple as there is no need to take the transition priority caused by the hierarchy into account. In order to first fire completion transitions leaving from a pseudostate, the block consists of two consecutive, similar sub-blocks: the first only considers pseudostates (lines 12–21 in Fig. 5) while the second (lines 22–33) takes care of the simple and composite states. Both sub-blocks are just large if...fi blocks with one option sequence for each (pseudo)state with outgoing completion transitions. Each option sequence first checks whether the (pseudo)state in question is active (and busy and completed if it is a simple or composite state), and then non-deterministically chooses a completion transition whose guard is true. If there was no such completion tran-

sition, then (i) a quiescing step is taken if the state was simple or composite, or (ii) following UML specifications, an error is reported if the state was a pseudostate.

The code for signal-triggered transitions (lines 35–57) starts with a consumption of a signal event from the queue, or non-deterministic generation of an external signal if the state configuration is in a state that can consume an external signal event. [5] The general structure for evaluating whether a signal-triggered transition can be fired is a large if...fi block (lines 40–57) with one option sequence for each state in the top level state machine. The guards for the option sequences are simply checks of whether a particular state is active. The option sequence then depends on whether the state is simple or composite. If the state s is composite, a nested if...fi block follows (for example lines 42–50 in Fig. 5), this time containing an option sequence for all the children of s. After this (possibly empty) block, a subsequent if...fi block evaluates for each outgoing signal-triggered transition whether (i) the trigger matches the consumed signal event, and (ii) the guard evaluates to true.

In both blocks described above, the code for firing a transition is similar. The encoding of the effect of the transition (a sequence of Jumbala statements) is presented in detail in Sect. 4.4. The translation of each transition is finished with PROMELA code that sets the new active state to be the target state of the transition. For composite states, we also have to set its regions to their initial states. If the target state has outgoing completion transitions, then the state is marked busy. After this, the control flow is transferred to the beginning of the block evaluating completion transitions.

4.4 Translation of Action Language

The supported constructs are presented in Section 3.5. Now, we describe their translation to PROMELA. The translation of simple assignments is simple, both variable declarations and assignments are syntactically similar in PROMELA and Jumbala.

Queue operations are used to send messages to active associations of an object or to read messages from its input queue. In order to manage the dynamic creation of new instances we store the associations of objects as well as their input queues in a global array. Each entry in an array of a particular association is an instance number. Thus sending a message along an association requires accessing this array with the instance number of the process to obtain the target object. The message is then sent to the input queue of this object.

New objects are created as follows. There is a global variable that is one greater than the instance number of the last created instance. This number is used to set the associations of the new object (entries in the global arrays). Then

[5] An *external* signal is a signal with no parameters and whose name starts with a $-sign. It can be non-deterministically generated by the environment of the UML model. Such signals are convenient when modeling open or underspecified systems.

the appropriate proctype is called with the instance number. Finally, this number is incremented.

We also support the if... else if structure of Jumbala by simulating this with the PROMELA if... fi structure. The constructs are different in that in Jumbala, the executed branch is the first one where the guard evaluates to true. In PROMELA, any branch whose guard evaluates to true can be chosen. However, this is easy to simulate by introducing Boolean flags that guarantee that if a particular guard evaluates to true, then all the preceding guards evaluate to false. The while statement is translated to a PROMELA do... od structure. This translation is straightforward.

Finally, it should be noted that the code in Fig. 5 does not accurately model scheduling of objects. Indeed, if this code were a part of a larger concurrent system, the system would have too many behaviors since SPIN could change the scheduled object in the middle of the execution of a UML transition. This omission is intentional due to lack of space and the fact that UML semantics does not define scheduling policy (see discussion at the end of Sect. 3.6). One can for instance allow an object to fire only a single transition or fire completion transitions until a stable configuration is reached. Both of these scheduling policies (and others) can be implemented using the PROMELA atomic statement.

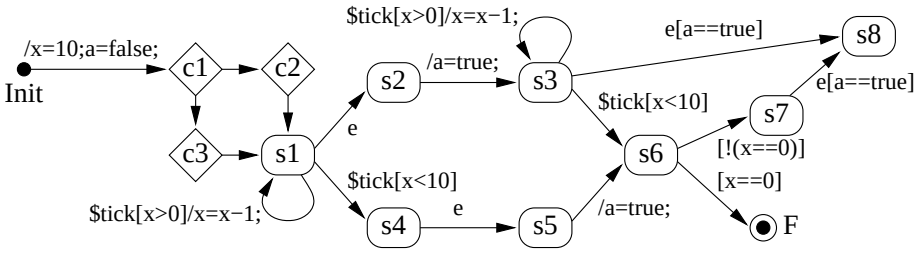


Fig. 4. The flattened version of the state machine in Fig. 1.

4.5 Flat State Machines

An interesting idea is to consider the case where UML state machines are flattened. Intuitively this means that hierarchical states are replaced with several simple states so that the behavior of the system is the same, Fig. 4 shows the flattened version of the state machine in Fig. 1. For flat state machines, we propose a translation where the state vector of SPIN is made shorter by removing the variables storing the component states. Instead, the PROMELA code has a label for each state of the flattened machine which is followed by an if ... fi block for its outgoing transitions. If a transition is fired, control flow is set to the transition's target state by using a goto statement to the corresponding label. Whether or not this added simplicity compensates the potential blowup in the state machine and PROMELA code sizes is a question we have yet to answer.

5 Evaluation

We have implemented our translation in a tool called PROCO [6]. Its input parameters are the UML (version 1.4) model in the XMI format supported by the Coral tool [11], the maximum number of active instances of a class, and the size of the input and deferred queues. The output is a PROMELA model.

We have tested the tool with several simple models. One of them models a protocol consisting of an environment and two protocol entities, a sender and a receiver. The environment initiates a session, after which the protocol entities shake hands. After the handshake, the protocol is running and the sender forwards data signals from the environment to the receiver. Our initial model contains a deadlock: the first data signal may reach the sender before the handshaking is complete, and the data is lost. This can be fixed by simply deferring the data signal in the state where the sender is waiting for the handshake. PROCO is able to detect the deadlock and it is possible to simulate the corresponding trace.

We have also applied PROCO to a simplified model received from an industrial partner. The model portrays a client-server architecture at an abstract level, using 4 active classes and a total of 33 state machine states. PROCO and SPIN find a deadlock in the model, or prove the absence of a deadlock if the model is modified, in fractions of a second, regardless of whether the model is flattened or not.

To better assess the scalability of the approach, we need to obtain larger models in XMI format and run experiments with them. We expect some of the big challenges to be the handling of data and polymorphism. Our tool does not currently support generalization of classes, and there are no arrays or passive classes representing data structures. Another possible issue is the efficient handling of advanced state machine concepts such as history states.

6 Conclusions

This paper outlines an approach to model check UML state machines. Although using SPIN as a back-end model checker has been tried before, our work differs from previous work in that it focuses on a UML subset for protocol models. We also support more action language features than some other previous work.

In the near future we plan to conduct case studies to evaluate our approach. We are especially interested in evaluating whether flattening of state machines can help analysis, and how PROMELA translations should be designed to increase the efficiency of partial order reductions. We also wish to investigate whether the subclassing mechanism of our action language can be used to support data abstraction.

Acknowledgements. This work has been financially supported by Tekes, Nokia, Conformiq, Mipro, the Academy of Finland, and the Emil Aaltonen Foundation.

[6] available at http://www.tcs.hut.fi/SMUML/

```
/* constant definitions for all states, signals etc. */
#define s_Top_Init 0
...
proctype M(int proc_id) {
5:      byte state_Top = s_Top_Init;
        byte state_Top_Main_R1 = s_Top_Main_R1_None;
        byte state_Top_Main_R2 = s_Top_Main_R2_None;
        byte x; bool a; /* Instance attributes of the owning class */
        byte trigger, p1, ...; /* for signal type & parameters */
10:     xr inputqueues[proc_id];

        evalcompletions:
        if /* Try to fire completion transitions from pseudostates */
        :: (state_Top == s_Top_Init) ->
          x = 10; a = false; state_Top = s_Top_Main_busy; state_Top_Main_R1 = s_Top_Main_R1_Init;
15:       state_Top_Main_R2 = s_Top_Main_R2_Init; goto evalcompletions;
        :: (state_Top_Main_R1 == s_Top_Main_R1_Init) ->
          state_Top_Main_R1 = s_Top_Main_R1_A; goto evalcompletions;
        :: (state_Top_Main_R2 == s_Top_Main_R2_Init) ->
          state_Top_Main_R2 = s_Top_Main_R2_Loop; goto evalcompletions;
20:     :: else -> skip;
        fi
        if /* Try to fire completion transitions from real states */
        :: (state_Top_Main_R1 == s_Top_Main_R1_B_busy) ->
          a = true; state_Top_Main_R1 = s_Top_Main_R1_Final; goto evalcompletions;
25:     :: (state_Top = s_Top_Main_busy && state_Top_Main_R1 == s_Top_Main_R1_final &&
          state_Top_Main_R2 == s_Top_Main_R2_final) ->
          if
          :: (x == 0) -> state_Top = s_Top_Final; state_Top_Main_R1 == s_Top_Main_R1_None;
            state_Top_Main_R2 == s_Top_Main_R2_None; goto evalcompletions;
30:       :: else -> state_Top = s_Top_Main; goto evalcompletions; /* Quiescing step */
          fi
        :: else -> skip; /* No completion transition was enabled */
        fi
        evaltriggers:
35:     if
        :: inputqueues[proc_id]?trigger,p1; /* Consume signal event (if any) */
        /* Non-deterministically create an external signal if in a state that can consume it */
        :: (state_Top_Main_R1 == s_Top_Main_R1_Loop) -> trigger = signal_$tick;
        fi
40:     if
        :: (state_Top == s_Top_Main_busy || state_Top == s_Top_Main) ->
          if
          :: (state_Top_Main_R1 == s_Top_Main_R1_A && trigger == signal_e && true) ->
            state_Top_Main_R1 = s_Top_Main_R1_B_busy; goto evalcompletions;
45:       :: (state_Top_Main_R2 == s_Top_Main_R2_Loop && trigger == signal_$tick && x > 0) ->
            x = x - 1; goto evalcompletions;
          :: (state_Top_Main_R2 == s_Top_Main_R2_Loop && trigger == signal_$tick && x < 10) ->
            state_Top_Main_R2 == s_Top_Main_R2_Final -> goto evalcompletions;
          :: else -> skip
50:       fi
          if /* Signal was not consumed by any substate of Main */
          :: (trigger == signal_e && a == true) -> state_Top_Main_R1 = s_Top_Main_R1_None;
            state_Top_Main_R2 = s_Top_Main_R2_None; state_Top = s_Top_Failure; goto evalcompletions;
          :: else -> skip
55:       fi
        :: else -> skip;
        fi
        goto evaltriggers; /* implicit consumption occurred */
}
```

Fig. 5. PROMELA code of running example.

References

1. Holzmann, G.J.: The Spin Model Checker. Addison Wesley (2004)
2. Latella, D., Majzik, I., Massink, M.: Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. Formal Aspects of Computing **11** (1999) 637–664
3. Mikk, E.: Semantics and Verification of Statecharts. PhD thesis, Christian-Albrechts-Universität (2000)
4. Porres, I.: Modeling and Analyzing Software Behavior in UML. PhD thesis, Åbo Akademi (2001)
5. Knapp, A., Merz, S.: Model checking and code generation for UML state machines and collaborations. In: 5th Workshop on Tools for System Design and Verification. Report 2002-11, Reisensburg, Germany, Institut für Informatik, Universität Augsburg (2002)
6. Gargantini, A., Riccobene, E., Rinzivillo, S.: Using SPIN to generate tests from ASM specifications. In: Abstract State Machines. Number 2589 in LNCS, Springer-Verlag (2003)
7. Ober, I., Graf, S., Ober, I.: Validation of UML models via a mapping to communicating extended timed automata. In: 11th International SPIN Workshop on Model Checking of Software, 2004. Volume LNCS 2989. (2004)
8. Schinz, I., Toben, T., Mrugalla, C., Westphal, B.: The Rhapsody UML verification environment. In: SEFM, IEEE Computer Society (2004) 174–183
9. Telelogic: Telelogic Tau G2 v2.7 (2006) Software. http://www.telelogic.com/.
10. Dubrovin, J.: Jumbala — an action language for UML state machines. Research Report A101, Helsinki University of Technology, Lab. for Theoretical Computer Science (2006)
11. Alanen, M., Porres, I.: Coral: A metamodel kernel for transformation engines. In: Proc. Second European Workshop on Model Driven Architecture (MDA). Number 17-04 in Tech. Report, Computing Laboratory, Univ. of Kent (2004) 165–170

Static Verification of UML Model Consistency

Andrea Baruzzo[1] and Marco Comini[1]

Dipartimento di Matematica e Informatica (DIMI), University of Udine,
Via delle Scienze 206, 33100 Udine, Italy.

Abstract. In a UML model, different aspects of a system are covered by different types of diagrams and this bears the risk that an overall system specification becomes inconsistent or incomplete. Hence, it is important to provide means to check the consistency and completeness of a UML model.
Many approaches for model validation and verification rely on generation of suitable code which dynamically (i.e., at run-time) checks the validity of OCL constraints. This approach has several well-known drawbacks. For example, it cannot generally guarantee that a constraint will never be violated, unless an infinite number of tests is performed. On the other hand, even more desirable static approaches are not immune from weaknesses. The techniques based on model checking suffer from the state explosion problem and thus cannot scale to most system sizes. Moreover, the static approach is in general undecidable.
In this paper we propose a static verification framework based of Abstract Interpretation techniques, a theory of approximation of mathematical structures. This framework, by using OCL constraints together with Class Diagrams, certifies that the dynamic part (Sequence Diagrams) of the model is satisfied. Our approach keeps the advantages of static verification and, at the same time, avoids the weaknesses of the other mentioned methods as it does not require users to build test scenarios and also it can make undecidable methods decidable, up to a specified level of precision.

1 Introduction

Finding program bugs is a long-standing problem in software construction. There has been considerable theoretical research activities and published results starting from the mid-nineties about using formal specifications to help the debugging phase. All these theoretical efforts have produced also relevant practical results. Indeed, the software engineering community has nowadays accepted the fact that the specification of various kinds of pieces of software is not only a topic of theoretical interest but also one of practical importance. A steadily increasing number of papers dealing with the concepts and the practical use of assertions in general have been published during the last few years (amongst all [1]). The basic foundations have been laid by Bertrand Meyer with his concept of *Design by Contract* (DbC) as realized in the Eiffel language (see [2, 3]). This approach

has then rapidly spread to other languages, for instance there emerged lot of support for assertions for Java (e.g. Jass[1], the iContract tool[2], *etc.*) and C++ (e.g., C^2 [3], *etc.*). Most importantly, the Unified Modeling Language (UML) has now, as one of its integral parts, the Object Constraint Language (OCL), which has its root in the Syntropy method. With OCL we can naturally implement the contract mechanism of Eiffel.

In a UML model, different aspects of a system are covered by different types of diagrams and this bears the risk that an overall system specification becomes inconsistent or incomplete. Hence, it is important to provide means to check the consistency and completeness of a UML model.

Many approaches for system debugging and for model validation/verification rely on generation of suitable code which dynamically (i.e., at run-time) checks the validity of OCL constraints (i.e., the compliance of the system status w.r.t. the constraint). This approach has several drawbacks. For example, it undoubtedly slows down performance and can potentially alter the behavior (if the inserted code has side effects by mistake). But most of all it does not ensure to reveal a bug unless the specific run of the system effectively enters a state which is not compliant w.r.t. the specification. One can argue that not all runs are actually needed to manifest an error, since most symptoms (wrong traces) are caused by the same error. However also the generation of just a *significative* finite subset of the possible runs is not so feasible because, on one hand, a considerable manual effort is needed even to produce a single test scenario and, on the other hand, test-case generation is well-known to be a hard problem.

On the contrary *static* (semantics-based) tools could guarantee that *any* run will be compliant w.r.t. the specification, without even adding extra overhead. The problem with this approach is that it is in general (well-known to be) undecidable and, in any case, much more difficult to tackle.

Many researchers are proposing static approaches based on Model Checking, but this suffers of the state explosion problem and thus (while suitable for protocols and small hardware systems) cannot scale to most software system sizes, typical of (commercial) software production. Moreover there is also an inherent limit to verification of a single specific property of the system at a time.

This paper is motivated by the fact that we believe we can attack the undecidability of the static approach by using Abstract Interpretation techniques [4–9]. Abstract Interpretation is a theory of approximation of mathematical structures, in particular those involved in the semantic models of computer systems. Abstract interpretation can be applied to the systematic construction of methods and effective algorithms to approximate undecidable or very complex problems in computer science such as the semantics, the proof, the static analysis, the verification, the safety and the security of software or hardware computer systems. In particular, abstract interpretation-based static analysis, which automatically

[1] See http://semantik.informatik.uni-oldenburg.de/~jass/.
[2] See http://icontract2.org/.
[3] See http://www.aechmea.de/html/german/Information01_e.htm.

infers dynamic properties of computer systems, has been very successful these last years to automatically verify complex properties of real-time, safety critical, embedded systems.

We already had plenty of experience in Debugging and Verification of Declarative Languages where, by using Abstract Interpretation techniques, we could develop effective semantic-based tools [10–15]. The nice feature of this approach is that it can discover bugs even in absence of symptoms. Moreover it does not need a complete system to work, since we can (must) use, in place of missing components, their specification to diagnose existing parts.

This could be the case also for most systems providing UML diagrams with OCL specifications. However, given the level of complexity of such systems, it can easily be the case that the UML diagram *in itself* is not consistent. This would render the use of (complex), either static or dynamic, code diagnosis tools completely pointless. Hence it is important to have a tool to statically check the consistency of an UML model to achieve a good design *even before* the implementation starts. It can help further debugging stages and it is important in itself for Model Validation. This is even more important in Model Driven Architecture (MDA) approaches where new diagrams and code are automatically synthesized from the initial model: all the constructed artifacts would inherit the initial inconsistency. These considerations lead us to propose the conceptual framework described on the following.

The paper is structured as follows: in section 2 we introduce some concepts about assertions in UML with OCL. In section 3 we present our Conceptual Framework with an example to show how it works. Then in section 4 we discuss about its applicability for development of software verification tools.

2 Assertions in the Software Engineering Practice

2.1 Design by Contract

In this section we will look how some of the concepts introduced above can be transferred to software systems in practice. Design by Contract (DbC) [16] is inspired by formal approaches embodied in specification languages such as Z and VDM. Bertrand Meyer has coined the concept of DbC to denote a software development style which (1) emphasizes the importance of formal specifications, (2) interleaves them with actual code, and (3) makes these contracts executable. DbC is a systematic method of assertion usage and interpretation introduced as a standard feature of the Eiffel language [2]. Without it, no trial would have ever been made to provide a similar mechanism in other languages and, by no means, would we have discussion papers like this and the ones mentioned in the references.

Software contracts have been invented to capture mutual obligations and benefits among classes, as they are e.g. needed in design patterns, where each of the involved classes is expected to exhibit a “proper” behavior [17, 18]. A

software contract is the specification of the behavior of a class and its associated methods. The contract outlines the responsibilities of both the caller and the method being called. Failure to meet any of the responsibilities stated in the contract results in a break of the contract itself, and indicates the existence of a bug somewhere in the design, in the implementation, in both of them, or - one must not forget this possibility in earlier project phases - in the assertions themselves. Software contracts can be completely specified through the use of preconditions, postconditions, and class invariants in object-oriented software. DbC views software construction as based on contracts between clients (callers) and suppliers (routines). Each party expects some benefits from the contract, and accepts some obligations in return. As in human affairs, the contract document spells out these mutual benefits and obligations and protects both the client, by specifying how much should be done, and the supplier, by stating that the supplier is not liable for failing to carry out tasks outside of the specified scope. The DbC paradigm is as follow:

> The client's obligation is to call a method only in a program state where both the class invariant and the method's precondition hold. The method, in return, guarantees that the work specified in the postcondition has been done, and the class invariant is still respected.

A precondition violation is a manifestation of an error in the client, while a postcondition failure is a manifestation of a bug in the (implementation of the) supplier, which does not fulfill its promise (Note: The phrase "An assertion fails" in real life means just the opposite: the assertion did its job well, because it has found a bug). For this reason, in order to call a method, the client should verify only its preconditions. If the preconditions are satisfied, it should take for grant the postcondition after the termination of the method execution. The supplier, vice versa, should check the postconditions in order to guarantee its part of the contract, but under no circumstances shall the body of the method ever test for its preconditions. Under the *Non Redundancy Principle* [16], hence, the DbC encourage the developer to "check less and get more". DbC is, in this respect, the opposite of defensive programming, which recommends to protect every software module by as many checks as possible. This may result in redundancy and makes it also difficult to precisely assign responsibilities among modules.

2.2 UML and OCL

In the last few years, much effort has been spent to make the UML language more precise. Since its beginning, UML was conceived as a standard graphical language suitable to support the development of object-oriented systems. A clear intent in the UML design was the unification of the previous modeling languages, which all provided different notations for the same concepts. The standardization process was made by the Object Management Group (OMG), involving both the industry and the academia worlds. The results of this process was a relatively stable language, with an informal semantics. This level of definition was sufficient

for sketching analysis and design models. However, when the model needed to be elaborated by automated tools for validation and verification purposes, the lack of a more formal foundation was immediately recognized. Because UML focused primarily on the diagrammatic elements and gave meaning to those elements through English text, a constraint language was added to the specification, in order to provide a more precise definition of the UML meta-model. That language was the Object Constraint Language (OCL) [19], initially developed in 1995 at IBM. OCL allows the integration of both well-formedness rules and assertions (i.e., preconditions, postconditions, invariants) in UML models. The former are useful to validate especially the syntax of a UML model, whereas the latter can be exploited to verify the conceptual constraints.

Preconditions and postconditions provide a mechanism to specify the properties required before and after the execution of an operation, respectively. They do not specify how that operation internally works. The recent development of version 2 for both OCL [20] and UML [21] is a breakthrough in order to completely define the semantics of a method in an object-oriented system. In these last versions, it is possible to define a behavior specification in OCL for any query operation (an operation without side-effects).

Following [22], now we summarize the relevant concepts about UML diagrams and the OCL specification language. For the sake of simplicity, here we present just a summary of the most important results.

In this work we use OCL as specification language to define software contracts such as method preconditions and postconditions, class invariants, and assertions in general. Hence we now define an object model $\mathcal{M}$ that contains the UML elements relevant for this task. Because preconditions, postconditions and invariants are defined typically for class diagram elements (i.e., class attributes and methods), we consider for the moment only the static structure of a UML model. A (static) object model $\mathcal{M}$ can be represented by the following tuple:

$$\mathcal{M} = \langle CLASS, ATT, OP, ASSOC, \preceq, associates, roles, multiplicities \rangle$$

where $CLASS$ is a set of UML classes, ATT is a set of attributes, OP is a set of operations, ASSOC is a set of associations, $\preceq$ is a generalization hierarchy over classes, and *associations*, *roles*, and *multiplicities* are functions that give for each $as \in ASSOC$ its dedicated classes, classes' role names, and multiplicities, respectively (see [23] for complete definitions).

For an object model $\mathcal{M}$ providing a set of types $T_{\mathcal{M}}$, a relation $\leq$ on types reflecting the type hierarchy, and a set of operations $\Omega_{\mathcal{M}}$, the definition of OCL expressions is based upon the signature:

$$\Sigma_{\mathcal{M}} = (T_{\mathcal{M}}, \leq, \Omega_{\mathcal{M}})$$

According to [22], by using this signature we can define the OCL expressions syntax in the following way. Let $\mathtt{Var} = \{\mathtt{Var}_t\}_{t \in T_M}$ be a family of variable sets where each variable set is indexed by a type t. An expression over the signature $\mathtt{Expr}_{\mathcal{M}}$ is given by a set $\mathtt{Expr} = \{\mathtt{Expr}_t\}_{t \in T_M}$ and a function $\mathtt{free} : \mathtt{Expr} \rightarrow \mathcal{F}(\mathtt{Var})$ defined as follow.

– If $v \in \texttt{Var}_t$ then $v \in \texttt{Expr}_t$ and $\texttt{free}(v) := \{v\}$.
– If $v \in \texttt{Var}_{t_1}, e_1 \in \texttt{Expr}_{t_1}, e_2 \in \texttt{Expr}_{t_2}$ then $\texttt{let} \gtrsim = \mathbin{\not\Vdash} \sqsupset \ltimes \not\Vdash \in \texttt{Expr}_{t_2}$ and $\texttt{free}(\texttt{let}\ v = e_1\ in\ e_2) := \texttt{free}(v) - \{v\}$.
– If $\omega : t_1 \times ... \times t_n \to t \in \Omega_{\mathcal{M}}$ and $e_i \in \texttt{Expr}_{t_i}$ for all $i = 1, ..., n$ then $\omega(\mathbin{\not\Vdash}, ..., \ltimes) \in \texttt{Expr}_t$ and $\texttt{free}(\omega(e_1, ..., e_n)) := \texttt{free}(e_1) \cup ... \cup \texttt{free}(e_n)$.
– If $e_1 \in \texttt{Expr}_{Boolean}$ and $e_2, e_3 \in \texttt{Expr}_t$ then $\texttt{if} \mathbin{\not\Vdash} \texttt{then} \not\Vdash \texttt{else} \not\Vdash \texttt{endif} \in \texttt{Expr}_t$ and $\texttt{free}(\text{ if } e_1 \text{ then } e_2 \text{ else } e_3 \text{ endif}) := \texttt{free}(e_1) \cup \texttt{free}(e_2) \cup \texttt{free}(e_3)$.
– If $e \in \texttt{Expr}_t$ and $t' \leq t$ or $t \leq t'$ then $(\ \texttt{asType} \approx') \in \texttt{Expr}_{t'}$,$(\ \texttt{isType} \approx') \in \texttt{Expr}_{Boolean}$, $(\ \texttt{isKindOf} \approx') \in \texttt{Expr}_{Boolean}$ and $\texttt{free}((e\,\texttt{asType}t')) := \texttt{free}(e)$, $\texttt{free}((e\ \texttt{isTypeOf}\ t')) := \texttt{free}(e), \texttt{free}((e\ \texttt{isKindOf}\ t')) := \texttt{free}(e)$.
– If $\mathbin{\not\Vdash} \to \sqsupset \approx \smile \supset \approx (\gtrsim \mathbin{\not\Vdash};\ \gtrsim \not\Vdash = \not\Vdash \mid \not\Vdash) \in \texttt{Expr}_{t_2}$ and $\texttt{free}(e_1 \to \mathit{iterate}(v_1;\ v_2 = e_2 \mid e_3)) := (\texttt{free}(e_1) \cup \texttt{free}(e_2) \cup \texttt{free}(e_3)) - \{v_1, v_2\}$.

In order to properly address the subtyping relation, an expression of type t' is also an expression of a more general type t. Hence, for all $t' \leq t$, if $e \in \texttt{Expr}_{t'}$ then $e \in \texttt{Expr}_t$.

Using the syntax defined above, we can start to write assertions in OCL, embedding them in a UML model, as we will show in Example 1.

3 A Conceptual Framework for Static Verification of Dynamic Diagrams Consistency

The expressive power of object-oriented paradigm makes it better suited for development of large software systems than the traditional imperative paradigm. However, the statically checks enforced by e.g. C++ or Java compilers test for such syntactic and typing restrictions only that guarantee the lack of runtime type errors. This is the contracting and specification level that has been used for too many years in the past by most software developers. Obviously, this is not enough to prevent surprising and often disastrous behavior of programs. In other words, the checks done by compilers are only part of what is needed to reason about the behavior (i.e., the semantics) of software.

Software contracts are a necessary prerequisite for being able to introduce a notion of correctness: if you do not state what your program should do, you are lacking the norm to which to compare what your program does in reality. In defining class correctness we follow [16], p. 370:

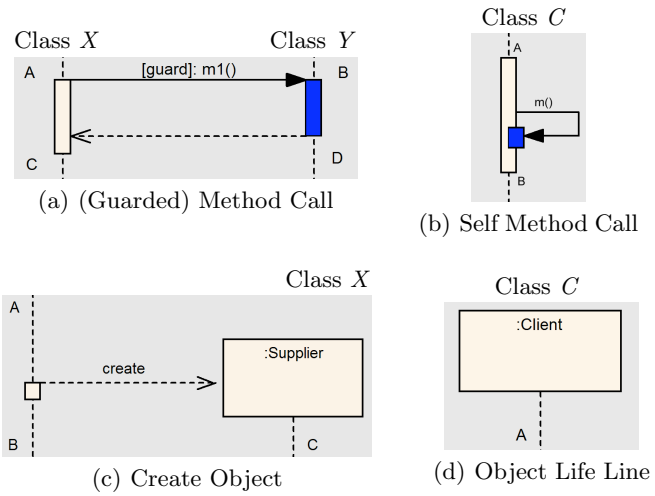
Definition 1 ([16]). *A class C is* correct *with respect to its specification if*

– *For any set of valid arguments $e_1, \ldots, e_n$ to a creation procedure p:*

$$\{Default_C \wedge Pre_{\mathtt{p}}[x/e]\}\ p\ \{Post_{\mathtt{p}}[x/e] \wedge Inv_{\mathtt{C}}\}$$

– *For every public method m and any set of valid arguments $e_1, \ldots, e_n$:*

$$\{Pre_{\mathtt{m}}[x/e] \wedge Inv_{\mathtt{C}}\}\ m\ \{Post_{\mathtt{m}}[x/e] \wedge Inv_{\mathtt{C}}\}$$



(a) (Guarded) Method Call

(b) Self Method Call

(c) Create Object

(d) Object Life Line

Fig. 1. Sequence Diagrams Basic Building Blocks

where $Default_C$ *denotes the assertion expressing that the attributes of* C *have the default values of their type.*

This notion clearly states what has to happen when we call a method in a state which satisfies $Pre_{\mathtt{m}}[x/e] \wedge Inv_{\mathtt{C}}$, but what happens when this does not hold? As already said, failure to meet any of the responsibilities stated in the contract results in a break of the contract, and indicates the existence of a bug somewhere in the design or implementation of the software or in the assertions themselves. Due to the size of most systems, the latter chance is not so unlikely. In this paper we want to focus on this situation, proposing our conceptual framework. In particular we want to check dynamic diagrams (and in particular Sequence Diagrams) against static diagrams and OCL specifications. In other words, the idea is to consider Class Diagrams and OCL specifications as a kind of meta-specification and all the dynamic diagrams as meta-code which has to conform the specification.

Thus we aim to guarantee that, by following the control flow on the diagram, the state is strong enough to satisfy the entry precondition of methods calls. For the sake of simplicity, we further restrict our attention to Sequence Diagrams which does neither involve concurrence nor timing constraints. This would require to define a much more complex verification method due to the complexity of considering more than one control flow at a time. We believe that, even with this restrictions, we have nevertheless a good level of generality to cover most of the existing software systems.

We will define our verification method by structural induction on the (graphical) syntax of Sequence Diagrams. Thus, in order to proceed, we need to specify

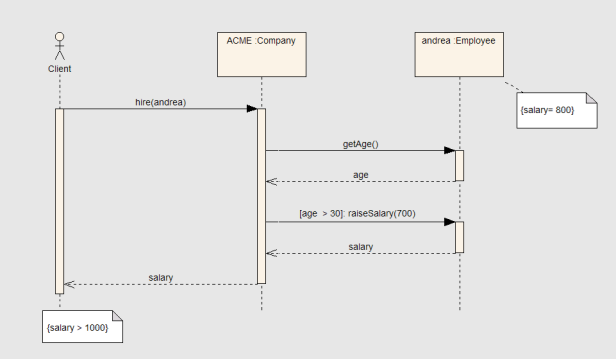


Fig. 2. An Example of Sequence Diagram

in a formal way the graphical syntax of Sequence Diagrams; mainly formalize the way to compose (connect) basic diagrams to obtain bigger ones. Since messages (calls) involves at most two objects at a time, we consider graphical blocks that refer to the lifetime of at most 2 objects at the same time.

Let us start, for the sake of simplicity, with the basic diagrams of Figure 1. They have entry and exit points which are graphically connected to exits and entries of other blocks. We introduce a function link that, given an entry point of a block, returns the exit point of the block to which the former is connected, and vice versa.

Most important than this, blocks can be nested. Inside the colored parts of blocks of type 1(a) and 1(b) we can plug blocks of type 1(b) or the left side of blocks of type 1(a) and 1(c). We can also plug any arbitrary sequential composition of the latter. We can trivially extend function link to take this kind of connections into account. The following example surely clarifies better than many words.

Example 1 (Decomposition of Sequence Diagrams in Blocks).

The sequence diagram of Figure 2 is decomposed in blocks according to our schema as in Figure 3. The whole diagram is composed of 2 blocks β_1, β_2 of type 1(d) connected to a outer block β_3 of type 1(a) which inside contains the sequential composition of two other blocks β_4, β_5, both of them of type 1(a). Thus function link in this case is defined as

$$\begin{array}{lll} \text{link}(B) = L & \text{link}(N) = B & \text{link}(C) = M \\ \text{link}(D) = O & \text{link}(E) = P & \text{link}(G) = Q \end{array}$$

while the blue boxes in the diagram indicate the block division. (Note: these boxes are not part of the UML syntax.) □

In UML Sequence Diagrams, especially version 2, there are also several diagrammatic elements which are naturally composed with inner blocks. This is

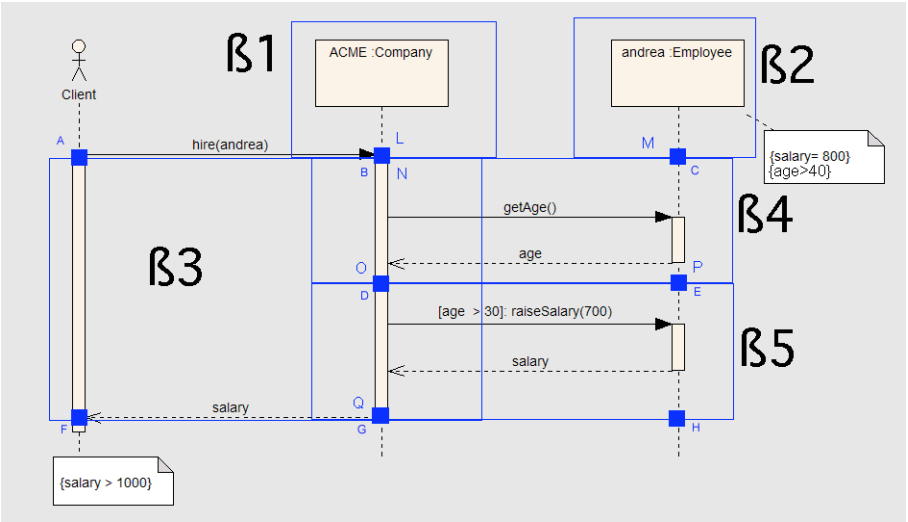


Fig. 3. Decomposition of Sequence Diagrams in Building Blocks

the case of all fragments, for example those in 4(a)4(b), where we define a new block by inserting blocks inside the "blank holes".

We can handle Interaction Use Fragments simply by properly extending the link function. We can handle Gates simply by glueing two corresponding diagrams along the corresponding connection points.

For economy of space, we do not explicitly show all other possible diagrammatic elements as they can be treated analogously.

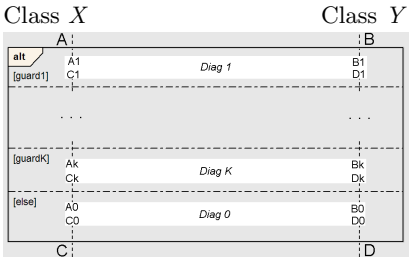
3.1 The Static Verification Method

We can now define our verification method by structural induction on the graphical syntax of Sequence Diagrams. The idea we follow here is first to introduce formula variables for all points of the blocks, then collect equalities between formula variables of the linked points and then add all the implications that must hold within the formula variables inside the various blocks according to their semantics. The implications that do not hold show manifestly an inconsistency of the sequence diagram.

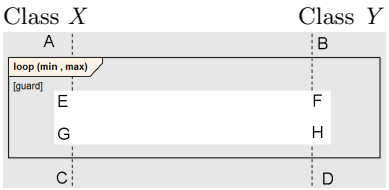
Let now present the various possibilities. We assume now that methods are called with actual arguments $e_1, \ldots, e_n$ (denoted by e) and that its formal parameters are $x_1, \ldots, x_n$ (denoted by x).

(Guarded) Method Call (1(a)) We need to impose that

$$\Phi_C = \mathrm{result}(\Phi_A) \wedge Post_{\mathtt{m1}}\,[x/e] \qquad \Phi_A = \Phi_{\mathrm{link}(A)}$$
$$\Phi_D = \Phi_B \wedge Post_{\mathtt{m1}}\,[x/e] \qquad \Phi_B = \Phi_{\mathrm{link}(B)}$$



(a) Alternative Fragment



(b) Loop Fragment

Fig. 4. Sequence Diagrams Fragments

and check that

$$\Phi_A \wedge guard \wedge \Phi_B \Longrightarrow Pre_{\texttt{m1}}\,[x/e] \tag{1}$$

$$\Phi_D \Longrightarrow Inv_{\texttt{Y}} \qquad\qquad \Phi_C \Longrightarrow Inv_{\texttt{X}} \tag{2}$$

where result(Φ_A) denotes the formula Φ_A modified (if it is the case) by inserting the result of method `m1` in the container specified by the call in class X.

Equation (1) prescribes that in order to call method `m1` the states of caller and callee, under the guard condition, have to be strong enough to guarantee that the precondition of the method holds. Equations (2) prescribe that the states reached by the caller and that by the callee do not invalidate the corresponding class invariant.

The Unguarded Method Call is just a particular case with $guard := True$.

Self Method Call (1(b)) Analogously to the previous case, we need to impose that

$$\Phi_B = \text{result}(\Phi_A) \wedge Post_{\texttt{m}}\,[x/e] \qquad\qquad \Phi_A = \Phi_{\text{link}(A)}$$

and check that

$$\Phi_A \wedge guard \Longrightarrow Pre_{\texttt{m}}\,[x/e] \qquad\qquad \Phi_B \Longrightarrow Inv_{\texttt{C}}$$

Create Object (1(c)) We need to impose that

$$\Phi_B = \Phi_A \qquad\qquad \Phi_A = \Phi_{\text{link}(A)} \qquad\qquad \Phi_C = Default_X$$

and check that $\Phi_C \Longrightarrow Inv_{\texttt{X}}$.

Object Life Line (1(d)) We need to impose that $\Phi_A = Inv_{\texttt{C}}$.

Alternative Fragment (4(a)) We need to impose, for all $0 \leq i \leq k$, that

$$\Phi_A = \Phi_{\text{link}(A)} \qquad\qquad \Phi_{A_i} = \Phi_A \wedge guard_i$$

$$\Phi_B = \Phi_{\text{link}(B)} \qquad\qquad \Phi_{B_i} = \Phi_B \wedge guard_i$$

$$\Phi_C = \bigvee_{0 \leq i \leq k} (guard_i \wedge \Phi_{C_i}) \qquad\qquad \Phi_D = \bigvee_{0 \leq i \leq k} (guard_i \wedge \Phi_{D_i}),$$

and check that $\Phi_D \Longrightarrow Inv_{\mathtt{Y}}$, $\Phi_C \Longrightarrow Inv_{\mathtt{X}}$, where

$$guard_0 := \neg \bigvee_{1 \leq i \leq k} guard_i$$

Note that the Alternative Method Call, as well as the Option and Break Fragments, are just a special case of this one.

Loop (4(b)) The loop fragment can be handled with special care. If we would have formulas within the body of loop which depend, even indirectly, upon each loop iteration then we would need to use a universal quantifier. Thus, to limit the expressive power of the underlying logic, we chose to treat just loop bodies which do not depend on iterations. With this assumption, we simply need to impose that

$$\begin{array}{lll} \Phi_A = \Phi_{\mathrm{link}(A)} & \Phi_E = \Phi_A \wedge guard & \Phi_C = \Phi_G \\ \Phi_B = \Phi_{\mathrm{link}(B)} & \Phi_F = \Phi_B \wedge guard & \Phi_D = \Phi_H \end{array}$$

The Fragments which inherently require concurrency, like Parallel, Critical Regions, Weak Sequential (when it does not boil down to Strict Sequentiality), cannot be considered without a concurrent semantics and thus fall out of our current scope.

Up to now we cannot consider too either the Assertion and Negative Fragments, as they would require to use an universal quantification over diagrams.

3.2 The Verification Method at work

Now we provide a complete example in order to show how our method can be applied in practice. We start to describe the static description of a software system, building the class diagram shown in Figure 5. In this diagram, the *Company* and *Employee* classes are defined. In particular, *Employee* has the following attributes: *age* of type *Integer*, name of type *String*, and *salary* of type *Double*. Similarly, the attributes of class *Company* are *location* and *name* (both of type *String*). *Employee* has two methods: *getAge*, which takes no arguments and returns an *Integer* value (the age), and *raiseSalary* which takes a *Double* and return a *Double* (the raised salary). *Company* has two methods: *fire* and *hire*, both of which takes an object of type *Employee* as argument. The method *hire* returns a *Double* (the salary of the hired employee). If the employee to be hired has an age greater than 30 years, the *hire* method call *raiseSalary* in order to increase the current employee's salary of 700 units. This scenario is depicted in the sequence diagram shown in Figure 3. Then we add the following OCL software contracts specifications for both Employee and Company classes. For the class Employee:

```
context Employee
  inv: (self.age >= 18)
```

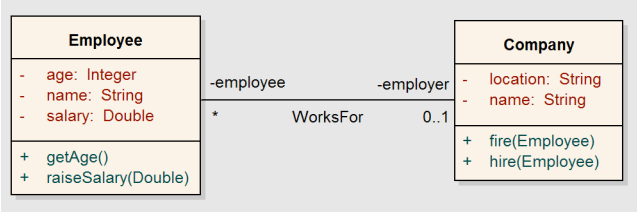


Fig. 5. Example Class Diagram

```
context Employee::getAge() : Integer
  pre:  true
  post: (result =  self.age)

context Employee::raiseSalary(amount : Double) : Double
  pre:  true
  post: (self.salary = (self.salary@pre + amount))
  post: (result = self.salary)
```

For the class Company:

```
context Company
  inv: self.employee->size() == self.employee->asSet()->size()

context Company::hire(p : Employee)
  pre  hirePre1: p.isDefined
  pre  hirePre2: self.employee->excludes(p)
  post hirePost: self.employee->includes(p)

context Company::fire(p : Employee)
  pre  firePre: self.employee->includes(p)
  post firePost: self.employee->excludes(p)
```

Now we show what we obtain with our method on the sequence diagram of Figure 3. The equalities on formula variables are:

$\Phi_B = \Phi_N = \Phi_L = Inv_{Company} =$
$(Company.employee \rightarrow size() \equiv Company.employee \rightarrow asSet() \rightarrow size())$
$\Phi_C = \Phi_M = (Andrea.age \geq 40 \wedge salary \equiv 800)$
$\Phi_D = \Phi_O = (Inv_{Company} \wedge Result \equiv Andrea.age)$
$\Phi_E = \Phi_P = (Andrea.age \geq 40 \wedge salary \equiv 800 \wedge Result \equiv Andrea.age)$
$\Phi_Q = (Inv_{Company} \wedge Result \equiv 1500)$
$\Phi_H = (Andrea.age \geq 40 \wedge salary \equiv 1500 \wedge Andrea.salary \equiv 1500 \wedge Result \equiv 1500)$
$\Phi_A = (Company.isDefined \wedge Andrea.isDefined)$
$\Phi_F = (salary \equiv 1500 \wedge Result \equiv 1500)$

$$\Phi_G = (Inv_{Company} \wedge Company.employee -> includes(Andrea))$$

While the implications that we have to check are

$$Inv_{Company} \wedge Andrea.age \geq 40 \wedge salary \equiv 800 \Longrightarrow True$$
$$Andrea.age \geq 40 \wedge salary \equiv 800 \wedge Result \equiv Andrea.age \Longrightarrow Andrea.age \geq 18$$
$$Inv_{Company} \wedge Result \equiv Andrea.age \Longrightarrow Inv_{Company}$$
$$Inv_{Company} \wedge Andrea.age \geq 30 \wedge Andrea.age \geq 40 \wedge salary \equiv 800 \Longrightarrow True$$
$$Inv_{Company} \wedge Result \equiv 1500 \Longrightarrow Inv_{Company}$$
$$Andrea.age \geq 40 \wedge Andrea.salary \equiv 1500 \Longrightarrow Andrea.age \geq 18$$
$$Company.isDefined \wedge Andrea.isDefined \wedge Inv_{Company} \Longrightarrow \quad Andrea.isDefined \wedge Company.employee -> excludes(Andrea) \qquad \text{(i)}$$
$$Inv_{Company} \wedge Company.employee -> includes(Andrea) \Longrightarrow Inv_{Company}$$

All implications can be verified except of (i). Actually looking at (i) we discover that there is nothing in the diagram which specifies that *Andrea* is not already an hired employee. If we add in the diagram an initial constraint specifying that $Company.employee -> excludes(Andrea)$ then we can prove the new (i) and then the diagram becomes consistent. This example suggests that in practical situations the assertions to be added in order to reach consistency can be quite easily derived by just inspecting the failing formula.

4 Applicability of the conceptual method

As already stated, the conceptual method that we have just presented is only a first step in a much more ambitious direction. Clearly in its generality it cannot be implemented because automatic proof of the verification formulas is undecidable. We think that even in this case the dimension of the state space generated is so large that it cannot be explored explicitly by model-checking techniques nor reasonably covered by testing.

However there are two possible nowadays well explored directions which we can follow from now on. One is that of using some proof assistant, like Coq[4]. The Coq tool is a formal proof management system: a proof done with Coq is mechanically checked by the machine. This direction of research is quite fertile in the literature. Several tools are being built on top of Coq, for object-oriented software verification purposes. For example Krakatoa[5] is a Java code certification tool that uses Coq to verify the soundness of implementations with regards to the specifications and Caduceus[6] is a verification tool for C programs.

However even computer-aided formal proofs tend to be humanly demanding and economically costly. An alternative is to use Abstract Interpretation Techniques where an abstraction of the semantics of the programs is automatically

[4] See http://coq.inria.fr/.
[5] See http://krakatoa.lri.fr/.
[6] See http://why.lri.fr/caduceus/.

computed. This leaves out all information about reachable states which is not strictly necessary for the proof. Of course if the abstraction is too precise, the computation cost are too high (resource exhaustion) and if it is too rough, nothing can be proved (false alarm). Although the best abstraction does exist, it is not computable, and so, must be found experimentally.

There has been a lot of research on these topics with promising results. Indeed recently we find tools based on Abstract Interpretation like Astree[7] [9] that can be used with great success for verification purposes of large C software systems.

5 Related Approaches and Future Works

In the literature we find also other completely different approaches to Model Consistency Verification, like the one of [24] where the information specified in class and Statechart Diagrams has been explicitly integrated into Sequence Diagrams. With these enriched diagrams, designers can *hopefully* identify gaps and contradictions in the specifications. In the future we would like to follow a similar idea and extend our method in order to collect state formulas from the Statechart Diagrams and inject them in the formula schema. This would *automatize* the identification of this kind of inconsistencies, without even cluttering the Sequence Diagrams.

The USE Tool [25] and the related works [26][27] represent another approach to achieve similar goals. These works consider validation by generating snapshots as prototypical instances of a model and comparing them against the specified model. In this way, snapshots provide immediate feedback and can be visualized using UML object diagrams. However the snapshots are defined by the user using a snapshot specification language (for object creation and operation calls). Eventually the sequence diagrams are automatically generated from the specified snapshot. Conversely, we prefer to use the standard UML notation for describing interactions, instead of a proprietary textual language, more suited to describe constraints than pictures, without mention the fact that modeling the system dynamics with diagrams is a standard practice in (object-oriented) software development.

There are also many approaches based on the ideas of software contracts and proof obligation generation. For example, the Java Modeling Language (JML) [28] is a formal behavioral interface specification language for Java. As such it allows one to specify both the syntactic interface of Java code and its intended behavior. However the JML approach is tightly coupled with the implementation programming language (in this case Java), whereas our framework is language independent. Moreover, our semantic-based verification approach can be applied even before the implementation starts. On the proof obligation side, the B formal method [29] provides a formal notation based on set theory and supported by automated tools, allowing specification, refinement, and proof. The B specification language come from Z and is not integrated neither in UML, nor in tool

[7] See http://www.astree.ens.fr/.

which support UML. There are works such as [30] which propose transformation rules in order to map OCL constraints into B formal expressions, but they are motivated by the lack of direct, automatic, static, and semantic-based verification tools in the UML/OCL arena. The conceptual framework presented here is a preliminary work toward the development of such tools.

References

1. Toth, H.: On theory and practice of Assertion Based Software Development. Journal of Object Technology **4** (2005) 109–130
2. Meyer, B.: Eiffel – the Language. Prentice-Hall, Englewood Cliffs, NJ (1992)
3. Meyer, B.: Applying "Design by Contract". Computer: Innovative Technology for Computer Professionals **25** (1992) 40–51
4. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, Los Angeles, California, January 17–19, New York, NY, USA, ACM Press (1977) 238–252
5. Cousot, P., Cousot, R.: Systematic Design of Program Analysis Frameworks. In: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, San Antonio, Texas, January 29–31, New York, NY, USA, ACM Press (1979) 269–282
6. Cousot, P., Cousot, R.: 'A la Floyd' induction principles for proving inevitability properties of programs. In Nivat, M., Reynolds, J.C., eds.: Algebraic Methods in Semantics. Cambridge University Press, Cambridge, UK (1985) 277–312
7. Cousot, P., Cousot, R.: A language independent proof of the soundness and completeness of generalized Hoare logic. Information and Computation **80** (1989) 165–191
8. Cousot, P.: Constructive Design of a Hierarchy of Semantics of a Transition System by Abstract Interpretation. Theoretical Computer Science **1-2** (2002) 47–103
9. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A Static Analyzer for Large Safety-Critical Software. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03), June 7–14, San Diego, California, USA, New York, NY, USA, ACM Press (2003) 196–207
10. Comini, M., Gori, R., Levi, G., Volpe, P.: Abstract Interpretation based Verification of Logic Programs. Science of Computer Programming **49** (2003) 89–123
11. Comini, M.: VeriPolyTypes: a tool for Verification of Logic Programs with respect to Type Specifications. In Falaschi, M., ed.: Proceedings of 11th International Workshop on Functional and (constraint) Logic Programming. Number UDMI/18/2002/RR in Research Reports, Udine, Italy, Dipartimento di Matematica e Informatica, Università di Udine (2002) 233–236
12. Comini, M., Gori, R., Levi, G.: Logic programs as specifications in the inductive verification of logic programs. In Dovier, A., Meo, M.C., Omicini, A., eds.: Declarative Programming – Selected Papers from AGP 2000. Volume 48 of Electronic Notes in Theoretical Computer Science., North Holland, Elsevier Science Publishers (2001) 1–16 Available at URL: http://www.elsevier.nl/locate/entcs/volume48.html.

13. Comini, M., Gori, R., Levi, G.: How to Transform an Analyzer into a Verifier. In Nieuwenhuis, R., Voronkov, A., eds.: Logic for Programming and Automated Reasoning. Proceedings of the 8th International Conference (LPAR'01). Volume 2250 of Lecture Notes in Artificial Intelligence., Berlin, Springer-Verlag (2001) 595–609
14. Comini, M., Gori, R., Levi, G.: Assertion based Inductive Verification Methods for Logic Programs. In Seda, A.K., ed.: Proceedings of MFCSIT'2000. Volume 40 of Electronic Notes in Theoretical Computer Science., North Holland, Elsevier Science Publishers (2001) 1–18 Available at URL: http://www.elsevier.nl/locate/entcs/volume40.html.
15. Alpuente, M., Comini, M., Escobar, S., Falaschi, M., Lucas, S.: Abstract Diagnosis of Functional Programs. In Leuschel, M., ed.: Logic Based Program Synthesis and Transformation – 12th International Workshop, LOPSTR 2002, Revised Selected Papers. Volume 2664 of Lecture Notes in Computer Science., Berlin, Springer-Verlag (2003) 1–16
16. Meyer, B.: Object-Oriented Software Construction, 2/E. Prentice-Hall, Englewood Cliffs, NJ (1997)
17. Gamma, E., Helm, R., Johnson, R., Vlissides, J.M.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA (1995)
18. Jézéquel, J.M., Train, M., Mingins, C.: Design Pattern and Contracts. Addison-Wesley, Reading, MA (2000)
19. Warmer, J., Kleppe, A.: The Object Constraint Language: Getting Your Models Ready for MDA. Addison-Wesley, Reading, MA (2003)
20. Object Management Group: (UML 2.0 OCL Specification) Document – ptc/05-06-06 (OCL FTF report – convenience document).
21. Object Management Group: (UML 2.0 Superstructure Specification, v2.0) Document – formal/05-07-04 (UML Superstructure Specification, v2.0).
22. Richters, M., Gogolla, M.: OCL: Syntax, Semantics, and Tools. In Clark, T., Warmer, J., eds.: Object Modeling with the OCL: The Rationale behind the Object Constraint Language. Springer-Verlag, Berlin (2002) 42–68
23. Richters, M.: A precise approach to validating uml models and ocl constraints. Technical report, University of Bremen (2001) draft.
24. Tsiolakis, A.: Integrating Model Information in UML Sequence Diagrams. In Baresi, L., Pezzè, M., Taentzer, G., eds.: Proc. of the Satellite Workshops of the 28th ICALP (GT-VMT 2001). Volume 50 of Electronic Notes in Theoretical Computer Science., North Holland, Elsevier Science Publishers (2001) 268–276
25. Richters, M.: The use tool: A uml-based specification environment. (2001)
26. Gogolla, M., Bohling, J., Richters, M.: Validating uml and ocl models in use by automatic snapshot generation. Software and System Modeling **4** (2005) 386–398
27. Richters, M., Gogolla, M.: Validating uml models and ocl constraints. In Evans, A., Kent, S., Selic, B., eds.: UML 2000 - The Unified Modeling Language, Advancing the Standard, Third International Conference, York, UK, October 2-6, 2000, Proceedings. Volume 1939 of Lecture Notes in Computer Science., Springer (2000) 265–277
28. Leavens, G., Cheon, Y.: Design by contract with jml (2003)
29. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (2005)
30. Marcano, R., Levy, N.: Transformation rules of OCL constraints into B formal expressions. Technical report, Université de Versailles (2002)

Author Index

Abd-El-Razik, Mohamed 61

Baruzzo, Andrea 110
Bouquet, Fabrice 1

Comini, Marco 110

Debricon, Stéphane 1
Dubrovin, Jori 93

Engels, Gregor 15

Güldali, Baris 15
Giese, Holger 77
Glesner, Sabine 77
Gupta, Atul 29

Junttila, Tommi 93

Jussila, Toni 93

Küster, Jochen M. 61

Lano, Kevin 45
Latvala, Timo 93
Legeard, Bruno 1
Leitner, Johannes 77
Lohmann, Marc 15

Nicolet, Jean-Daniel 1

Porres, Ivan 93

Schäfer, Wilhelm 77

Wagner, Robert 77
Whittle, Jon 1