Frédéric Boulanger, Michalis Famelis and Daniel Ratiu, editors

**Proceedings**

# 10th Workshop on Model Driven Engineering, Verification and Validation MoDeVVa 2013

**co-located with Models 2013**

**Miami, Florida, October 1st 2013**

Copyright © 2013 for the individual papers by the papers' authors.
Copying permitted for private and academic purposes.
This volume is published and copyrighted by its editors.

*Editor's addresses:*

**Frédéric Boulanger**
Supélec - Département informatique
3 rue Joliot-Curie
91192 Gif-sur-Yvette cedex, France
frederic.boulanger@supelec.fr

**Daniel Ratiu**
fortiss GmbH
Guerickestraße 25
80805 München, Deutschland
ratiu@fortiss.org

**Michalis Famelis**
Department of Computer Science
10 King's College Road
University of Toronto
Toronto, Ontario, Canada M5S 3G4
famelis@cs.toronto.edu

## Contents

# Preface

The MoDeVVA workshop series brings together researchers and practitioners interested in combining MDE with validation and verification. The 10th edition took place on the 1st of October 2013 and was co-located with MODELS'13 in Miami. The special topic of this edition was the use of models to increase the the usability of verification tools.

Out of the 13 papers submitted and reviewed by at least three members of the program committee, 9 were selected. About 30 participants attended this edition of the workshop.

In addition to the presentation of the selected papers from the technical program, MoDeVVA'13 featured an invited presentation by Marsha Chechik from the University of Toronto. Her highly inspiring talk gave an overview of research results in formalizing and checking the consistency and completeness of incomplete models.

This volume contains versions of the selected papers that the authors had the opportunity to enhance after the workshop and the fruitful discussions that occurred during the whole day. The papers where collected using the EasyChair conference system, formatted according to the LNCS style, and assembled using pdfLaTeX and the pdfpages package.

## Program Committee

Ait Sadoune, Idir, *Supélec - E3S, France*
Ammann, Paul, *George Mason University, USA*
Balaban, Mira, *Ben-Gurion University of the Negev, Israel*
Barroca, Bruno, *CITI-FCT/UNL, Portugal*
Bouquet, Fabrice, *University of Franche-Comté, France*
Bousse, Erwan, *Université de Rennes 1, France*
Cheng, Chih-Hong, *Fortiss - Munich Software and Systems Institute, Germany*
Derrick, John, *University of Sheffield, United Kingdom*
Fondement, Frédéric, *Université de Haute Alsace, France*
Jacquet, Christophe, *Supélec - E3S, France*
Legeard, Bruno, *Smartesting, France*
Lúcio, Levi, *MSDL, Canada*
Merayo, Mercedes, *Universidad Complutense de Madrid, Spain*
Minea, Marius, *Politehnica University of Timisoara, Romania*
Motta, Alfredo, *Politecnico di Milano, Italy*
Salay, Rick, *University of Toronto, Canada*
Scheidgen, Markus, *Humboldt-Universitt zu Berlin, Germany*
Schieferdecker, Ina, *FU Berlin/Fraunhofer FOKUS, Germany*
Sokenou, Dehla, *GEBIT Solutions, Germany*
Taha, Safouan, *Supélec - E3S, France*
Weißleder, Stephan, *Fraunhofer FOKUS, Germany*
Williams, James, *University of York, United Kingdom*
Wimmer, Manuel, *Vienna University of Technology, Austria*
Zurowska, Karolina, *Queen's University, Canada*

October 25, 2013
Gif-sur-Yvette

Frédéric Boulanger
Michalis Famelis
Daniel Ratiu

# Abstract of the Keynote: Partial Behavior Modeling

Marsha Chechik
Department of Computer Science
University of Toronto
chechik@cs.toronto.edu

Although software behavior modeling and analysis has been shown to be successful in uncovering subtle requirements and design errors, adoption by practitioners has been slow. One of the reasons for this is that traditional approaches to behavior models are required to be complete descriptions of the system behavior up to some level of abstraction, i.e., the transition system is assumed to completely describe the system behavior with respect to a fixed alphabet of actions. This completeness assumption is limiting in the context of software development process best practices which include iterative development, adoption of use-case and scenario-based techniques and viewpoint or stakeholder-based analysis; practices which require modeling and analysis in the presence of partial information about system behavior.

We believe that there is much to be gained by shifting the focus in software engineering from traditional behavior models to partial behavior models, i.e. operational descriptions that are capable of distinguishing known behavior (both required and proscribed) from unknown behavior that is yet to be elicited. Our overall aim is to develop the foundations, techniques and tools that will enable the automated construction of partial behavior models from multiple sources of partial specifications, the provision of early feedback through automated partial behavior model analysis, and the support for incremental, iterative elaboration of behavior models.

In this talk, I highlighted some of the results obtained in this line of research, joint work with Sebastian Uchitel and many students at the University of Toronto, University of Buenos Aires, and Imperial College London.

# A Framework for Testing UML Activities Based on fUML

Stefan Mijatov, Philip Langer, Tanja Mayerhofer, and Gerti Kappel

Business Informatics Group, Vienna University of Technology, Austria
{mijatov, langer, mayerhofer, gerti}@big.tuwien.ac.at

**Abstract.** In model-driven engineering (MDE), models constitute the main development artifacts. As a consequence, their quality significantly affects the quality of the final product. Thus, adequate techniques are required for ensuring the quality of models. We present a testing framework, comprising a test specification language and an interpreter, for validating the functional correctness of UML activities. For this purpose, we utilize the executability of a subset of UML provided by the fUML standard. As UML activities are employed for different purposes, from high-level process specifications to low-level object manipulations, the proposed testing framework not only allows to validate the correctness in terms of input/output relations, but also supports testing intermediate results, as well as the execution order of activity nodes. First experiments indicate that the proposed testing framework is useful for ensuring the correct behavior of fUML activities.

## 1 Introduction

In model-driven engineering (MDE), models are used to specify the structure and behavior of the system to be built. By using model transformations and code generation, artifacts, such as source code, database schema, and deployment scripts, can be generated from the models. This helps developers to abstract from technical details and increase their productivity by automating parts of the development process [1,13]. MDE shifts the development process from being code-centric to being model-based. As a consequence, it is of uttermost importance to ensure a high quality of the models. Otherwise, every error not captured at the model level is propagated to the final product [4].

One important quality aspect of models is *functional correctness*. For validating the functional correctness of models, precisely defined semantics of the used modeling language is a prerequisite. The semantics of a subset of UML [8], one of the most adopted modeling languages, has recently been precisely defined and standardized with fUML [9]. fUML specifies a virtual machine for executing UML models compliant to this subset, which consists of concepts for modeling classes and activities.

Although the semantics of fUML is precisely defined, adequate means for systematically testing fUML models are missing. We argue that a unit testing framework for fUML may provide the same benefits as for code-centric approaches, as it enables to validate the functional correctness of models, helps to avoid regressions, and test cases on model level may serve as input for testing the artifacts generated from the models.

Providing unit testing techniques for fUML activities is a challenging task because they can be specified for different purposes and on different levels of abstraction. Hence,

different means for validating their correctness are required. For instance, activities that serve as a high-level specification of processes cannot be tested adequately using assertions on input/output relations; constraints regarding the execution order of process steps modeled by activity nodes seem to be more adequate in such cases. For testing activities specifying low-level object manipulations and computations, assertions on input/output relations might be helpful. In addition, developers may also need to specify assertions on mutable intermediate results.

In this paper, we propose a dedicated *test specification language* and an accompanying *test interpreter* enabling the validation of the correct behavior of fUML activities. Using the test specification language the modeler can specify assertions on the execution order of the activity nodes, input and output values, and the runtime state of the model. The test interpreter is based on the reference implementation of the fUML virtual machine, which is used to execute the activities under test and to obtain execution traces that are used for evaluating the assertions defined in the test specification.

The remainder of this paper is structured as follows. In Section 2 we introduce a motivating example used to present our model testing approach throughout this paper. Section 3 gives an overview of fUML and the execution traces used for evaluating test cases on fUML activities. In Section 4 our test specification language for fUML activities and our test interpreter for evaluating them are presented. Related work is addressed in Section 5. Section 6 concludes this paper with an outlook on future work.

## 2 Motivating Example

In this section, we introduce a simple example fUML model specifying the withdrawal functionality of an automatic teller machine (ATM), which serves as running example throughout this paper, and discuss some test cases for validating its correct behavior. From these test cases, we derive the requirements that we aim to address with the proposed testing framework for fUML activities which is presented in Section 4.

An excerpt of the class diagram specifying the structure of the ATM system is depicted in Figure 1. An ATM card (class *Card*) has a *number* and a *pin* and is associated with exactly one account (class *Account*) which has a unique *number* and a *balance*. For realizing the withdrawal functionality, the classes *ATM* and *Account* have dedicated operations, called withdraw, validatePin, and reduceBalance.

The activities specifying the behavior of the operations withdraw and reduceBalance of the classes *ATM* and *Account* are shown in Figure 2. The activity specified for the operation withdraw (cf. ATMWithdrawActivity in Figure 2) requires as input the client's card as well as the PIN and the amount of money to be withdrawn entered by the client. First, it checks whether the PIN is valid by calling the operation validatePin. If the PIN is valid (i.e., the operation validatePin provides true as output), the operation reduceBalance is called for the account associated with the provided card. If this operation returns true, the ATMWithdrawActivity provides also true as output indicating the successful withdrawal; otherwise it returns false. The activity specifying the behavior of the operation reduceBalance (cf. AccountReduceBalanceActivity in Figure 2) takes as input the amount of money to be withdrawn and checks whether it exceeds the account's balance. In case the balance is not exceeded, the balance is accordingly updated and true is returned; otherwise false is returned.

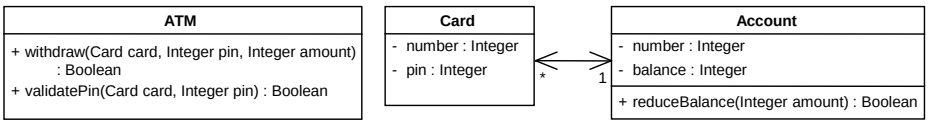


Fig. 1: Classes of the ATM system

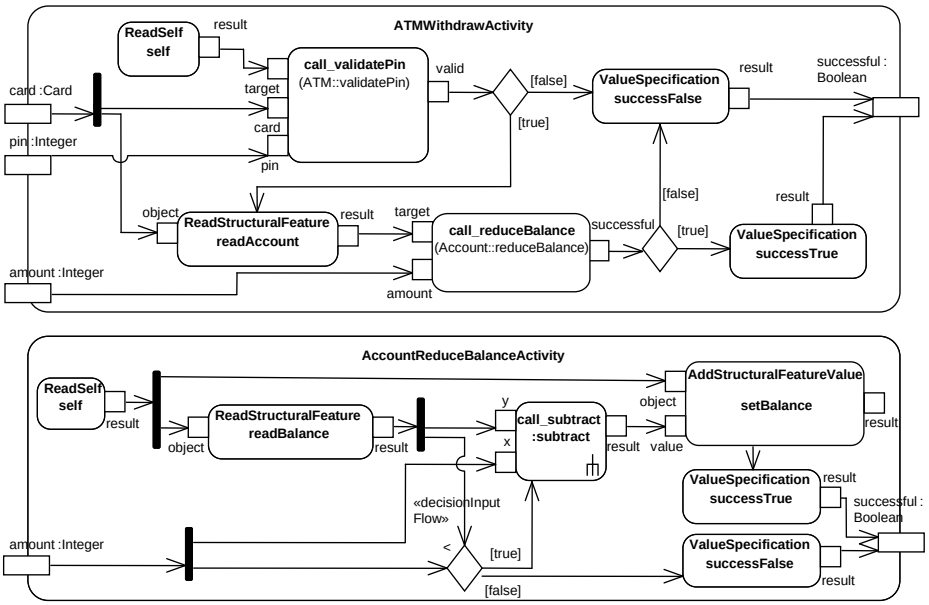


Fig. 2: Activities of the ATM system

For validating the correct behavior of the activity ATMWithdrawActivity, which specifies the withdrawal functionality of the ATM, we state the following test cases.

**Test case 1.** If the correct PIN is provided and the amount of money to be withdrawn does not exceed the balance of the client's account, the balance of the account should be reduced by the withdrawn amount and the activity should return true. Furthermore, the validation of the PIN should happen *before* the balance is reduced.

**Test case 2.** A withdrawal should also be possible if the amount of money to be withdrawn is equal to the balance of the account.

**Test case 3.** If the amount of money to be withdrawn exceeds the balance of the account, the balance must remain unchanged and the activity should return false.

A testing framework for fUML activities has to provide the means for expressing and evaluating these and similar test cases. Therefore, it has to fulfill the following requirements.

**R1: Execution order.** It should be possible to test the chronological order in which activity nodes are executed during the execution of the activity under test. Also other activities that are called from the activity under test should be considered. Furthermore, it should be possible to state the *relative* execution order of activity nodes without having to state the order of all activity nodes that are expected to be executed.

**R2: Input / output validation.** The testing framework should enable to check whether an input of an activity results in a given output. Further, the same should be possible for activity nodes contained by activities to allow for testing intermediate results.
**R3: State validation.** Assertions regarding the runtime state of the tested model, consisting of objects, their feature values, and links, should be possible for any point in time as well as for time periods of the execution of the activity under test.
**R4: Test input data.** The testing framework should allow to specify input data for the parameters of the activity under test in order to test different execution scenarios.

## 3 Foundational UML

The fUML standard [9] provides a formal definition of the execution semantics of a subset of UML 2. This subset contains the structural and behavioral Kernel of UML, as well as a major subset of the UML sublanguages Activities and Actions. Its semantics is defined through an operational approach by the specification of a virtual machine providing the capability of executing fUML-compliant models.

Whereas the standardized fUML virtual machine provides the facilities to execute activities and retrieve the output values for their parameters, it lacks in providing means for analyzing the performed model execution. To address this shortcoming, we extended the reference implementation of the virtual machine[1] in previous work [7] with the functionality of recording *execution traces* during the execution of activities. An excerpt of our metamodel for capturing execution traces is depicted in Figure 3. A trace provides information about the executed activities (class ActivityExecution), the executed activity nodes (class ActivityNodeExecution), as well as the chronological order in which these activity nodes have been executed (references chronologicalPredecessor, chronologicalSuccessor). Furthermore, the trace captures the input and output (classes Input, Output) of the execution of actions (class ActionExecution) and records the call hierarchy among activities (class CallActionExecution), as well as the input and output (classes InputParameterSetting, OutputParameterSetting) of activities. Also the evolution of the runtime state, that is, objects and links existing at any specific point in time during the execution, is captured by the trace: each modification of a value (class ValueInstance) is recorded by capturing a snapshot of the modified value (class ValueSnapshot). It is worth noting that for each execution of an action or activity the trace captures which snapshot of a value was provided as input to the execution and which snapshot resulted as an output of the execution (references to ValueSnapshot). Furthermore, the trace captures the destroyer and creator (references destroyer, creator) of values.

In summary, the trace of an executed activity enables to reason about the execution order of activities and activity nodes, inputs and outputs, and the runtime state of the executed model at a specific point in time of the execution. It therefore builds the crucial basis for the proposed testing framework for fUML activities presented in the following.

## 4 Testing Framework

In this section we first present a dedicated test specification language for expressing test cases on fUML activities and illustrate its usage on the ATM example introduced

[1] `http://fuml.modeldriven.org/`

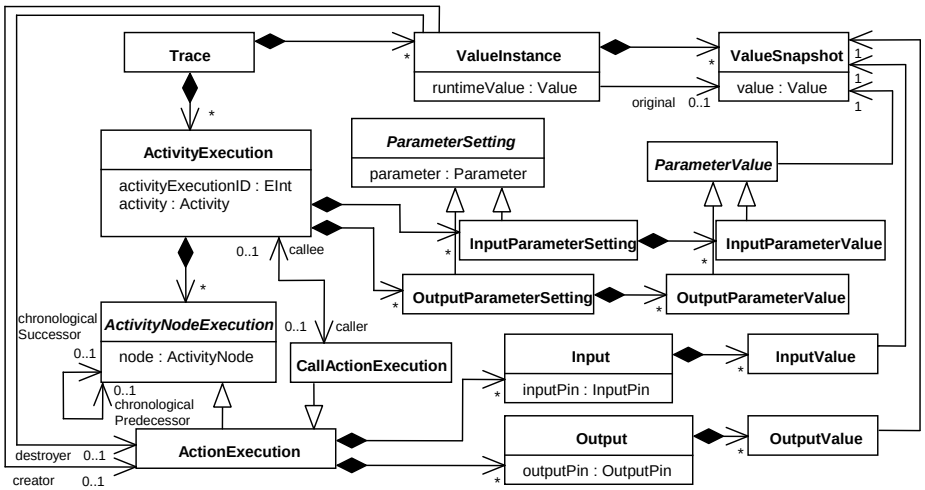


Fig. 3: Trace metamodel for fUML

in Section 2. Subsequently, we present a test interpreter capable of evaluating test cases as well as the result from evaluating the ATM test cases.

We provide an implementation of our testing framework integrated with the Eclipse Modeling Framework (EMF) [14]. For more detailed information about the implementation, we kindly refer the interested reader to our project website[2].

### 4.1 Test Specification Language

Our test specification language for fUML activities enables to specify a *test suite* which consists of import declarations, test scenarios, and test cases. An *import declaration* is used for referencing elements of the fUML model containing the activities under test. *Test scenarios* allow to specify objects, their feature values, and links, which can be used as input for the activities under test, as well as in state assertions (which are explained below). Each *test case* has a name, refers to the activity under test, and is composed of a set of assertions, which may either concern the execution order of activity nodes or the runtime state of the model. An *execution order assertion* is used for validating the execution order of activity nodes during the execution of the activity under test. In such assertions, the proposed language also allows to test the execution order of nodes contained by called activities and to accept any unspecified node before, in between, or after specified nodes using wildcard characters. The wildcard character * stands for an arbitrary number of activity nodes being executed, whereas _ stands for the execution of exactly one arbitrary activity node. A *state assertion* validates the state of objects at a certain point in time of the execution of the activity under test. For expressing the point in time at which the state of an object should be validated, an activity node has to be stated in combination with a temporal operator and a temporal quantifier. The *temporal operators after* and *before* are used to define whether the snapshots of an object captured before or after the execution of the stated activity node should be

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

checked. The *temporal quantifier always* denotes that *all* snapshots captured before or after the stated node should be checked, whereas the quantifier *exactly* denotes that only the *single* snapshot captured directly before or after the execution of the stated node should be checked. The properties of the object that shall be validated are defined in the *state expression* of a state assertion, which can either be an *object state expression* for validating the state of a complete object (i.e., all its properties), or it can be a *property state expression* for validating the value of a single property of an object.

Listing 1 shows the specification of the test cases for the ATM system defined in Section 2 using the proposed test specification language.

Listing 1: Test suite for the ATM system

```
1  import model.*
2  scenario TestData {
3    object atmTO : ATM {}
4    object accountTO : Account {
5      number = 323454676;
6      balance = 800;
7    }
8    object cardTO : Card {
9      number = 323454676;
10     pin = 1234;
11   }
12   link card_account {card = cardTo; account = accountTo;}
13 }
14 test testcase1 activity ATMWithdrawActivity (card = TestData.cardTO, pin = 1234, amount = 300)
15   on TestData.atmTO {
16     var account = readAccount.result;
17     var successful = ATMWithdrawActivity.successful;
18     assertOrder *, call_validatePin, *, call_reduceBalance, *;
19     assertState always before call_reduceBalance {
20       account::balance = 800;
21     }
22     assertState always after successTrue {
23       account::balance = 500;
24       successful = true;
25     }
26 }
27 test testcase2 activity ATMWithdrawActivity (card = TestData.cardTO, pin = 1234, amount = 800)
28   on TestData.atmTO { // only the differences to testcase1 are shown
29     assertState always after successTrue {
30       account::balance = 0;
31       successful = true;
32     }
33 }
34 test testcase3 activity ATMWithdrawActivity (card = TestData.cardTO, pin = 1234, amount = 900)
35   on TestData.atmTO { // only the differences to testcase1 are shown
36     assertState always after successFalse {
37       account::balance = 800;
38       successful = false;
39     }
40 }
```

After specifying the fUML model which we want to test using an import declaration (line 1), we define a test scenario (line 2–13) composed of one *ATM* object, one *Account* object, one *Card* object, and one link between the *Account* and the *Card* object.

The first test case (line 14–26) tests the activity ATMWithdrawActivity and provides as input the *Card* object defined in the test scenario and the Integer values 1234 and 300 for the parameters card, pin, and amount, respectively. In this test case, we first declare a variable *account* (line 16), which refers to the object provided as output by the action readAccount through its output pin result. This variable can now be used in state

assertions. In line 18, an execution order assertion is specified defining that the action call_validatePin should be executed before call_reduceBalance, whereas accepting the execution of any other activity node before, in between, and after these nodes using *. The state assertion in line 19–21 checks that before the operation reduceBalance is called, the balance of the account is always 800. The state assertion in line 22–25 checks that after the execution of the last activity node sucessTrue the balance of the account should always be 500 (i.e., it is and remains updated to 500 accordingly). Furthermore, this state assertion defines that the output (parameter successful) should be true.

The second test case (line 27–33) implements, similar to testcase1, the assertions regarding the correct behavior for a withdrawal of 800; hence, it is expected that the account's balance is updated to 0.

The third test case (line 34–40) tests the behavior of the activity ATMWithdrawActivity for the case that the amount to be withdrawn from the client's bank account (900) exceeds the account's balance (800). Accordingly, we assert that the balance of the client's account is not modified and that the output of the activity is false.

### 4.2 Test Interpreter

For executing and evaluating test cases on fUML activities specified in the presented test specification language we make use of the fUML virtual machine, as well as of execution traces obtained from executing the activities under test (cf. Section 3).

The process of executing and evaluating tests is shown in the Figure 4. The input provided to the test interpreter consists of the fUML model to be tested and the test suite. Each test case in the test suite is evaluated by executing the activity under test using the fUML virtual machine with the parameter values defined in the test case. From this execution, we obtain an execution trace, which is used to evaluate each assertion of the test case. Finally, a test report is generated which provides the test verdict.

To evaluate *execution order assertions*, we simply investigate the ActivityNodeExecution instances contained in the execution trace, which represent the executed activity nodes, as well as the links between them defined for the references chronologicalPredecessor and chronologicalSuccessor.

These snapshots maintain all different versions of the object that existed during the entire execution of the activity under test. In fUML, objects can only be modified by certain kinds of actions, which all provide the modified object as output. Consequently, in the execution trace, an ActionExecution instance that represents the execution of such an action also refers to the ValueSnapshot instance representing the modified object

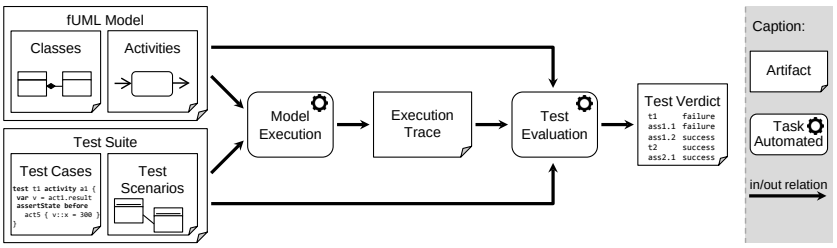


Fig. 4: Test interpreter

as output (cf. Output, OutputValue). Thus, to evaluate a state assertion, we obtain the ValueSnapshot instances of the ValueInstance representing the object of interest, which are referenced as outputs by the ActionExecution instances that have been executed in the time period specified in the state assertion (e.g., *always after actionX*). Whether the modification took place within the respective time period can be easily derived from the chronologicalPredecessor/Successor of the activity node defined in the state assertion. The resulting set of snapshots is then checked concerning the specified condition.

When executing the test suite defined for the ATM system in Listing 1, the test interpreter reports failures for all test cases.

For test case 1, a failure is reported for the state assertion defined in line 24–27 because the account's balance was updated to -500. The bug causing this failure resides in the activity AccountReduceBalanceActivity: the incoming edges of the input pins of the action call_subtract have to be switched to calculate the account's new balance.

When evaluating test case 2, the execution order assertion, as well as the second state assertion (validating that the balance was set to 0) fail, because the decision node of the activity AccountReduceBalanceActivity defines that a withdrawal is only possible if the amount of money to be withdrawn is smaller than the account's balance (amount $<$ account::balance). However, according to the test case, the withdrawal should also be possible if the amount is *equal* to the account's balance (amount $\leq$ account::balance).

Test case 3 fails because no output value was provided for the output parameter successful of the ATMWithdrawActivity. The bug leading to this failure was introduced at the action successFalse of the activity ATMWithdrawActivity. This action has two incoming control flow edges, whereas only one of them can provide a control token but never both, which is, however, required to execute this action.

In summary, our testing framework fulfills the requirements identified in Section 2. It enables to assert the execution order of activity nodes (requirement R1), to evaluate the input and output of activities and actions (R2), as well as the runtime state of the tested model at a specific point in time of the executing (R3), and it enables to define test input data for testing different execution scenarios (R4).

## 5 Related Work

Gogolla *et al.* [2] propose a UML-based specification environment (USE) for analyzing UML models, where the structure is specified with class diagrams and the behavior with operations. Class invariants and pre- and post-conditions of operations can be specified using OCL [10], which can then be validated on snapshots of the system state (i.e., objects and links existing at a certain point in time). The behaviors of operations are defined using their own imperative language and are therefore executed as specified in sequence diagrams leading to changes of the system state (i.e., snapshots).

Dinh-Trong *et al.* [3] present an approach for testing UML design models consisting of class diagrams, interaction diagrams, and activity diagrams by simulating the model's behavior and validating OCL class invariants and pre- and post-conditions of operations. In this approach a test case consists of the definition of the initial objects and links of the system under test and a sequence of operation calls. For executing test cases, Java code is generated from the UML model under test. The generated code simulates the behavior of the defined activity diagrams which is specified with their own action language JAL. For evaluating OCL constraints during the simulation, USE is applied.

Pilskalns *et al.* [12] present an approach for testing UML models composed of class and sequence diagrams. OCL class invariants and pre-/post-conditions of operations are used to validate the correct behavior of models. To execute test cases, a UML model is transformed into another format called Testable Aggregate Model (TAM) on which symbolic execution is applied. The OCL constraints are validated after the execution of each message defined in the sequence diagrams with USE.

The described approaches differ from our testing framework in the following respects. *(i)* For defining behavior in UML models, the presented approaches use their own formalisms. Thus, the execution semantics they apply is different from fUML's semantics. However, they are not restricted to the fUML subset. *(ii)* The presented approaches only evaluate invariants and pre- and post-conditions defined within the UML model for specified scenarios, whereas our approach enables the specification of arbitrary test cases that are separated from the UML model. *(iii)* The presented approaches only provide the possibility to validate changes on the system state caused by the execution of a whole operation, while our testing framework enables to also validate the state changes caused by distinct actions contained by the activity defining the operation's behavior. Furthermore, no validation of the execution order of operations or even actions is possible in these approaches.

Regarding the specification of test cases, the UML testing profile (UTP) [11] has to be named. It is an extension of UML, intended to support model-based testing by providing a standardized language for designing, visualizing, specifying, analyzing, constructing, and documenting the artifacts commonly required for testing software-based systems. While UTP allows to specify test cases, compared to the test specification language proposed in this paper, UTP is less expressive. For instance, execution order assertions cannot be expressed using UTP.

## 6 Conclusion and Future Work

We presented ongoing research towards developing a testing framework for validating the correct behavior of UML activities based on fUML. Our testing framework provides a dedicated test specification language for specifying the expected behavior of fUML activities in a set of test cases as well as a test interpreter which enables to evaluate the test cases by executing the activities under test and analyzing their actual behavior by utilizing execution traces.

First experiments with the proposed testing framework indicate its usefulness for ensuring the correct behavior of fUML models. In-depth case studies are necessary in future work in order to confirm this first impression. The experiments revealed the following interesting extensions of our testing framework left for future work.

**Parallelism.** fUML activities provide modeling concepts for specifying concurrent execution flows (e.g., fork nodes). The current implementation of our testing framework only supports the evaluation of test cases for *one possible* sequential execution order of concurrent flows and not *all possible* sequential execution orders.

**Object-centric testing.** Another possible extension of our framework is to support object-centric testing. User should be able to specify state validation expressions without referring to the activity under test directly, but by specifying how the state of the system changes during the execution. This would enable to express test cases on the

expected behavior of a fUML activity, without coupling the test case with the activity. In this respect OCL might be a valuable extension of our testing framework.

**Corrective feedback.** Our testing framework provides as output the information regarding the success or failure of each assertion. An interesting line of future work is to investigate techniques for slicing fUML models (e.g., [6]) based on failing assertion, which enables to determine the actual cause of a failing assertion. Based on this slice, recommendations for possible fixes could be computed.

**Model-based testing.** Another possible future research direction is to apply model-based testing approaches to generate test cases for fUML activities based on coverage criteria. We are currently investigating the approach proposed by Holzer *et al.* [5] who use UML activity diagrams for generating test cases for programs written in ANSI C.

## References


1. J. Bézivin. On the unification power of models. *SoSyM*, 4(2):171–188, 2005.
2. J. Brüning, M. Gogolla, L. Hamann, and M. Kuhlmann. Evaluating and Debugging OCL Expressions in UML Models. In *Tests and Proofs*, volume 7305 of *LNCS*, pages 156–162. Springer, 2012.
3. T. Dinh-Trong, N. Kawane, S. Ghosh, R. France, and A. Andrews. A Tool-supported Approach to Testing UML Design Models. In *Proc. of the 10th IEEE Conf. on Engineering of Complex Computer Systems (ICECCS)*, pages 519–528. IEEE Computer Society, 2005.
4. R. France and B. Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *Proc. of the Workshop on the Future of Software Engineering (FOSE) @ ICSE'07*, pages 37–54, 2007.
5. A. Holzer, V. Januzaj, S. Kugele, B. Langer, C. Schallhart, M. Tautschnig, and H. Veith. Seamless Testing for Models and Code. In *Fundamental Approaches to Software Engineering*, volume 6603 of *LNCS*, pages 278–293. Springer, 2011.
6. K. Lano and S. Kolahdouz-Rahimi. Slicing of UML Models Using Model Transformations. In *Model Driven Engineering Languages and Systems*, volume 6395 of *LNCS*, pages 228–242. Springer, 2010.
7. T. Mayerhofer, P. Langer, and G. Kappel. A Runtime Model for fUML. In *Proc. of the 7th Workshop on Models@run.time (MRT) @ MoDELS'12*, pages 53–58. ACM, 2012.
8. Object Management Group. OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1, August 2011. Available at: http://www.omg.org/spec/UML/2.4.1.
9. Object Management Group. Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.0, February 2011. Available at: http://www.omg.org/spec/FUML/1.0.
10. Object Management Group. OMG Object Constraint Language (OCL), Version 2.3.1, January 2012. Available at: http://www.omg.org/spec/OCL/2.3.1.
11. Object Management Group. UML Testing Profile (UTP), Version 1.2, April 2013. Available at: http://www.omg.org/spec/UTP/1.2.
12. O. Pilskalns, A. Andrews, A. Knight, S. Ghosh, and R. France. Testing UML designs. *Information and Software Technology*, 49(8):892–912, 2007.
13. D. C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *IEEE Computer*, 39(2):25–31, 2006.
14. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2nd edition, 2008.

# Building Test Harness from Service-based Component Models

Pascal André, Jean-Marie Mottu, and Gilles Ardourel

LUNAM Université - LINA CNRS UMR 6241 - University of Nantes
2, rue de la Houssinière, F-44322 Nantes Cedex, France
firstname.lastname@univ-nantes.fr

**Abstract.** In model-driven development, the correctness of models is essential. Testing as soon as possible reduces the cost of the verification and validation process. Considering separately PIM and PSM reduces the test complexity and helps the evolution of the system model. In order to test models before their final implementation, we propose an approach to guide the tester through the process of designing tests at the model level. We target model testing for Service-based Component Models. The approach produces test harness and provides information on how to bind the services of the components under test and how to insert the relevant data. The tests are then executed by plunging the harness into a technical platform, only dedicated to running the tests.

**Keywords:** Test Harness, Model Conformance, Test Tool, MDD

## 1 Introduction

Testing as early as possible reduces the cost of Verification and Validation (V&V) [1]. In Model-Driven Development (MDD), V&V is improved by directly testing models for correctness [2]. Testing models helps to focus the effort on the early detection of *platform independent* errors, which are costly when detected too late. Testing models does not consider implementation specific errors and thus reduces the complexity of testing [3]. Designing the tests on models ease their adaptation when models are refactored. However, building and running tests on models remain a challenge.

We target service-based component models with behaviour, including data and communications. The specification detail level is sufficient enough to enable various kinds of test at the model level. While testing such software components, the encapsulation should be preserved. Therefore, finding where and how providing test data is difficult: variables are accessed through services, which are available from the interface of the component but also from other required services. Designing the *test fixture* of a service implies a sequence of service calls additionally to the component initialisation. In [4], Gross mentioned issues in Component-Based Software (CBS) testing: testing in a new context (i.e. development context one, deployment context ones), lack of access to the internal

working of a component. Ghosh et al. [5]. also identify several problems: the selection of subsets of components to be tested and the creation of testing components sequences. Our work is concerned with these issues.

We consider the building of *test harness* for unit and integration testing dedicated to Service-based Component Systems. Test harnesses are used to provide test data, to run the test, to get the verdict (fail or pass). We create test harness from the platform independent model (PIM) of the system under test (SUT) and from test intentions. Those intentions declare which services are tested in which context, with which data. The tester incrementally builds a harness to run the test in an appropriate context, especially without breaking the encapsulation. The tester needs assistance for this task.

In this paper, we propose an approach assisting the test of service-based component models. This approach integrates the tests at the model level and is provided with tools. *First*, we propose operating at the model level, we modelise the tests at the same abstraction level as the component model. We build *test harnesses* as component systems made of Components Under Test (CUT) and Test Components (TC), which are created to test. As a consequence, the assembly of test components and regular components can benefit from the development tools associated with the component model. Additionally, test components could be transformed into platform specific models, thus capitalising the early testing effort. *Second*, we propose a guided process to assist the tester to manage the complexity of the component under test. The test harness is obtained from transformations of the SUT, guided by the test intention, enriched by test components, and assisted by several analyses and heuristics proposing relevant information and ensuring model correctness through verifications. *Third*, we illustrate the approach on a motivating example, a platoon of vehicles. We have developed a set of prototype tools designed to experiment the proposals. It is implemented in COSTO, the tool support of the Kmelia component model [6].

## 2 Testing Service-based Component Model

This section introduces the testing of service-based component model and the challenges to be tackled. It describes our approach on a motivating example.

### 2.1 Service-based Component Model

A *component system* is an assembly of *components* which services are bound by *assembly links*. The interface of a component defines its *provided and required services*. The interface of a service defines a *contract* to be satisfied when calling it. The service may communicate, and the assembly links denote communication channels. The set of all the services needed by a service is called its *service dependency*. The required services can then be bound to provided services. These needs are either satisfied internally by other services of the same component, or specified as required services in the component's interface and satisfied by other components. A *composite* component encapsulates an assembly.

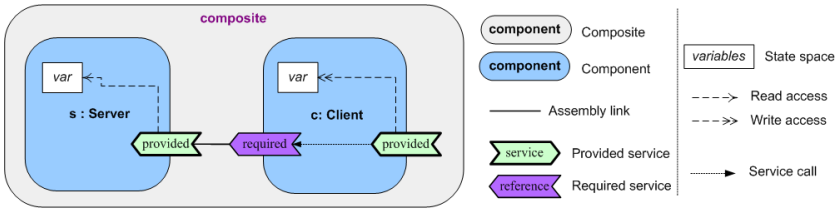


**Fig. 1.** Component and service notation

In the example of Figure 1, the component c: Client requires a service *provided* by the component s: Server, through an assembly link. This assembly is illustrated with the SCA notation [7]. We extended the SCA notation (right part of the legend of Figure 1) to make explicit the dependencies and data useful for testing: (i) typed data define the component state, (ii) services access these variables, (iii) provided services internally depends on required services.

Figure 2 represents a motivating example: a simplified platoon of several vehicles. Each vehicle computes its own state (speed and position) by considering its current state, its predecessor's state, and also a safety distance between vehicles. Each vehicle is a component providing its speed and position and requiring predecessor's speed and position. Three vehicles are assembled in that example. Each vehicle provides a configuration service conf initiating its state, a service run launching the platoon and requiring the service computeSpeed to calculate new position and speed. The leader is another component controlling its own values according to a position goal. The Figure 2 represents completely only the dependencies of computeSpeed, the service under test later.

### 2.2 Testing Components at the Modelling Stage

Conformance testing aims to control that system under test behave correctly according to their specification. With Service-based Components, we test that the behaviour conforms to the specification of their interfaces. In the context of

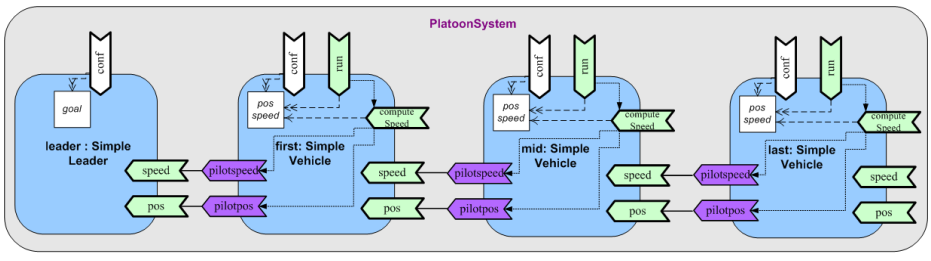


**Fig. 2.** Example of a component model of the Platoon system

Service-Based Component Applications, the test harness is designed by selecting the components under test, replacing their servers and their clients (the first with mocks, the last with test drivers), then providing test data to the test drivers and getting their verdicts [8]. Test harness should preserve component encapsulation. JUnit classes are an example of test harnesses for Java OO programs.

In MDD, Service-based Component System is modelised with a PIM, and part of its components (the SUT) is tested depending on a *test intention*. The test intention defines for each test which part of the system is tested (component(s)), with which entry points (service(s)), in which situation (SUT state), with which request (service call with parameters), expecting which results. At the beginning, the test intention is tester's knowledge, then she has to concretely prepare and run the test to get a verdict from oracles.

Testing such a SUT is not just creating test data and oracles, the difficulty is to consider components in an appropriate context (assembly of components), to run the test data on them, get the output to be checked with the oracles. The building of test harness to fulfil this task is an important work. Moreover, the process of creating a test harness is not trivial for models of components.

We consider several characteristics making the test of component models an original process: (i) Early specifications are often too abstract or incomplete to be executable. In order to test service-based component, we have to execute it in a consistent environment: its dependencies have to be satisfied by compatible services and every operation used in the environment has to be concrete enough or bound to a concrete operation. Furthermore, its environment must be able to run test cases. (ii) In strongly encapsulated components, finding where and how to insert data is challenging: variables are usually only accessible through services and some data are required from external services. (iii) At the model level, tests should be designed as models. Describing test artefacts like *mocks* and *drivers* in the component modelling language provides several benefits: it is easy to communicate with designers; tests follow the same validity rules as the model for both consistency and execution, allowing the use of the development tools associated with the component model. (iv) Models are subject to frequent refactoring or even evolutions, which can compromise the applicability of existing tests.

In this work, we are concerned with the building of test harness for service-based component models. The tester chooses the components and their services she tests (defined with the test intention). She orders the tests. She creates the test data based on existing works considering the specification of component: work of [9] with state machines, work of [10] performing LTS behaviours testing, etc. She creates oracles, for instance, based on the contracts usually available defining a service interface. Our approach assists the tester in building the test harness allowing her to run such test data and to apply such oracles.

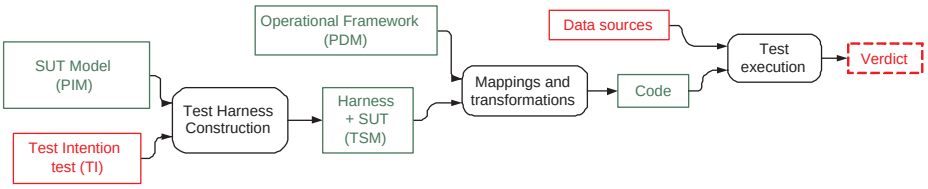


**Fig. 3.** Testing process overview

## 3 Assisting the test harness building

The construction process takes as input the System Under Test (SUT) model which is a PIM (Platform Independent Model) and a test intention (Figure 3). It is made of three activities (transformations). The first one builds the *test harness* as an assembly with the SUT in a *Test Specific Model* (TSM) at the model level. The second composes the harness with a *Platform Description Model* (PDM) to get an executable model (code). During the execution, the concrete data providers give the test data needed to run the test, and the concrete assert functions return the verdicts. Designing the oracles and the data sets influences the operational level. This point is out of concern in the rest of the paper.

### 3.1 Test Harness Construction

This section explains how the tester is guided to build a consistent test harness. As noted in [5] (and filed under the issue C3), it is sometimes necessary to consider different subsets of the architecture when testing a component, because of the influence (e.g. race conditions) of the other components on the SUT. The test harness is designed by (i) selecting the relevant subset of original components and services to test, (ii) replacing all or parts of their clients (resp. servers) by test *drivers* (resp. *mocks*), (iii) providing data sources. The test intention serves as a guideline during the construction process.

As an example, we are intent on testing the conformance of the computeSpeed ( safeDistance : Integer ) : Integer service of the mid component with the following *safety property*: *the distance between two neighbour vehicles is greater than a value safeDistance*. The service behaviour depends on (a) the recommended safe distance from the predecessor, (b) the position and speed of the vehicle itself and of its predecessor. Coming back to the example of Figure 2, testing the computeSpeed service of mid implies to give a value to the safeDistance parameter, to initialise the values of the pos and speed variables, which are used by the computeSpeed service and to find providers for pilotspeed et pilotpos which are required by computeSpeed. Only one component vehicle is under test here because other ones are not necessary to test the computeSpeed service, but a more complex architecture could have been retained.

The challenge for the tester is to manage the way these data can be provided. They can be provided by the test driver, by the configuration step, or by other components (original components or mock components). Finding the service to

be invoked in order to set the component in an acceptable state for the test is not trivial. Our goal is to help the tester to manage this complexity, in order to reduce testing effort. At this stage, assisting the harness building consists in:

1. Detecting missing features between the SUT and the test intention. This detection is achieved by the verification of two properties:
   (a) *Correctness*. All the service dependencies are satisfied in the scope of the test harness. No pending or incompatible dependencies remain.
   (b) *Testability*. All the formal test data are linked to values in the TSM (variables, parameters, results...).
2. Proposing candidates for the missing features: which service can configure data, can handle the test data or can fulfil missing required services?
3. Generating and linking test components (mocks, drivers as mirror services, bindings, abstract functions...).

In the TSM of Figure 4, the vtd test driver is responsible to put the computeSpeed service in an adequate context (providing input data and oracle). The service testcase1 of vtd contains only a computeSpeed call and an oracle evaluation. To fix the computeSpeed requirements, the test designer could (1) ask for mock components (with random values), (2) reuse the first component coming from the SUT (and fulfil the requirements of first ...) or (3) design its own mock. This variability enables several kinds of test and answers to the issue C3 of Ghosh et al. [5]. Note that a double integer mock component im1 is used here because it offers a better control of the delivered speed (intdata1) and position (intdata2), but a harness with two independent integer mocks providing a single intdata service would also be possible.

The *data* of computeSpeed test harness are bound in the following way: (a) the safeDistance parameter is assigned in the call sequence of the test driver, (b) the position and speed (pos and speed variables) of the mid vehicle under test are bound to the conf service of mid, (c) the im1 mock component playing the role of the predecessor is configured to return the position and speed of the predecessor.

### 3.2 Mappings and Transformations

According to the MDD principles, the test will be executed by composing the test harness with the PDM (Figure 3). The test data and test primitives (oracle...)

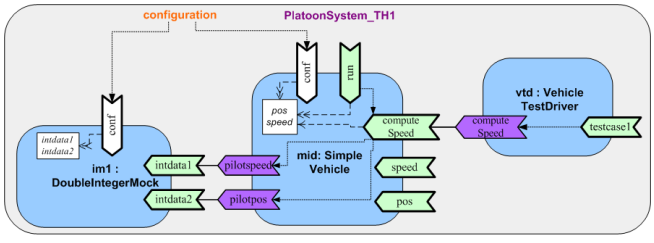


**Fig. 4.** A TSM : test harness for the mid component's service computeSpeed (SUT)

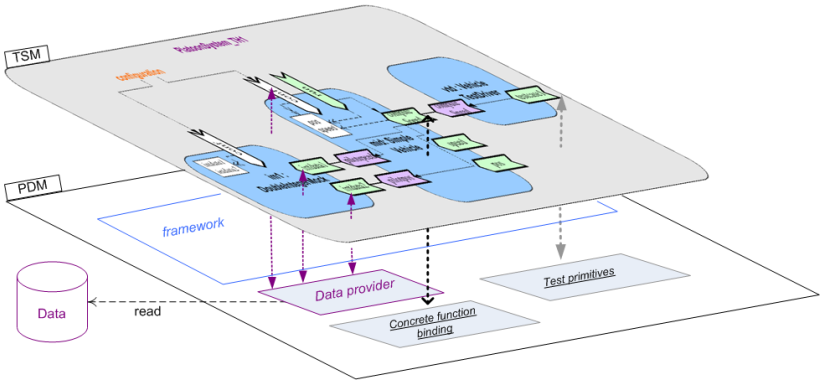


**Fig. 5.** Test harness Concrete data and Function Mapping

are provided by the PDM test support or implemented by the tester. The PIM may include primitive types and functions (numbers, strings, I/O...) that must also be mapped to the code level. These mappings are predefined in standard libraries or user-defined. High-level TSM primitives are connected to low level functions (PDM, code), as illustrated by Figure 5. If the mapping is complete and consistent, then the model is executable.

The goal of the *Mappings and Transformations* activity (Figure 3) is to fix this composition. Each service involved in the test must be executable in a consistent environment: its dependencies must be satisfied by compatible services, and every operation used in the environment must be sufficiently concrete or related to a specific operation. Every abstract type should be mapped to a concrete type. At this stage, assisting the harness building consists in (1) detecting missing mappings between the TSM and PDM (*Executability*), (2) proposing tracks for the missing mappings based on signatures, pre/post conditions... (all the primitive types and functions are mapped to concrete ones, the test data inputs have concrete entry points), (3) generating standard primitive fragments (idle functions, random functions...). The mappings are stored in libraries in order to be reused later and the entries can be duplicated to several PDM. This activity induces an additional cost at model level, but the errors detected are less expensive to solve than those of components and services. Moreover, if the component model is implemented several times targeting several PDM, the tests would be reused and part of the behaviour would have been checked before runtime, as proposed here.

## 4 Experimentation

The test process of Figure 3 has been instrumented using the COSTO/Kmelia tool[1]. Kmelia is a wide-spectrum language dedicated to the development of correct components and services [6]. Kmelia includes an abstract language for com-

[1] `http://www.lina.sciences.univ-nantes.fr/aelos/projects/kmelia/`

putations, instead of links to the source code as in related languages like SCA, Sofa2, Fractal or SystemC. This layer is useful to check models before transforming them to PSM. Kmelia is supported by the COSTO tool, a set of Eclipse plugins including an editor, a type checker and several analysis tools.

The test harness construction has been experimented on the platoon example. The goal was to build a harness to test the conformance of the service computeSpeed as introduced in Section 3.

- The test intention is a special Kmelia component, where only the name and description are mandatory. This trick enables to reuse the tool facilities and to consider test design at the model level.

```
TEST_INTENTION PlatoonTestIntention
DESCRIPTION "the vehicle will stop if it is too close to the previous one"
INPUT VARIABLES
  pos, previous_pos, mindistance:Integer;
OUTPUT VARIABLES
  speed:Integer
ORACLE
  speed=0
```

- The input data are provided and output data are collected in text files.
- During the process, the building tool (1) starts from a test intention, (2) asks the user to select target system and services (or proposes some if the test intention is detailed), (3) displays the board to match the test intention with the SUT (Figure 6), (4) proposes candidates by looking to variable types, (5) checks the correction and testability properties, (6) goes back to step (2) until the TSM is complete, (7) checks the executability properties and proposes candidates (from the libraries) for the missing elements, (8) generates the code.

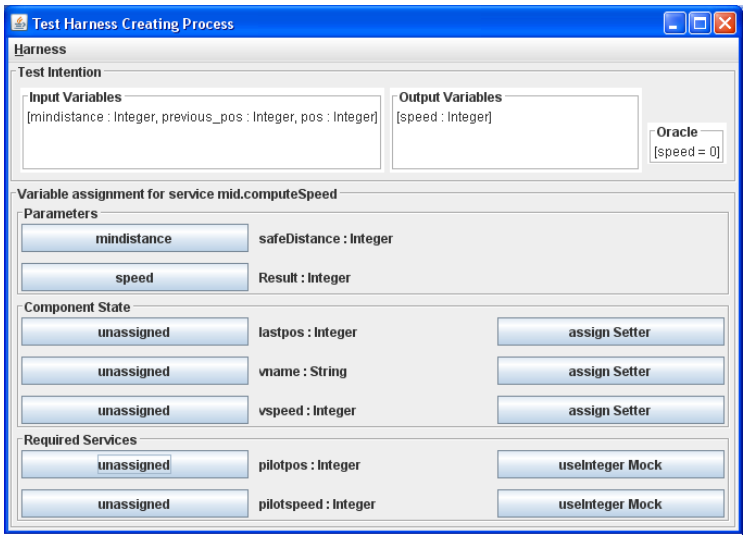


**Fig. 6.** Test harness assignments: mapping the test intention to the SUT

A web-appendix[2] shows the details of this example.

The harness building activities (transformations, decisions, assistance and generation) are implemented and integrated as COSTO functionalities. We developed the PDM framework in Java (7 packages, 50 classes, 540 methods and 3400 LOC). The service instanciation uses Java threads with an ad-hoc communication layer based on buffered synchronous channels (monitors). We keep strong traceability links from the code level to the PIM level for a better feedback on errors and also for animating the specification at the right level (GUI). We add specific support for contract checking (assertions and oracles).

Compared with the Junit practice, the tester could focus on the "business" part and did not need to care with Java details of the implementation. While the specification of the components was available, the tester could not modify them to make the test case more easy to write. She discovers the specification elements on-the-fly when mapping them to the test intention.

## 5 Related Work

There are several works interested in generating tests for testing components.

In [11], Mariani et al. propose an approach for implementing self-testing components. They move testing of component from development to deployment time. In [12], Heineman applies Test Driven Development to component-based software engineering. The component dependencies are managed with mocks, and tests are run once components can be deployed. In contrary, in our proposal we propose to test the components at the modelling phase, before implementation.

In [13], Edwards outlines a strategy for automated black-box testing of software components. Components are considered in terms of object-oriented classes, whereas we consider component as entity providing and requiring services.

In [14], Zhang introduces test-driven modeling to apply the XP test-driven paradigm to an MDD process. Their approach designs test before modelling when we design test after modelling. In [9], the authors target robustness testing of components using rCOS. Their CUT approach involves functional contracts and a dynamic contract. However, these approaches apply the tests on the target platform when we design them at the model level and apply them on simulation code.

## 6 Conclusion

In this paper, we described a method to integrate testing early in an MDD process, by designing test artefacts as models. The test designer is assisted in building component test harnesses from the component model under test and test abstract information through a guided process. The COSTO/Kmelia framework enabled to implement the process activities and to run the tests on the target specific execution platform with a feedback at the model level. The approach

[2] `http://www.lina.sciences.univ-nantes.fr/aelos/download/ModeVVa_app.pdf`

can be ported to any modelling languages supporting rich behavior modelling such as Sofa, rCOS or AADL. The benefits are a shorter test engineering process with early feedback and the tests are reified to be run again.

In future work, we will study different kinds of oracle contracts in order to find which logics are best fitted to express them in a testable way as well as ensuring a good traceability of the verdict.

## References


1. G. Shanks, E. Tansley, and R. Weber, "Using ontology to validate conceptual models," *Commun. ACM*, vol. 46, no. 10, pp. 85–89, Oct. 2003.
2. M. Gogolla, J. Bohling, and M. Richters, "Validating uml and ocl models in use by automatic snapshot generation," *Software and Systems Modeling*, vol. 4, no. 4, pp. 386–398, 2005.
3. M. Born, I. Schieferdecker, H.-g. Gross, and P. Santos, "Model-driven development and testing - a case study," in *First European Workshop on MDA with Emphasis on Industrial Application*. Twente Univ., 2004, pp. 97–104.
4. H.-G. Gross, *Component-based Software Testing With Uml*. SpringerVerlag, 2004.
5. S. Ghosh and A. P. Mathur, "Issues in testing distributed component-based systems," in *In First International ICSE Workshop on Testing Distributed Component-Based Systems*, 1999.
6. P. André, G. Ardourel, C. Attiogbé, and A. Lanoix, "Using assertions to enhance the correctness of kmelia components and their assemblies," *ENTCS*, vol. 263, pp. 5 – 30, 2010, proceedings of FACS 2009.
7. OSOA, "Service component architecture (sca): Sca assembly model v1.00 specifications," Open SOA Collaboration, Specification Version 1.0, March 2007.
8. C. R. Rocha and E. Martins, "A method for model based test harness generation for component testing," *J. Braz. Comp. Soc.*, vol. 14, no. 1, pp. 7–23, 2008.
9. B. Lei, Z. Liu, C. Morisset, and X. Li, "State based robustness testing for components," *Electr. Notes Theor. Comput. Sci.*, vol. 260, pp. 173–188, 2010.
10. B. Schätz and C. Pfaller, "Integrating component tests to system tests," *Electr. Notes Theor. Comput. Sci.*, vol. 260, pp. 225–241, 2010.
11. L. Mariani, M. Pezzè, and D. Willmor, "Generation of integration tests for self-testing components," in *FORTE Workshops*, ser. LNCS, vol. 3236. Springer, 2004, pp. 337–350.
12. G. Heineman, "Unit testing of software components with inter-component dependencies," in *Component-Based Software Engineering*, ser. LNCS. Springer Berlin / Heidelberg, 2009, vol. 5582, pp. 262–273.
13. S. H. Edwards, "A framework for practical, automated black-box testing of component-based software," *Softw. Test., Verif. Reliab.*, vol. 11, no. 2, pp. 97–111, 2001.
14. Y. Zhang, "Test-driven modeling for model-driven development," *IEEE Software*, vol. 21, no. 5, pp. 80–86, 2004.

# Feature-based Development of State Transition Diagrams with Property Preservation

Christian Prehofer

fortiss GmbH, Munich, Germany, Prehofer@fortiss.de

**Abstract.** In this paper, we consider incremental development of state transition diagrams by adding features, which add new states and transitions. The goal is to capture when properties of a state transition diagram are preserved when adding a feature. We classify several typical cases of such state transition diagram extensions and show when commonly used properties are preserved. In some cases, we add restrictions to the input events. In others, we need to transform properties to account for new failure cases. Properties are specified on the externally visible input and output events. To formalize the properties and to reason about internal state transition diagram extensions we use a computation tree logic with states and events.

## 1 Introduction

The idea of *incremental development* is to start with a base model and then to add small features in succession, which add previously unspecified behavior. Here, we extend state transition diagrams (SDs) by features, which means to add new states and transitions. With extension of an SD we refer to such a syntactic extension. The core question addressed here is whether properties are preserved when incrementally extending an SD by a new feature which adds new states and transitions.

As an example consider Figure 1, where a new, alternative path is added to an SD. By convention, added features are shown in red, with bold lines and bold labels. In this example, consider the main property that the output *"Issue ticket"* occurs eventually. As can be seen easily, the extension by this new path preserves this property. As a second example, consider the example in Figure 2, which adds a new loop. In this case, the above property (i.e., that a ticket is

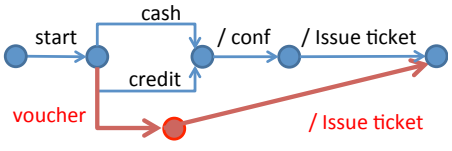


**Fig. 1.** Adding an alternative Path

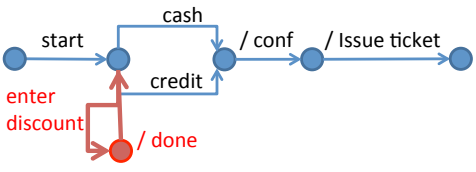


**Fig. 2.** Adding a local Loop

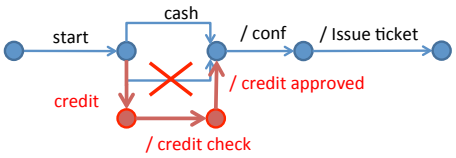


**Fig. 3.** Refining a Transition

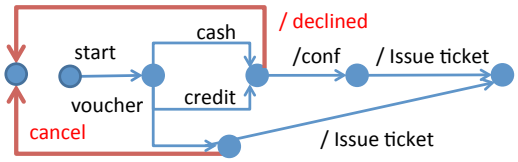


**Fig. 4.** Adding Failure Paths

issued eventually) is not preserved, as the loop may be traversed infinitely many times. Next, consider Figure 4, which adds two new failure paths. In this case, the above property that a ticket is issued is not preserved. We have two options here. As the first option, we may assume that such a failure does not occur. Alternatively, we may also transform the property to account for this new case.

More precisely, we consider the case where an SD is extended by a new feature, which adds new states and transitions. We classify several typical cases of such SD extensions and show when commonly used properties are preserved. In some cases, we also transform properties to account for new failure cases.

Our approach is, on a conceptual level, similar to adding aspects on a programming language level. For this, [3] has analyzed typical patterns of aspects w.r.t properties expressed in temporal logic. However, to our knowledge, such an approach has not yet been pursued for state transition diagrams. The main advantage here is that we can formalize properties on the control flow to determine when a property is preserved. Also, we adapt properties for expressiveness, unlike [3].

We formalize properties in a specific computation tree logic (CTL) which considers both states and events, following the logic in [9, 6]. Properties are specified on the externally visible traces of input and output events. To analyze when properties are preserved, we also need to capture the possible traversals of the state transition diagram, i.e., the internal view of states. This is why conventional CTL logics with consider either states or events are insufficient (see e.g. [7] for a comparison).

There exists ample work on refinement for SDs, e.g. [8], which aims to preserve all properties of an existing system, which is however not applicable in many cases. Hence we present custom solutions for specific features of SDs and specific classes of properties.

Other work on modularity for model checking [2, 5] considers the problem of extending automata models by new states and transitions. In these works,

composition of statecharts leads to proof obligations for specific properties to maintain. These are in turn to be validated by a model checker. Similar goals have been pursed in the context of aspect-modeling for state machines in [10]. These approaches require the specification and establishment of each individual property after the extension.

## 2 State Transition Diagrams and Event/State-based Temporal Logic

We model software systems by SDs, which we formalize as labeled transition systems.

**Definition 1 (Labelled Transition System).** *A* labelled transition system *(or* LTS*) is a structure* $L = (S, s_0, A, \rightarrow)$ *where*

- $S$ *is a set of states.*
- $s_0 \in S$ *is the initial state.*
- *Two disjoint sets* $I$ *and* $O$ *of input and output events and the silent event* $\tau$ *is not in* $I$ *nor* $O$*.*
- *Pairs of input and output events* $A = (I \cup \{\tau\}) \times (O \cup \{\tau\})$*, called labels.*
- $\rightarrow \subseteq S \times A \times S$ *is the* transition relation*; an element* $(r, \alpha, s) \in \rightarrow$ *is called a* transition*, and is usually written as* $r \xrightarrow{\alpha} s$*.*

To model practical examples we use explicit labels with a pair of input and output events. We write $(s, (i, o), s')$ or $(s, \alpha, s')$, where $\alpha = (i, o)$, for transitions. We let $A_\tau = A \cup \{\tau\}$. We let $\alpha, \beta, \ldots$ range over $A_\tau$. As $I$ and $O$ are disjoint, we also write $i$ instead of $(i, \tau)$ and $o$ instead of $(\tau, o)$.

Let $L = (S, s_0, A, \rightarrow)$ be an LTS. A sequence $(s_0, \alpha_0, s_1)(s_1, \alpha_1, s_2) \ldots \in \rightarrow^\infty$ is called a *path* from $s_0$; if a path cannot be extended anymore because it is either infinite or ends in a state without outgoing transitions, it is called a *maximal path*. We write $\text{state}_i(\pi)$ for the $i$'th state of the path, $i \geq 0$. We write $\text{event}_i(\pi)$ for the $i$'th event of the path, $i \geq 0$.

In the following, we briefly introduce a computation tree logic (CTL) based on both states and events. This is needed as we aim to separate the external view on an SD, in terms of events, and the internal view in terms of states, as discussed below.

Our logic, called ESCTL, is a special case of the UMC logic presented in [9]. In more detail, UMC permits multiple labels on a transition; here we use pairs of input and output events, which is easily embedded in UMC.

**Definition 2 (ESCTL).** *The syntax of the logic ESCTL(Event/State-based CTL) is defined by the following grammar where we let* $\phi, \phi', \ldots$ *range over* ESCTL-state formulas*:* $\phi ::= T \mid s \mid \neg\phi \mid \phi \wedge \phi' \mid E\varphi \mid A\varphi$ *where* $s \in S$ *and* $\varphi$ *is a path formula.* ESCTL-path formulae *are formed according to the following grammar:* $\varphi ::= \mathrm{X}\phi \mid \mathrm{X}_\chi\phi \mid \phi_\chi U_{\chi'}\phi' \mid \phi_\chi W_{\chi'}\phi'$ *where* $\phi$ *and* $\phi'$ *are ESCTL-state formula and* $\chi$ *and* $\chi'$ *are event formulae. An* event formula *is defined by the following grammar where we let* $\chi, \chi'$ *range over event formulas:* $\chi ::= true \mid \alpha \mid \neg\chi \mid \chi \wedge \chi'$

In the following we will say state and path formula instead of ESCTL-state and ESCTL-path formula.

Let $L = (S, s_0, A, \rightarrow)$ be an LTS. The satisfaction relation $\models$ between states $s \in S$ and state formulae is defined as usual. For a detailed treatment, we refer to [9]. Satisfaction of path formulae by maximal paths is defined as follows:

$$\begin{aligned}
\pi \models \mathrm{X}\phi \quad & \text{iff } \mathrm{state}_1(\pi) \models \phi; \\
\pi \models \mathrm{X}_\chi\phi \quad & \text{iff } \mathrm{event}_0(\pi) \models \chi \text{ and } \mathrm{state}_1(\pi) \models \phi; \\
\pi \models \varphi_\chi U_{\chi'}\varphi' \quad & \text{iff } \exists j \geq 0 : \mathrm{state}_j(\pi) \models \varphi \wedge \mathrm{state}_{j+1}(\pi) \models \varphi' \wedge \mathrm{event}_j(\pi) \models \chi' \wedge \\
& \quad \forall 0 \leq k < j.\mathrm{state}_k(\pi) \models \varphi \wedge \mathrm{event}_k(\pi) \models \chi; \\
\pi \models \varphi_\chi W_{\chi'}\varphi' \quad & \text{iff } \pi \models \varphi_\chi U_{\chi'}\varphi' \text{ or } \forall k \geq 0.\mathrm{state}_k(\pi) \models \varphi \wedge \mathrm{event}_k(\pi) \models \chi;
\end{aligned}$$

Satisfaction of event formulae by events is defined as usual [9]. Informally, $\pi \models \varphi_\chi U_{\chi'}\varphi'$ holds if for some path $\pi$, the property $\varphi$ holds on the states and $\chi$ holds on the events, until a transition with $(s_j, \alpha, s_{j+1})$ occurs, where $\chi'$ holds for $\alpha$ and $\varphi'$ holds for $s_{j+1}$.

We define the usual operators $F$ and $G$, as well as $F_\chi$, for a event $\chi$, as follows: $F = \neg T, EF\phi = E(T_{true}U\phi), AF\phi = A(T_{true}U\phi), EF_\chi\phi = E(T_{true}U_\chi\phi), EF_\chi = E(T_{true}U_\chi T), AG\phi = \neg EF\neg\phi, AG_\chi = \neg EF_{\neg\chi}, EG\phi = AG\neg\phi, E(\varphi_\phi U\varphi') = \varphi' \vee E(\varphi_\phi U_\phi\varphi')$. Without ambiguity, we omit $T$ in formulae. Similarly, we write $eU_\chi$ instead of $T_eU_\chi$.

Note that our combined event/state logic treats events and state separately, and many properties on states are easier to formalize than similar ones on events. For instance consider a property state $s$ implies state $s'$; we denote this as $AG(s \rightarrow AFs')$, which is defined as $AG(\neg s \vee AFs')$. On the other hand, an event $e$ implies $P$ is denoted as $AG(e \rightarrow P)$, defined as $AG((X_e(AFP)) \vee X_{\neg e}T)$.

## 3 SD Composition and Property Preservation

In the following, we first motivate our specifications in state/event logic and then classify properties on SDs. This is followed by a detailed analysis of SD composition for different cases.

We use our event/state logic in order to specify externally visible behavior and internal behavior, respectively. Assume an SD $S$ and a set of input traces $I$ consisting only of input events.

- We express properties on *externally visible behavior* by properties on input and output events. No formulae with states are permitted, as these are considered internal.
- *(Full) behavior specifications* use properties over states, input and output events created by $S$, using all of the above.

One reason why we use externally visible events for behavior is that two composed SDs may share input or output events. This is not possible if only names of states are considered. On the other hand, we need states to reason about internals of a composition, as discussed below.

### 3.1 Specification Patterns

Our goal is to study what properties are preserved when adding a new feature to an SD. For the properties to consider, we use the specification patterns as in [4]. From the extensive study in [4], it was observed that around 80%-90% of all properties in specifications are of the three kinds. First, we have the pattern "a leads to b", called response in [4]. Formally, we have, $AG(a \rightarrow AF_b)$. The other two classes, called universality and absence, are invariants stating that a property holds globally or something never happens. Such invariants must hold for all states and/or transitions, which means that validation is compositional in this sense. We should note here that the properties in [4] also consider different temporal scopes for the validity of the formulae. This is not considered here for simplicity.

In this paper, we hence focus on leads to or response properties. Together with the invariants, this covers a significant amount of specifications. The other pattens in [4] formalize precedence of states and chained response and precedence. We conjecture that other patterns can be handled in a similar way, but detailed analysis is left to further work.

### 3.2 SD Composition

In the following, we define formally how to extend an SD by a new feature. We also use SDs to denote such extensions, which are then glued together at specified join states. Based on join states, similar to join points in aspect-oriented languages, we use graphical notation to denote the composition of SDs. For a more formal definition, we refer to [8].

For adding a new path, consider the schematic view in Figure 5. By convention, the SD $E$ in red color represents the path to be added to the blue, base SD $S$. We assume that $E$ and $S$ have disjoint states, and further that $E$ has initial state $s'$ and a state $j'$. In $S$, we assume a start state $s$ and a join state $j$ for adding a path. The two SDs will be composed by merging these two states, $s'$ and $j'$, with the states $s$ and $j$, respectively. This is denoted by the light dashed lines. In the merged state, we use the states $s$ and $j$ for these. Formally, the states $s'$ and $j'$ are renamed to $s$ and $j$ in $E$, and the two SDs are then merged to obtain $S \cup E$. Note that we do not use the same names for the states $i$ and $j$ in $S$ and $E$ to be merged in order to avoid confusion when expressing preconditions on $E$ or $S$.

We also consider refinement of transitions as follows. We write $S - E$, where $E$ denotes a set of transitions in $S$ which are removed. Based on this, we can express transition refinement by $S - E \cup E'$ as shown in Figure 3. If $E$ removes a single transition for which a corresponding path with same start and end states and the same trigger event in the first transition in this path, then we denote this as a transition refinement by $S \cup^r E'$ and leave $E$ implicit (without ambiguity).

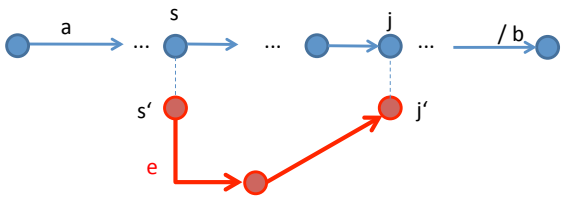


**Fig. 5.** Adding an Alternative Path (Schematic view)

### 3.3 Property Preservation for Response/Leadsto Properties

The main goal of this section is to define rules for the extension of an SD. As defined above, assume an SD $S$ which is extended to $S \cup E$ by an extension $E$. The goal is now to define criteria when a property for $S$, e.g. $S \models P$, also holds for $S \cup E$, i.e., $S \cup E \models P$.

This achieves modularity on properties w.r.t. extension, and also simplifies formal proofs as $S \cup E$ is larger than $S$ and $E$. Furthermore, we will see that many properties can be obtained easily by local analysis of the state diagram. We should observe also that some properties are not preserved by extensions. For this, we will transform the properties into extended properties with extra conditions.

**Adding Alternative Paths** We consider the schematic view of adding an alternative path in Figure 5. We use the states $s, s'$ and $j, j'$ as shown in this figure. However, the schematic case as in Figure 5 is simplified for illustration and does not show all cases covered by this rule. More precisely, this case is specified by the following assumptions: First, there is a path from $s$ to $j$ in $S$, i.e., $S \models AG(s \rightarrow EF_j)$. This means that $j$ is reachable from $s$ on at least one path; thus we add an alternative path to this. Second, $E \models AG(s' \rightarrow AF_{j'})$. This means that $j'$ is reached eventually when entering $E$. Furthermore, we assume $s \neq j$ to avoid trivial loops which are covered in a separate case below. Third, we assume that $j'$ is a final state in $E$. Hence we have no newly added infinite loops and termination only in state $j$ for $E$.

We consider first a property of the form $AG(a \rightarrow AF_b)$. The main idea here is to identify cases when an alternative path does not obstruct this property. In other words, the event $b$ is observed in any path after $a$, even with the new alternative path.

Assuming $S, E$ with states $s, s'$ and $j, j'$ as specified above for Figure 5. In case $S \models AG(a \rightarrow AF_b)$ holds for $S$, we obtain $S \cup E \models AG(a \rightarrow AF_b)$ if one of the following holds:

- $S \models AG(a \rightarrow (\neg s U_b))$
- or $S \models AG(s \rightarrow (\neg j W_b))$
- or $E \models AG(s' \rightarrow AF_b)$.

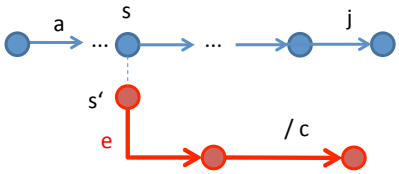


**Fig. 6.** Adding a Failure Path

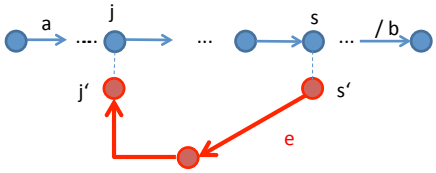


**Fig. 7.** Adding a Fallback Path

**Adding a Failure Path** We consider the schematic view of adding a failure path in Figure 6. The assumptions for this case are as follows: there is a failure event $c$ which occurs in $E$, $E$ is entered by event $e$ and furthermore that there is no paths back to the SD $S$. Formally, we have no $j'$ in this case and $E \models AG(e \rightarrow AF_c)$.

In case of leadsto properties, we have several cases. The base case happens when the property is not affected by the failure, i.e., if the added path is not taken before $b$, formally $a \rightarrow (\neg s U_b)$. Then, we have the case that $b$ always occurs in $E$, similar to above.

If these simple cases do not hold, we have the option to weaken the property. Assuming $S \models AG(a \rightarrow AF_b)$ and $AG(E \models e \rightarrow AF_c)$, we obtain $S \cup E \models AG(a \rightarrow AF_{c \vee b})$.

**Adding Fallback Paths** The idea of a fallback path is that a new path is added, similar to the alternative path case. Here, the new path leads to a state already traversed earlier. Thus we fall back to an earlier state and create a possible loop, which is different from the above cases.

We consider the schematic view of adding an alternative path in Figure 7. As above, the SD $E$ in red color represents the path to be added to the blue, base SD $S$, with join states $s$ and $j$. We assume that $E$ has initial state $s'$ and a final state $j'$.

The fallback case, as illustrated schematically in Figure 7, is specified by the following assumptions: First, there is a path from $j$ to $s$ in $S$, i.e., $S \models AG(j \rightarrow EF_s)$. Secondly, $E \models AG(s' \rightarrow AF_{j'})$. Third, $j'$ is a final state in $E$. The main problem in this case is that the fallback path adds a loop which may be traversed infinitely often.

Consider a response property of the form $a$ leads to $b$. The main idea here is to identify cases when an alternative path does not obstruct this property. In other words, the event $b$ is observed in any path after $a$, even with the new alternative path.

Assuming $S, E$ with states $s, s'$ and $j, j'$ as specified above for Figure 7. Assuming $S \models AG(a \rightarrow AF_b)$, we obtain $S \cup E \models AG(a \rightarrow AF_b)$ if

- $S \models AG(a \rightarrow (\neg s W_b))$
- or $E \models AG(s' \rightarrow AF_b)$.

For adding a fallback path, we have another important case. If we can avoid non-termination by the added loop of the fallback path, we can also establish

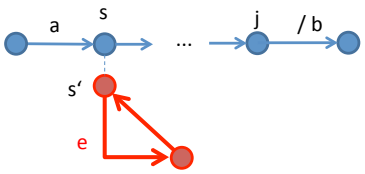


**Fig. 8.** Adding a local Loop

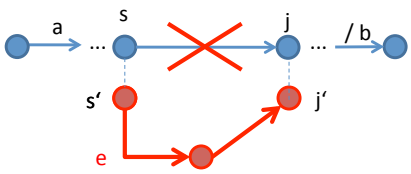


**Fig. 9.** Refining a Transition

the desired property. It is sufficient to express that in state $s$, the input $e$ does not occur infinitely often.

The problem is that we cannot express such fairness properties in a CTL-logic. We can formalize this as $S \cup E \models \neg AG(s \wedge X_e)$, but this is a CTL* formula and not a CTL formla. Fortunately, for our state/event logic, there also exists an extension, including model checking, for the $\mu$-calculus, which can embed CTL* [6]. Another, suitable solution is to add fairness conditions on the permitted input events, as done for CTL-logic in [1]. We expect that this can also be transferred to action/state CTL logics, but this goes beyond the scope of the paper.

**Refining Transitions** In this case, a transition is refined into several new states and new transitions. This is typical when moving from a higher level model to a more detailed model, which shows more details by refining a transition into several transitions. Note that removing transitions is a special case of this refinement. An example is shown in Figure 9.

The schematic case is illustrated in Figure 9. Formally, it is specified by the following assumptions: First, there is a transition from $s$ to $j$ in $S$. Second, $E \models AG(s' \rightarrow A_{j'})$. This means that $j$ is reached eventually when entering $E$. Hence we have no infinite loops and termination only in state $j$. Refining a transition could be modeled by removing a transition and adding a new path; for refinement it is however more effective to consider this in conjunction.

We consider first a property of the form $a \rightarrow AF_b$. The main idea here is to identify cases when the added path does not obstruct this property. In other words, the event $b$ is observed in any path after $a$, even with the new alternative path.

Assuming $S \models AG(a \rightarrow AF_b)$, we obtain $S \cup^r E \models AG(a \rightarrow AF_b)$ if

- $S \models AG(a \rightarrow (\neg sW_b))$
- or $S \models AG(s \rightarrow (\neg jW_b))$
- or $E \models AG(s' \rightarrow AF_b)$.

The cases above are as in the case of adding a transition: In the first case, $(\neg sUb)$ means $s$ never happens before $b$. Hence $b$ occurs always before the alternative path is taken. In the second case, $s \rightarrow (\neg jW_b)$ means this: if $s$ happens, then $b$ happens after $j$. This is to avoid the case that $b$ only occurs between $s$ and $j$. Finally, $E \models AG(s' \rightarrow AF_b)$ means that $b$ happens in $E$ when the path is taken. Hence the property also holds for the combined SD.

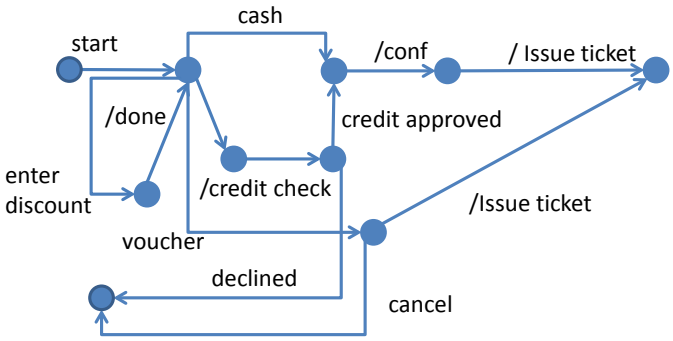


**Fig. 10.** Combining the Examples

### 3.4 Example

In the following, we apply the above results to the example in the introduction. We consider the base SD as in Figure 1 and the property $AG(start \rightarrow AF_{Issue\_ticket})$. Combining the examples from Section 1, we obtain the SD in Figure 10

1. Adding the voucher option in Figure 1 preserves this property.
2. Adding the discount option in Figure 2 does not preserve this property. Assuming a fairness precondition, this turns into $AG(start \rightarrow AF_{Issue\_ticket})$. More formally, we need to assume in the following that $enter\_discount$ does not occur infinitely often in sequence.
3. Refining the credit transition in Figure 3 preserves this last property.
4. Adding the failure cases as in Figure 4 does not preserve this property. We need to weaken the property as follows: $AG(start \rightarrow AF_{declined \vee cancel \vee Issue\_ticket})$, assuming the above fairness precondition regarding $enter\_discount$.

## 4 Conclusions

In this paper, we have presented a new approach for incremental development of state transition diagrams. We have developed a classification of individual features to be added to an SD, and then discussed when properties can be established. In some failure cases, we had to modify the properties to account for failure cases. In case of loops, we may need a notion of fairness for some properties. The conditions for the rules only assume properties of the SDs to be composed, which means that the rules are modular. Hence, one benefit is that we can validate the rules more efficiently. In many cases the conditions can be checked by simple analysis on the SD level.

We have formalized our rule by using a recently developed Event/State CTL logic. This permits us to express properties on externally visible behavior in

terms of input and output events, while expressing composition conditions by internal states. We have validated our rules by several examples in a workflow example. We assume that our rules and examples are easy to express in a model checker for our state/event logic, e.g. in the UMC tool [6].

The main novelty is to focus on different kinds of properties individually, which permits new ways for incremental development. Regarding other works on refinement, we note that usual notions of refinement preserve all properties, which is too strong for our case. Further work will address other patterns of property specifications, also including different scopes.

**Acknowledgements** The author would like to thank Sebastian Bauer, Martin Wirsing, Franco Mazzanti and Rolf Hennicker for stimulating discussions.

## References

1. C. Baier, J.-P. Katoen, et al. *Principles of model checking*, volume 26202649. MIT press Cambridge, 2008.
2. C. Blundell, K. Fisler, S. Krishnamurthi, and P. Van Hentenrvck. Parameterized interfaces for open system verification of product lines. In *Automated Software Engineering, 2004. Proceedings. 19th International Conference on*, pages 258 – 267, sept. 2004.
3. S. D. Djoko, R. Douence, and P. Fradet. Aspects preserving properties. In *Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation*, PEPM '08, pages 135–145, New York, NY, USA, 2008. ACM.
4. M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 411 –420, 1999.
5. J. Liu, S. Basu, and R. Lutz. Compositional model checking of software product lines using variation point obligations. *Automated Software Engineering*, 18, 2011.
6. F. Mazzanti. UMC logics. http://fmt.isti.cnr.it/umc/V4.1/umc.html. [Online; accessed July 9th, 2013].
7. R. Nicola and F. Vaandrager. Action versus state based logics for transition systems. In I. Guessarian, editor, *Semantics of Systems of Concurrent Processes*, volume 469 of *Lecture Notes in Computer Science*, pages 407–419. Springer-Verlag, 1990.
8. C. Prehofer. Assume-guarantee specifications of state transition diagrams for behavioral refinement. In *iFM 2013: 10th International Conference on integrated Formal Methods*. Springer-Verlag, June 2013.
9. M. H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. A state/event-based model-checking approach for the analysis of abstract system properties. *Science of Computer Programming*, 76(2):119 – 135, 2011.
10. G. Zhang and M. Hölzl. Hila: High-level aspects for uml state machines. In S. Ghosh, editor, *Models in Software Engineering*, volume 6002 of *Lecture Notes in Computer Science*, pages 104–118. Springer-Verlag, 2010.

# Global State Checker: Towards SAT-Based Reachability Analysis of Communicating State Machines*

Petra Kaufmann[1], Martin Kronegger[2], Andreas Pfandler[2], Martina Seidl[1,3], and Magdalena Widl[4]

[1] Business Informatics Group, TU Wien
[2] Database and Artificial Intelligence Group, TU Wien
[3] Institute for Formal Models and Verification, JKU Linz
[4] Knowledge-Based Systems Group, TU Wien
{firstname.lastname@tuwien.ac.at}

**Abstract.** We present a novel propositional encoding for the reachability problem of communicating state machines. The problem deals with the question whether there is a path to some combination of states in a state machine view starting from a given configuration. Reachability analysis finds its application in many verification scenarios. By using an encoding inspired by approaches to encode planning problems in artificial intelligence, we obtain a compact representation of the reachability problem in propositional logic. We present the formal framework for our encoding and a prototype implementation. A first case study underpins its effectiveness.

## 1 Introduction

In model-based engineering (MBE), software models take over the role as core development artifact, which textual code has in traditional software engineering. The goal of MBE is to leverage the abstraction power of models in order to deal with the complexity of modern software systems [2]. Executable code is to be directly generated from the models with little or no intervention of a developer [12]. With this valorization of software models, stronger requirements on their correctness come along. As a consequence, formal methods found their way into MBE to verify models, often by reusing techniques successfully applied for the verification of traditional software systems. Among the most successful techniques to verify hardware and software systems are approaches based on *model checking* [4], which exhaustively traverse the states of a system to answer questions related to the reachability of certain states from an initial configuration. In order to deal with large state spaces, symbolic methods have been introduced to compactly encode state spaces. Thereby propositional logic turned out to be particularly useful. The success of model checking is closely connected to the observation that in many cases it is sufficient to show correctness only for a restricted number of execution steps, which resulted in the method of bounded model checking [3].

In the context of MBE, model checking is used for the verification of behavioral models like UML state machines. For this purpose, many encodings of state machines have been proposed which translate the state machines to the input format of the model checkers. Usually,

* This work was partially funded by the Vienna Science and Technology Fund (WWTF) under grant ICT10-018 and by the Austrian Science Fund (FWF): P25518-N23.

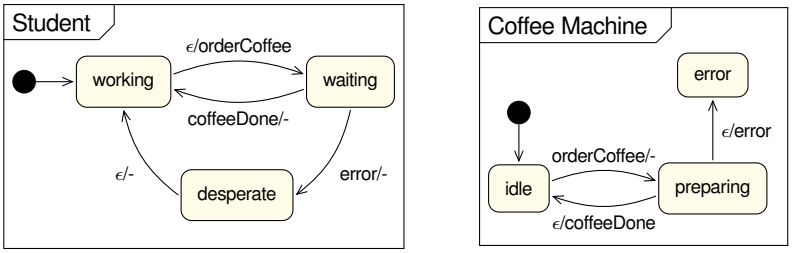


**Fig. 1.** State machines of a student and a coffee machine.

the model checkers provide languages to describe finite state automata, which are also the conceptual basis of state machines. However, it still can be challenging to find a semantics-preserving translation as even similar concepts may strongly diverge in their semantics. In this paper, we propose to directly encode the reachability problem for composite state machines into the problem of satisfiability of propositional logic (SAT). The motivation behind this approach is an integration into our formal MBE framework [15] where we successfully used SAT technologies in the context of optimistic model versioning. Our current encoding reuses ideas from SAT-based planning [11] to encode the search of paths between global states of a state machine view.

This paper is structured as follows. First, we review related work in Sec. 2. Then we provide a concise problem definition in Sec. 3 which serves as basis for our encoding presented in Sec. 4. In Sec. 5 we introduce our Eclipse-based implementation and report on a first case study. Finally, we conclude with an outlook to future work.

## 2 Related Work

Several works have been presented which deal with the transformation of UML state machines to input languages of model checkers (see for example in [1,5,6,8,10]). These languages provide high-level constructs to model software systems and in many aspects they provide similar constructs as do modeling languages like UML, in particular UML state machines.

However, one of the major challenges of such approaches is to overcome semantical heterogenities, which raises the question if this translation step is really required. This has already been recognized by Niewiadomski et al., who propose an encoding to propositional logic for bounded reachability analysis of state machines, which they show to be more efficient than translations to standard model checkers [9]. In this paper, we follow the approach of [9] to encode the reachability problem to SAT, but propose an alternative encoding where we formulate the reachability analysis problem of UML state machines inspired by encodings as used for solving planning problems [11]. As a result, we obtain an intuitive encoding which allows to directly extract a path of a length bounded by $k$ from a solution without the need of looping through values lower than the bound.

## 3 Problem Definition

To motivate our approach consider the following example. Fig. 1 shows the state machines of a student and a coffee machine implementing a simplified workflow of a student's interaction

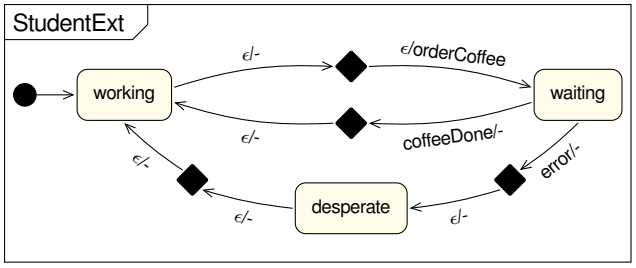


**Fig. 2.** Extended state machine of the student machine.

with a coffee machine. The initial state of each state machine is indicated by an arrow from a black filled circle. States are visualized as rounded rectangles and are connected to each other by transitions. Each transition carries a label consisting of a symbol called *trigger* to the left, and a set of symbols called *effects* to the right of a "/". The empty trigger $\epsilon$ indicates that the transition can be executed without receiving any trigger symbol. This symbol can be used to model an on-completion event. The symbol "–" represents an empty set of effects. The receipt of the trigger symbol causes the state machine to change its current state from the source state to the target state of the transition. The symbols in the set of effects are sent during the execution of the transition. The communication among state machines is synchronous and therefore the execution of a transition is only possible if each of its effects is understood by a different state machine in its current state.

For a given set of state machines, the *Global State Checking* (GSC) problem asks whether a certain combination of states of the state machines can be reached from an initial configuration. In this paper we consider the *k-Global State Checking (*$k$-GSC*) problem*. Given a set $\mathcal{M}$ of state machines which communicate over an alphabet $\mathcal{A}$, the $k$-GSC problem asks whether a certain combination of states of the state machines can be reached from an initial configuration by a path with a length of at most $k$. For example, it can be checked whether the combination of the states working in state machine student and preparing in state machine coffee machine is reachable starting from the initial states through a path of length 10, or whether the combination of waiting and error is not reachable by a path of length 1000. In the following, we present a precise definition of the semantics of a state machine view and of the $k$-GSC problem. We start by defining a state machine as follows:

**Definition 1 (State Machine).** *Given an alphabet $\mathcal{A}$, a* state machine *$M$ is a quintuple $(S,\iota,A^{tr},A^{eff},T)$, where $S$ is a set of* states*, $\iota \in S$ is a designated* initial state*, $A^{tr} \subseteq \mathcal{A}$, $A^{eff} \subseteq \mathcal{A}$, and $T \subseteq S \times A^{tr} \cup \{\epsilon\} \times \mathcal{P}(A^{eff}) \times S$.*

A state machine consists of a set of states, two alphabets, and a transition relation between the states. For a transition $t \in T$ with $t = (s,tr,eff,s')$, $s$ is the source state of the transition, $s'$ is the target state, $tr$ is a symbol (trigger) which upon receipt triggers the execution of the transition, and $eff$ is a set of symbols (effects) that are sent when the transition is executed. The trigger symbol can be the special symbol $\epsilon \notin \mathcal{A}$ standing for an empty trigger. A transition containing $\epsilon$ can be triggered no matter whether any symbol is received. In order for the execution of the transition to finish, each symbol in $eff$ must be received by a different state machine. In Fig. 1, state machine Student contains states $S = \{$working,desperate,waiting$\}$,

triggers $\mathcal{A}^{tr} = \{\mathsf{coffeeDone}, \mathsf{error}\}$ and effect $\mathcal{A}^{eff} = \{\mathsf{orderCoffee}\}$. An example for a transition is $(\mathsf{working}, \epsilon, \{\mathsf{orderCoffee}\}, \mathsf{waiting})$.

In order to give a precise semantics of the interaction between state machines, we introduce the notion of an *extended* state machine.

**Definition 2 (Extended State Machine).** *Given a state machine* $M = (S, \iota, A^{tr}, A^{eff}, T)$*, the* extended state machine $M^*$ *is a quintuple* $(S \cup S^*, \iota, A^{tr}, A^{eff}, T^*)$ *where* $S^* = \{s_t^* \mid t \in T\}$ *and* $T^* = \{(s, tr, \emptyset, s_t^*), (s_t^*, \epsilon, eff, s') \mid t = (s, tr, eff, s') \in T\}$.

An extended state machine introduces an *intermediate state* $s_t^*$ for each transition $t$. This intermediate state has one incoming transition, which is triggered by the trigger of $t$ and has no effects. It also has one outgoing transition, which leads to the target state of $t$ with $\epsilon$ as trigger and the effects of $t$. The states contained in $S^*$ we call *extended states*. Fig. 2 depicts the extended state machine constructed from state machine Student in Fig. 1. The reason for the construction of an extended state machine is to distinguish between the event of having received the trigger and the event of being able to send the effects.

The communication between state machines takes place through a structure called *message set*. A message set contains a sender state machine and a set of pairs, each containing a symbol sent by the sender state machine and a receiving state machine. Each of the symbols is sent by the same state machine but received by a different state machine. This is captured in the following definition.

**Definition 3 (Message Set).** *Given a set* $\mathcal{M} = \{M_1^*, ..., M_l^*\}$ *of extended state machines with* $M_i^* = (S_i, \iota_i, A_i^{tr}, A_i^{eff}, T_i)$ *for* $1 \leq i \leq l$*, a* message set *is a pair* $(\sigma, \{(a_1, R_1^*), ..., (a_k, R_k^*)\})$ *where*

- $\sigma = M_d^*$ *with* $1 \leq d \leq l$ *and*
- $\{(a_1, R_1^*), ..., (a_k, R_k^*)\} \in \mathcal{P}(A_d^{eff} \times \mathcal{M} \setminus \{\sigma\})$

*such that for* $1 \leq i \leq k$ *all* $R_i^*$ *are pairwise distinct.*

For a message set $(\sigma, \{(a_1, R_1^*), \ldots, (a_k, R_k^*)\})$, $\sigma$ is an extended state machine which executes a transition leaving an extended state with the set $\{a_1, ..., a_k\}$ of effects, and for each $1 \leq i \leq k$, $R_i^*$ is an extended state machine which executes a transition leaving an original (non-extended) state through trigger $a_i$. Note that $\{(a_1, R_1^*), ..., (a_k, R_k^*)\}$ can be the empty set, which represents an empty set of effects on a transition.

A message set can be *admissible* in some global configuration. Such a configuration is given by a *global state*, a tuple of states containing exactly one state per state machine. By applying a message, a *global successor state* is reached.

**Definition 4 (Application of a Message Set).** *Given a set of extended state machines* $\mathcal{M} = \{M_1^*, ..., M_l^*\}$ *with* $M_i^* = (S_i, \iota_i, A_i^{tr}, A_i^{eff}, T_i)$ *for* $1 \leq i \leq l$*, and a global state* $\hat{s} = (s_1, ..., s_l) \in S_1 \times \cdots \times S_l$*, a message set* $m = (M_d^*, \{(a_1, R_1^*), ..., (a_k, R_k^*)\})$*, with* $1 \leq d \leq l$ *and* $1 \leq k < l$*, is* admissible *in* $\hat{s}$ *if*

*(i)* $(s_d, \epsilon, \{a_1, ..., a_k\}, s_d') \in T_d$*, and*
*(ii) there exists a set* $\mathcal{R} \subseteq \{1, ..., l\} \setminus \{d\}$ *and a bijective function* $\mathsf{rec} : \{1, ..., k\} \to \mathcal{R}$ *such that* $R_j^* = M_{\mathsf{rec}(j)}^*$ *and* $(s_{\mathsf{rec}(j)}, a_j, \emptyset, s'_{\mathsf{rec}(j)}) \in T_{\mathsf{rec}(j)}$ *for each* $1 \leq j \leq k$.

*Given a global state $\hat{s}$ and a message set $m$ that is admissible in $\hat{s}$, a* global successor state *$\hat{s}'$ of $\hat{s}$* after applying *$m$ is given by $\hat{s}' = (\mathsf{next}(s_1),...,\mathsf{next}(s_l))$ where*

$$\mathsf{next}(s_i) = \begin{cases} s'_d & \text{if } i{=}d \\ s'_i & \text{if } i \in \mathcal{R} \\ s_i & \text{otherwise} \end{cases}$$

There are two requirements for a message set to be admissible in a global state: (1) the sender's state in the global state is an extended state with an outgoing transition containing the set $\{a_1,...,a_k\}$ of effects, and (2) each receiver's state in the global state has an outgoing transition triggered by the respective symbol from the message set. Note that we are dealing with extended state machines, which means that a transition cannot carry a trigger symbol other than $\epsilon$ together with a non-empty set of effects. Therefore it can never happen that a receiver state machine $R_i^*$ sends any effects while executing the transition triggered by some symbol $a_i$. The global successor state $\hat{s}'$ is reached by applying a message set. It differs from $\hat{s}$ in states of the sender and the receiver state machines contained in the applied message set: The sender's state changes from an extended state to its only successor state, and the receivers' states change according to the received symbol into an extended state.

We combine message sets of disjoint sets of state machines in a *transaction* as follows.

**Definition 5 (Transaction).** *A* transaction *is a nonempty set of message sets $\{m_1,...,m_l\}$ with $m_i = (\sigma_i,\{(a_{i,1},R^*_{i,1}),...,(a_{i,k_i},R^*_{i,k_i})\})$ such that all state machines occurring in the message sets, i. e., all $\sigma_i$ and $R^*_{i,j}$ (with $1 \leq i \leq l, 1 \leq j \leq k_i$), are pairwise distinct.*

A transaction is admissible if all its message sets are admissible. The global state reached by applying a transaction is the global state reached by applying each of its message sets.

We further define a path as a sequence of transactions as follows.

**Definition 6 (Path).** *A* path *$\mu$* from a global state *$\hat{s}_0$* to a global state *$\hat{s}_k$ is a sequence $\mu = [n_1,...,n_k]$ of transactions such that there exists a sequence $[\hat{s}_0,...,\hat{s}_k]$ of global states where for all $1 \leq i \leq k$, $n_i$ is admissible in state $\hat{s}_{i-1}$ and $\hat{s}_i$ is the global successor state of $\hat{s}_{i-1}$ after applying $n_i$.*

A global state *$\hat{s}_j$ is reachable from $\hat{s}_i$* if there is a path from $\hat{s}_i$ to $\hat{s}_j$. The *length* of a path is the number of its transactions.

The $k$-GSC problem deals with reaching a combination of states of state machines in a state machine view. This combination contains at most one state of each state machine. Hence such a combination not necessarily specifies a complete global state. We therefore define a partial global state as follows.

**Definition 7 (Partial Global State).** *Given a set $\mathcal{M} = \{M_1^*,...,M_l^*\}$ of extended state machines with $M_i^* = (S_i,\iota_i,A_i^{tr},A_i^{eff},T_i)$ for $1 \leq i \leq l$, a* partial global state *is an $l$-tuple $\hat{s}_p \in S_1 \cup \{?\} \times \cdots \times S_l \cup \{?\}$, where ? is a new symbol not contained in any $S_i$. $\hat{s}_p = (s_1,...,s_l)$* matches *a global state $\hat{s} = (q_1,...,q_l)$ if for all $1 \leq i \leq l$, $s_i = q_i$ whenever $s_i \neq ?$.*

Finally, we define the $k$-GSC problem as follows.

$k$-GSC
*Instance:* A set $\mathcal{M}$ of state machines, a global state $\hat{s}$, and a partial global state $\hat{s}_p$.
*Question:* Is there a path of length at most $k$ from $\hat{s}$ to a global state $\hat{s}'$ that matches $\hat{s}_p$?

The global state $\hat{s}$ is also referred to as *initial state* and the partial global state $\hat{s}_p$ is also referred to as *goal*. The initial state usually contains each state machine's initial state.

## 4 Encoding

In order to find solutions to the $k$-GSC problem, we propose to encode it to the satisfiability problem of propositional logic (SAT) and hand it to a SAT solver. In the following we present a detailed description of the SAT formula representing the $k$-GSC problem. In the next section we describe our tool, the Global State Checker, which builds upon this encoding.

Let $k$ be a positive integer, $\hat{s}$ a global state representing an initial state, and $\hat{s}_p$ a partial global state representing a goal. Then the formula $\varphi$ is satisfiable if and only if there is a global state $\hat{s}'$ that is reachable from $\hat{s}$ by a path of length at most $k$ and that matches $\hat{s}_p$.

Recall that in a $k$-GSC instance we are given a set $\mathcal{M} = \{M_1, \ldots, M_l\}$ of state machines, a global state $\hat{s} = (x_1, \ldots, x_l)$, and a partial state $\hat{s}_p = (g_1, \ldots, g_l)$. For each $1 \leq i \leq l$ let $M_i = (S_i, \iota_i, A_i^{tr}, A_i^{eff}, T_i)$ and the corresponding extended state machine be $M_i^* = (S_i \cup S_i^*, \iota_i, A_i^{tr}, A_i^{eff}, T_i^*)$.

In order to define the set of variables of $\varphi$, we introduce a set $\mathcal{T} \coloneqq \bigcup_{1\leq i\leq l} T_i$ of transitions, a set $\mathcal{A} \coloneqq \bigcup_{1\leq i\leq l} \left(A_i^{tr} \cup A_i^{eff}\right)$ of symbols, a set $\mathcal{S} \coloneqq \bigcup_{1\leq i\leq l} S_i$ of states, and a set $\mathcal{S}^* \coloneqq \bigcup_{1\leq i\leq l} S_i^*$ of extended states. Then the set of variables is given by $\{v^i \mid v \in (\mathcal{T} \cup \mathcal{A} \cup \mathcal{S} \cup \mathcal{S}^*), 0 \leq i \leq k\}$ were $i$ is an index capturing the relative position of the variable in the path. That is, each transition, symbol, state, and extended state together with one index up to $k$ represents a variable.

Let $t = (s, tr, \mathit{eff}, s')$ be a transition of a state machine and $(s, tr, \emptyset, s_t^*)$ and $(s_t^*, \epsilon, \mathit{eff}, s')$ be the corresponding transitions in the respective extended state machine. To simplify the presentation, we use the functions $\mathsf{src}(t) \coloneqq s$, $\mathsf{int}(t) \coloneqq s_t^*$, $\mathsf{trg}(t) \coloneqq tr$, $\mathsf{eff}(t) \coloneqq \mathit{eff}$, and $\mathsf{tgt}(t) \coloneqq s'$.

Further let $\mathcal{T}^* \coloneqq \bigcup_{1\leq i\leq l} T_i^*$. Given a state $s \in \mathcal{S}$, let $\mathsf{environ(s)} \coloneqq \{s^* \mid (s^*, \epsilon, \emptyset, s) \in \mathcal{T}^*\} \cup \{s^* \mid (s, \epsilon, \mathit{eff}, s^*) \in \mathcal{T}^*\}$ be a set of extended states containing a predecessor of $s$ if the transition does not contain any effects and a successor of $s$ if the transition contains $\epsilon$ as trigger.

The formula $\varphi$ is then given by a conjunction of the following subformulas:

$$\varphi_{\text{init}} \coloneqq \bigwedge_{i=1}^{l} \left( \bigwedge_{s\in\mathcal{S}_i, s=x_i} s^0 \wedge \bigwedge_{s\in\mathcal{S}_i\cup\mathcal{S}_i^*, s\neq x_i} \overline{s}^0 \wedge \bigwedge_{a\in\mathcal{A}} \overline{a}^0 \right)$$

$$\varphi_{\text{goal}} \coloneqq \bigwedge_{i=1, g_i\neq ?}^{l} \left( g_i^k \vee \bigvee_{s\in\mathsf{environ(g_i)}} s^k \right)$$

$$\varphi_1 \coloneqq \bigwedge_{i=0}^{k-1} \bigwedge_{t\in\mathcal{T}} \left[ t^i \rightarrow \left( \mathsf{src}(t)^i \wedge \mathsf{int}(t)^{i+1} \wedge \mathsf{trg}(t)^i \wedge \overline{\mathsf{trg}(t)}^{i+1} \wedge \bigwedge_{\mathit{eff}\in\mathsf{eff}(t)} \left( \overline{\mathit{eff}}^i \wedge \mathit{eff}^{i+1} \right) \right) \right]$$

$$\varphi_2 \coloneqq \bigwedge_{i=0}^{k-1} \bigwedge_{\mathit{trg}\in\mathcal{A}} \left[ \mathit{trg}^i \wedge \overline{\mathit{trg}}^{i+1} \rightarrow \bigvee_{t\in\mathcal{T}, \mathit{trg}=\mathsf{trg}(t)} t^i \right]$$

$$\varphi_3 \coloneqq \bigwedge_{i=0}^{k-1} \bigwedge_{\mathit{eff} \in \mathcal{A}} \left[ \overline{\mathit{eff}}^{\,i} \wedge \mathit{eff}^{i+1} \rightarrow \bigvee_{t \in \mathcal{T}, \mathit{eff} \in \mathsf{eff}(t)} t^i \right]$$

$$\varphi_4 \coloneqq \bigwedge_{i=0}^{k-1} \bigwedge_{s \in S} \left[ s^i \wedge \overline{s}^{i+1} \rightarrow \bigvee_{t \in \mathcal{T}, s = \mathsf{src}(t)} t^i \right]$$

$$\varphi_5 \coloneqq \bigwedge_{i=0}^{k-1} \bigwedge_{t \in \mathcal{T}} \left[ \mathsf{int}(t)^i \wedge \overline{\mathsf{int}(t)}^{i+1} \rightarrow \bigwedge_{\mathit{eff} \in \mathsf{eff}(t)} \overline{\mathit{eff}}^{i+1} \right]$$

$$\varphi_6 \coloneqq \bigwedge_{i=0}^{k-1} \bigwedge_{t \in \mathcal{T}} \left[ \mathsf{int}(t)^i \wedge \mathsf{int}(t)^{i+1} \rightarrow \bigwedge_{\mathit{eff} \in \mathsf{eff}(t)} \mathit{eff}^{i+1} \right]$$

$$\varphi_7 \coloneqq \bigwedge_{i=0}^{k-1} \bigwedge_{t \in \mathcal{T}} \left[ \left( \mathsf{int}(t)^i \wedge \bigwedge_{\mathit{eff} \in \mathsf{eff}(t)} \overline{\mathit{eff}}^{i+1} \right) \rightarrow \left( \overline{\mathsf{int}(t)}^{i+1} \wedge \mathsf{tgt}(t)^{i+1} \right) \right]$$

$$\varphi_8 \coloneqq \bigwedge_{i=0}^{k-1} \bigwedge_{j=1}^{l} \left[ \left( \bigvee_{s \in (S_j \cup S_j^*)} s^i \right) \wedge \bigwedge_{s_1, s_2 \in (S_j \cup S_j^*), s_1 \neq s_2} \left( \overline{s_1}^{\,i} \vee \overline{s_2}^{\,i} \right) \right].$$

The intuition behind these subformulas is as follows: $\varphi_{init}$ initialises the path by setting the initial states with index 0 to true, and all other states and all symbols to false. $\varphi_{goal}$ encodes the goal states and the extended states in their environment for index $k$. For all other path indices, a symbol variable in its positive polarity means that the respective symbol has been made available as effect through the transaction at the respective index and is waiting to be consumed by some transition as a trigger in a later transaction. When the symbol has been consumed, the respective symbol variable occurs in its negative polarity. $\varphi_1$ ensures that whenever a transition is executed, the state machine changes to the respective extended state. Then the trigger symbol is set its negative polarity and the effect symbols are set to their positive polarity. $\varphi_2$ and $\varphi_3$ express that whenever the polarity of a symbol is changed, there must be a transition causing this change. $\varphi_4$ encodes that leaving a state is always caused by some transition. $\varphi_5$ and $\varphi_6$ ensure that after executing a transaction either all effect symbols of a transition are consumed or none of them. The formulas $\varphi_j$ with $j \in \{2,...,5\}$ are also called *framing axioms*. These formulas ensure that every change in a global state has a cause. Note that it is not necessary to state all changes explicitly, because they are already covered implicitly by other formulas. $\varphi_7$ forces a machine to move to the target state if all effect symbols have been consumed. $\varphi_8$ expresses that each machine is in exactly one state after each transaction.

Note that the encoding allows that nothing happens, i. e., no transaction takes place at an index. It is ensured by the framing axioms that in this case, the global state remains the same. This relaxation implicitly encodes the "at most $k$" formulation of the problem: If at $n$ indices nothing happens and the goal is reached at index $k$, it means that the length of the transaction sequence is $k-n$. The framing axioms also ensure that state machines not participating in a transaction do not change.

A solution returned by the SAT solver consists of a set of variables set to *true*. By extracting those variables that represent states and transitions (sets $\mathcal{S}$, $\mathcal{S}^*$, and $\mathcal{T}$) we obtain a path to the goal. If the length of the path is less than $k$, then for some consecutive indices the state variables represent the same states. The shorter path can therefore be easily extracted.

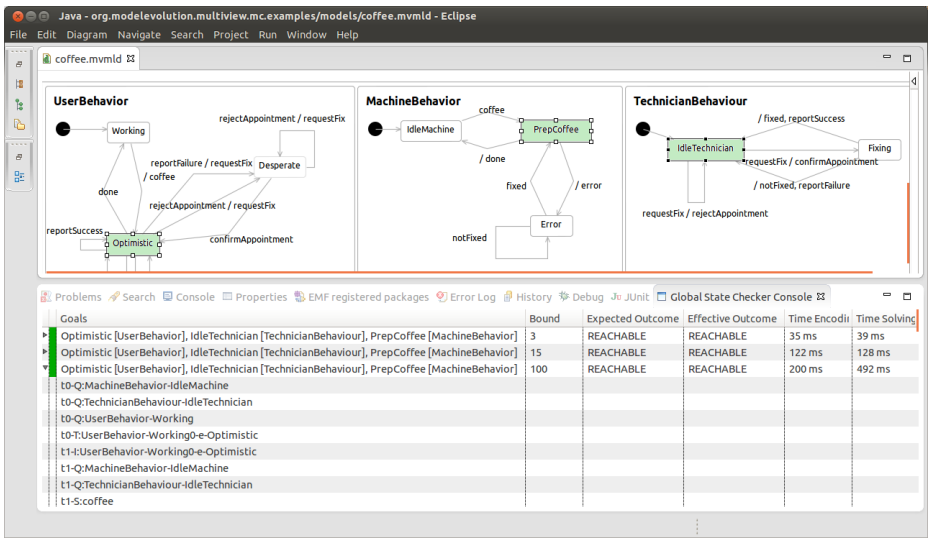


**Fig. 3.** UI of the Global State Checker.

In order to simplify the encoding we assume that after applying a transaction each symbol can be consumable only once. Allowing a symbol to be consumable multiple times after one transaction requires the integration of counters, which can be realized, e.g., by building upon ideas presented in [13]. We are currently developing such an extension.

## 5 Implementation and Case Study

We have implemented the encoding described above as Eclipse plugin[5] and designed a set of instances for an initial case study. Our prototype implementation is available on our Eclipse update site[6]. Further resources and instructions can be found on our project website[7]. In the following, we shortly describe our implementation and discuss a first case study.

*Implementation.* The *Global State Checker* prototype is embedded into the Eclipse modeling framework (EMF). The metamodel for the statemachine view described in Sec. 3 is provided by an Ecore metamodel. Instances of the $k$-GSC problem are created as models of the defined language. Our *encoder* module takes a state machine view as input and encodes it into a formula of propositional logic according to the encoding described in Sec. 4. Additional measures are taken to convert the formula into conjunctive normal form (CNF), which is a common format used by SAT solvers. The data structure representing the encoding is handed to the SAT4J [7] solver, a SAT solver written in Java, which integrates seamlessly into our tool. The SAT solver either returns `UNSAT` or `SAT`. The former means that the problem has no solution, i.e., that the specified state is not reachable in $k$ steps. The latter case means that there exists a solution, i.e., a path from the initial configuration to a global state matching

[5] `http://www.eclipse.org/`
[6] `http://modelevolution.org/updatesite`
[7] `http://modelevolution.org/prototypes/gsc`

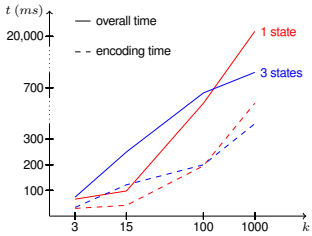


**Fig. 4.** Runtime measures for the coffee machine test case.

the specified goal. In this case, the SAT solver additionally returns a logical model of the formula representing the problem. A logical model assigns one of the truth values *true* and *false* to each propositional variable. Since each variable represents either a state, a transition, or a symbol with respect to a certain index, such a model can be easily translated into a path leading to the specified state. This way we retrieve a solution to our original problem.

Fig. 3 shows the graphical user interface of our prototype. The user can select a set of states directly in the modeling editor, enter a bound, and start the $k$-GSC tool. For convenience, it may be specified whether the given global state is expected to be reachable or not. Red or green highlighting indicate how the expected result compares to the effective result. All test cases are listed in the global state checker console. The expanded subitems of the third test case in Fig. 3 show the path found by the SAT solver.

*Case Study.* We have designed three different benchmark instances with varying number of state machines, states, and transitions in order to test our prototype. The instances have been adapted from our previous work [15]. The first instance is shown in Fig. 3 and represents the communication of a coffee machine, a PhD student, and a technician who repairs the coffee machine in case of an error. The second instance represents a simplified version of the SMTP protocol. The third instance represents the well-known dining philosophers problem with three philosophers. Of each instance we have created a correct and an erroneous version.

For all test cases, $\hat{s}$ (the starting state of the path) has been set to the global initial state, i.e., the global state where each state machine is in its initial state. For instances "coffee" and "mail", each possible combination of states for each state machine, and for the instance "philosophers" a meaningful selection of combination of states have been tested with the bound $k$ set to $\{3,15,100\}$. Details on the outcomes of the test cases are presented on our project website. The results of all test cases are as expected. The bugs in the erroneous versions have been found. The approach performs well up to a bound of $k=1000$. In Fig. 4 we exemplarily show the runtime measures for the coffee machine instance, which were evaluated on an Intel Core i5 CPU with 2.5 GHz running Linux. The red lines show a test case in which the goal was a partial global state containing a state of one out of three state machines, and the blue lines show a case where the goal was a complete global state, i.e., containing a state of each state machine. The dashed lines express the time needed to encode the problem and the solid lines give the total time, i.e., encoding time plus time spent by the SAT solver. As can be expected, the bottleneck for higher bounds is the task of solving the SAT instance.

## 6 Conclusion and Future Work

We presented a SAT-based approach to verify whether a combination of states in the state machine view of a software model can be reached through a path of bounded length from an

initial configuration. Using SAT allowed for a more direct translation than, e.g., using a model checker, and for a good integration within our existing framework which is implemented as plugin for the popular development environment Eclipse and therefore easily accessible to developers. We precisely defined the semantics of the state machine view of a software model. On this formal semantics we could build a formal problem definition. Based on this problem definition, the encoding to SAT turned out to be very intuitive, easy to understand and therefore easy to extend.

Immediately planned extensions of our encoding include the integration of a counter to handle the availability of multiple occurrences of a symbol representing an effect. Another important task is the optimization of the encoding in order to avoid unnecessary blowups and yield a more compact representation. On a longer time frame, the integration of transition guards and hierarchical state machines are planned. Before extending our approach, however, more extensive testing as well as a runtime comparison to alternative encodings will be necessary. To thoroughly test the tool, we will implement a solution verifier to automatically verify the solution returned by the SAT solver and apply our previously used approach of grammar-based fuzzing for MBE tools [14].

## References


1. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: A State/Event-based Model-Checking Approach for the Analysis of Abstract System Properties. Science of Computer Programming 76(2), 119–135 (Feb 2011)
2. Bézivin, J.: On the Unification Power of Models. SoSyM 4(2), 171–188 (2005)
3. Biere, A.: Bounded Model Checking. In: Handbook of Satisfiability, pp. 457–481. IOS Press (2009)
4. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. The MIT Press (1999)
5. Dubrovin, J., Junttila, T.A.: Symbolic Model Checking of Hierarchical UML State Machines. In: Int. Conf. on Application of Concurrency to System Design. pp. 108–117. IEEE (2008)
6. Knapp, A., Merz, S., Rauh, C.: Model Checking—Timed UML State Machines and Collaborations. In: Formal Techniques in Real-Time and Fault-Tolerant Systems. pp. 395–416 (2002)
7. Le Berre, D., Parrain, A.: The Sat4j Library, Release 2.2, System Description. Journal on Satisfiability, Boolean Modeling and Computation 7, 59–64 (2010)
8. Lilius, J., Paltor, I.: vUML: A Tool for Verifying UML Models. In: ASE. pp. 255–258 (1999)
9. Niewiadomski, A., Penczek, W., Szreter, M.: Towards Checking Parametric Reachability for UML State Machines. In: Ershov Memorial Conference. LNCS, vol. 5947. Springer (2010)
10. Ober, I., Graf, S., Ober, I.: Validation of UML Models via a Mapping to Communicating Extended Timed Automata. In: Model Checking Software, LNCS, vol. 2989, pp. 127–145. Springer (2004)
11. Rintanen, J.: Planning and SAT. In: Handbook of Satisfiability, pp. 483–504. IOS Press (2009)
12. Selic, B.: What Will it Take? A View on Adoption of Model-based Methods in Practice. SoSyM 11, 513–526 (2012)
13. Sinz, C.: Towards an Optimal CNF Encoding of Boolean Cardinality Constraints. In: Int. Conf. on Principles and Practice of Constraint Programming. LNCS, vol. 3709, pp. 827–831. Springer (2005)
14. Widl, M.: Test Case Generation by Grammar-based Fuzzing for Model-driven Engineering. In: Int. Haifa Verification Conference (2012)
15. Widl, M., Biere, A., Brosch, P., Egly, U., Heule, M., Kappel, G., Seidl, M., Tompits, H.: Guided Merging of Sequence Diagrams. In: Software Language Engineering. LNCS, vol. 7745, pp. 164–183. Springer (2013)

# Applying Model Transformation and Event-B for Specifying an Industrial DSL

Ulyana Tikhonova, Maarten Manders, Mark van den Brand,
Suzana Andova, and Tom Verhoeff

Technische Universiteit Eindhoven,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{u.tikhonova,m.w.manders,m.g.j.v.d.brand,s.andova,t.verhoeff}@tue.nl

**Abstract.** In this paper we describe our experience in applying the Event-B formalism for specifying the dynamic semantics of a real-life industrial DSL. The main objective of this work is to enable the industrial use of the broad spectrum of specification analysis tools that support Event-B. To leverage the usage of Event-B and its analysis techniques we developed model transformations, that allowed for automatic generation of Event-B specifications of the DSL programs. The model transformations implement a modular approach for specifying the semantics of the DSL and, therefore, improve scalability of the specifications and the reuse of their verification.

**Key words:** domain specific language, Event-B, model transformations, verification and validation, reuse, scalability

## 1 Introduction

Domain-Specific Languages (DSLs) are a central concept of Model Driven Engineering (MDE). A DSL provides domain notions and notation for defining models. It implements the semantic mapping of the models by means of model transformations. A DSL bridges the gap between the domain level and an execution platform. From a semantics point of view this gap can be quite wide, i.e. the DSL implementation usually includes rather complicated design solutions and algorithms. To manage the complexity of the industrial DSL, considered in this paper, we provide an explicit definition of its semantics by means of a formal method. This allows for formal specification of the DSL semantics and for assessing correctness of the specified semantic mapping via verification and validation.

In this paper we discuss the use cases of verification and validation applied to a DSL specification in an industrial context. We identify two different roles, that use different types of analysis of the DSL specification. A DSL developer is interested in validating and checking consistency of the DSL design and implementation. A DSL user is interested in getting better understanding of the DSL semantics, for example via simulation of its specifications. Correspondingly, in

the context of MDE a formal specification of a DSL can be given on two abstraction levels: the DSL metamodel level and the DSL model level.

There exists quite a number of formalisms for specifying behavior and tools for analyzing these specifications using different verification and validation techniques. In this research we use the Event-B formalism [2] and the Rodin platform [3], as they allow the implementation of all use cases listed above. By using Event-B and Rodin we can (1) prove consistency of the DSL semantics specifications with (automatic and interactive) provers, (2) find deadlocks and termination problems using model checkers, (3) use animators to validate the specified semantics with the help of domain experts, and (4) provide graphical visualization of the specification to help DSL users to understand how their programs run. All these tools are available in Rodin for Event-B.

In this paper, we show how Event-B and Rodin can be adopted in practice and applied to the industrial use cases, through model transformations from the DSL to Event-B. Our model transformations can automatically generate an Event-B specification for each concrete DSL program. For this, we apply the techniques of *composition* and *instantiation* of Event-B specifications. Composition of Event-B specifications simplifies the creation, maintenance and verification of larger specifications, because one can handle the smaller components separately. Instantiation is a way to concretize a generic Event-B specification, defined on the DSL metamodel level, to the model level of a concrete DSL program. The instantiation technique allows for the reuse of verification results from one level to the other. As a result of applying these techniques, our model transformations improve usability of Event-B and Rodin.

In the rest of the paper, Section 2 gives the overview of the industrial DSL and defines roles and use cases; Section 3 introduces the Event-B formalism; Sections 4.1 and 4.2 describe the decomposition and instantiation techniques; Section 4.3 outlines the implementation of our approach and results of its application. Related work is discussed in Section 5. Conclusions and directions for future work are given in Section 6.

## 2 Case Study and Use Cases

Our case study was performed at ASML,[1] a producer of complex lithography machines for the semiconductor industry. In our case study we specified the dynamic semantics of the LACE DSL. LACE (Logical Action Component Environment) is one of the DSLs, developed by and used within ASML for controlling lithography machines. A lithography machine consists of many physical *subsystems* (such as actuators, projectors, and sensors), which operate simultaneously in order to perform the required functions of the machine. LACE allows for specifying how subsystems operate in collaboration with each other by means of so-called *logical actions*. An example of a logical action is shown in Figure 1.

LACE has a graphical notation based on UML activity diagrams. A logical action consists of *subsystem actions* (rounded rectangles in Figure 1), each of

[1] `www.asml.com`, `http://en.wikipedia.org/wiki/ASML_Holding`

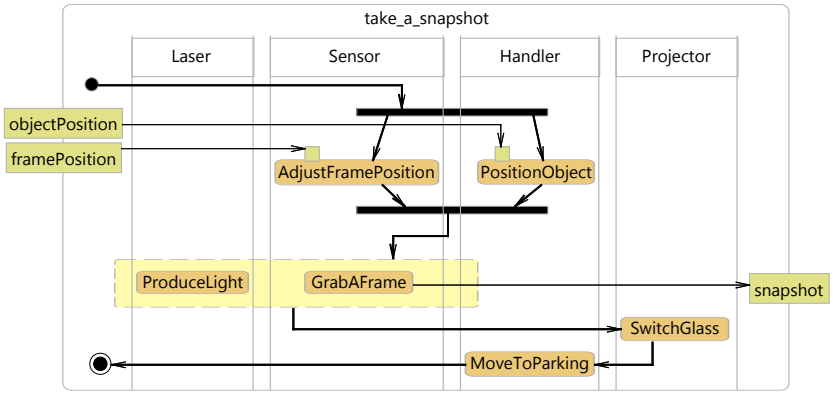


**Fig. 1: An example of a logical action for taking a snapshot**

which belongs to a subsystem, that executes this subsystem action (vertical column, containing the rounded rectangle). Subsystem actions, combined together into a so-called *scan* (dashed rounded rectangle), are executed synchronously. Thus, in Figure 1 the `Sensor` subsystem starts and stops the `GrabAFrame` action at the same moments, as the `Laser` subsystem starts and stops the `ProduceLight` action. Subsystem actions within a logical action can be executed *sequentially or concurrently* (thick arrows, fork and join nodes). For example, the subsystem actions `AdjustFramePosition` and `PositionObject` are independent actions and can be executed in any order or in parallel, but the `GrabAFrame` action can be performed only after both these actions are finished. Finally, subsystem actions may require and produce *data*. The dataflow in a logical action is depicted by means of thin arrows, input and output pins. For example, in Figure 1 the `GrabAFrame` action produces data, which is saved in the `snapshot` output parameter.

The high-level description of the machine subsystems' behavior, given in logical actions, is translated into the invocations of hardware drivers and a synchronization driver in such a way that the resulting execution matches the behavior specified in the logical actions. The semantic gap between LACE and driver functions is wide, and thus the translation is hard to develop, maintain, understand, and use. We construct a formal specification of LACE and apply different kinds of analysis to it, in order to enhance understandability, maintainability and usage of the DSL translation. We identify the following roles and use cases of specification of a DSL and its analysis in the industrial context (Figure 2).

A *DSL developer* designs and develops the DSL by constructing its metamodel and semantic translations. Formal specification of this DSL implementation allows for the verification, such as checking that it is consistent, non-contradictive, feasible and complete. A *DSL user* specifies DSL programs as instances of the DSL metamodel. In the context of formal methods the instantiation of metamodel by a DSL program needs to be verified separately. Formal specification of the DSL programs allows for the execution of specifications, and

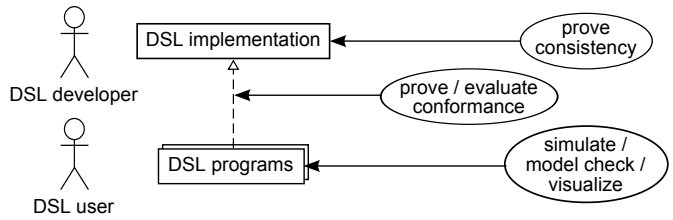


**Fig. 2: Roles and use cases of the DSL specification and its analysis**

thus for model checking and simulation. The construction of the LACE specifications and the implementation of the listed use cases are described in Section 4.

## 3 Event-B

Event-B is an evolution of the B method, both introduced by Abrial [2]. Event-B employs set theory and first-order logic for specifying software and/or hardware behavior. A big advantage of Event-B is its tool support, offered by the Rodin platform [3]. Using Rodin and its plug-ins, one can create and edit Event-B specifications, verify them using automatic or interactive provers, animate and model check Event-B specifications.

An Event-B specification consists of *contexts* and *machines*. A context describes the static part of a system: *sets*, *constants* and *axioms*. A machine uses (*sees*) the context to specify behavior of a system via a state-based formalism. *Variables* of the machine define the state space. *Events*, which change values of these variables, define transitions between the states. An event consists of *guards* and *actions*, and can have *parameters*. An event can occur only when its guards are true, and as a result of the event its actions are executed. The properties of the system are specified as *invariants*, which should hold for all reachable states. The properties are verified via proving automatically generated *proof obligations* and/or via model checking.

The attractive simplicity of Event-B is enhanced by techniques such as *shared event composition* and *generic instantiation*, which support scalability and reuse of Event-B specifications [4]. We discuss these techniques in detail in Section 4.

## 4 Model Transformations from LACE to Event-B

The Rodin platform allows for the implementation of the use cases described in Figure 2, provided that the corresponding Event-B specifications of LACE are given. This poses the following problems. First, the semantics of LACE is complex, therefore capturing it within Event-B machines is challenging and results in a big specification, which is hard to understand, maintain and verify. To tackle this problem we apply two types of composition of Event-B specifications: *composition of semantic features* and *composition of machines* (Section 4.2). The

LACE specification is composed using model transformations. Second, while a specification of LACE on the metamodel level can be created and analyzed once, specifications of the LACE programs need to be constructed and analyzed many times by DSL users. We cannot expect DSL users to create Event-B specifications of their LACE programs and to verify them themselves. Therefore, we apply model transformations from LACE to Event-B to automatically generate specifications of LACE programs and we use the *generic instantiation* technique to verify their conformance to the LACE specification, given on the metamodel level (Section 4.1). Moreover, we enhance simulation of Event-B specifications of LACE programs by providing a user-friendly visualization.

### 4.1 Instantiation of Event-B specification

Generic instantiation is a technique, proposed by Abrial and Hallerstede to reuse an existing Event-B specification by refining the data structures, specified in its constants and variables, in a new copy of this specification [4]. We apply generic instantiation as depicted in Figure 3 (on the left). The concepts of `conceptual machine` and of `composite machine` are introduced in Subsection 4.2.

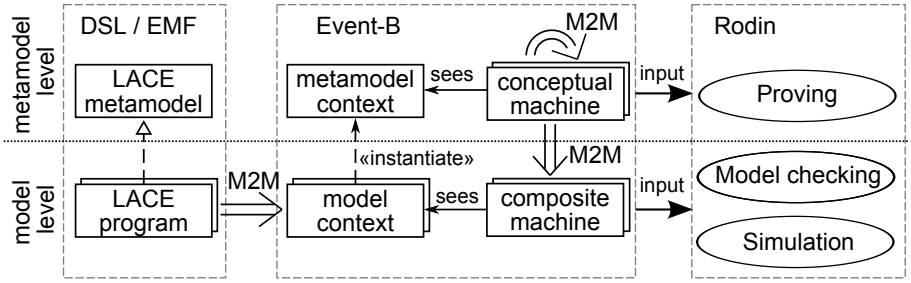


**Fig. 3: Instantiation and composition of Event-B specification**

The `metamodel context` captures the structure specified in the LACE metamodel. A `conceptual machine` uses this context to specify the dynamic semantics of LACE in terms of the metamodel. Based on the structural properties, specified in the axioms of the `metamodel context`, the `conceptual machine`s are proved to be consistent and complete by discharging the corresponding proof obligations using the Rodin provers. Thus, the semantics is verified on the metamodel level. The `metamodel context` and the `conceptual machine`s for a specific DSL are constructed manually and only once.

In the `model context`, values are assigned to the sets and constants, introduced in the `metamodel context`. The assignments are done in the axioms of the `model context`. Therefore, being used by a `composite machine`, this context specifies behavior of a concrete LACE program – on the model level. This specification can be model checked and animated, allowing for the analysis of a

particular LACE program. `Model contexts` are generated from LACE programs automatically by means of model transformations.

According to the generic instantiation technique, if all structural properties, defined in the `metamodel context`, can be derived for the structure, instantiated in the `model context`, then the verification of the Event-B specification can be extended straightforwardly from the metamodel level to the model level [4]. In [13] and [6] it is proposed to use theorem proving to show this derivation.

Due to the large sizes of the `model contexts`, generated from LACE programs, the automatic provers of Rodin fail to discharge instantiation theorems. On the other hand, we do not expect an average DSL user to prove these theorems using Rodin interactive provers, as it requires knowledge of propositional calculus and understanding of proof strategies. Therefore, instead of the theorem proving, we employ evaluation of structural properties predicates in the ProB animator integrated in Rodin [11]. Thus, we achieve automatic proof of instantiation in Event-B.

### 4.2 Composition of Event-B specification

As we mentioned before, capturing semantics of LACE within Event-B machines is rather complicated due to their different abstraction levels. To handle this complexity we employ modularity of LACE semantics. Each module is described separately as a *conceptual machine* in Event-B. Composition of the conceptual machines gives a resulting Event-B machine, which specifies the LACE dynamic semantics. The modular approach facilitates development, understandability and proving the correctness of the specification. We distinguish two types of modularity in the dynamic semantics of LACE: *semantic modularity* and *architectural modularity*.

To manage complexity of the LACE semantics we decompose it into separate *semantic features* (SFs): `Core SF`, `Order SF`, `Scan SF` and `Data SF`. The `Core SF` specification defines common concepts and interfaces: logical actions, consisting of subsystem actions, subsystems and events for requesting execution of logical actions and subsystem actions. `Order SF`, `Scan SF` and `Dataflow SF` are specified independently on the basis of the `Core SF` machine by adding extra variables, invariants, parameters, guards and actions to the `Core SF` machine and by changing some of the Event-B types, used in it. `Order SF` introduces partial order of execution of subsystem actions within a logical action. `Scan SF` joins subsystem actions into scans. `Data SF` introduces input and output parameters and dataflow within a logical action. The composition of semantic features is implemented via weaving Event-B code of the machines in the model transformation (the self-referential `M2M` arrow in Figure 3).

The LACE implementation consists of different software components, such as: logical action components (`LAC`), that translate logical action requests into subsystem actions, and subsystems (`SS`), that execute subsystem actions. This architectural modularity of LACE is implemented in Event-B using the *shared event composition* approach [14]. Software modules are specified in separate machines, which are then composed into one `composite machine` specifying the

whole system. The interaction of the modules is implemented via composition (or in fact, synchronization) of the events of the composing machines. Composition of events means conjunction of the events' guards and composition of the events' actions in one composite event. The composition of the `LAC` and `SS` machines is implemented using model transformation (the `M2M` arrow from `conceptual machine`s to `composite machine`s in Figure 3).

As a result of the intersection of two types of specification modularity, eight conceptual machines and four composition schemes need to be specified: for each semantic feature we specify a conceptual machine of each software module (`LAC` and `SS`) and a scheme of the interaction of `LAC` with `SS`. An Event-B machine that specifies the LACE semantics as a whole is composed of `LAC` and `SS` machines, that include Event-B code for all four semantic features, according to the compositional schemes of all four semantic features. Two dimensions of the modularity presented above simplify creation, verification and validation of Event-B components and maintenance of the model transformations.

### 4.3 Implementation

The LACE-to-Event-B transformations, described in Sections 4.1 and 4.2, were implemented using the *Operational QVT* (Query/View/Transformation) language [1] in the Eclipse environment. The input for the transformation is provided directly by the LACE implementation software, which employs model transformation and code generation techniques in the Borland Together environment, and therefore is compatible with EMF (Eclipse Modeling Framework). As a target metamodel for the transformation we use the Event-B Ecore implementation provided by the EMF framework for Event-B [15].

The LACE-to-EventB transformation is designed in a modular way, which follows the logic of instantiation and composition techniques as described in Sections 4.1 and 4.2. Thus, the transformation is possible to reuse and to generalize.

**Table 1: Characteristics of the LACE-to-Event-B transformation**

| Event-B components | Semantic features | | | | core+scan+order+data |
|---|---|---|---|---|---|
| | core | scan | order | data | |
| Metamodel context | 3 constants<br>5 axioms | 4 constants<br>8 axioms | 5 constants<br>9 axioms | 11 constants<br>22 axioms | – |
| Model context | 20 constants<br>7 axioms | 21 constants<br>8 axioms | 23 constants<br>10 axioms | 37 constants<br>17 axioms | – |
| LAC machine | 3 events<br>21 POs | 3 events<br>23 POs | 4 events<br>26 POs | 4 events<br>28 POs | 4 events<br>34 POs |
| SS machine | 3 events<br>7 POs | 3 events<br>11 POs | 3 events<br>7 POs | 3 events<br>7 POs | 3 events<br>11 POs |
| composition of LAC and SS machines | 10 events<br>70 POs | 30 events<br>386 POs | 10 events<br>82 POs | 10 events<br>89 POs | 30 events<br>491 POs |

Table 1 shows the representative characteristics of the transformation: sizes of the metamodel contexts vs. model contexts and sizes of the conceptual machines (`LAC` and `SS` machines for `core`, `scan`, `order` and `data` semantic features) vs. composite machines (bottom row). The automatically generated Event-B components are shaded. As an input for the transformation the LACE program depicted in Figure 1 is used. All proof obligations (POs) of the LAC and SS machines are discharged by invocation of the automatic provers in Rodin. The proof obligations of the composite machines (bottom row) can be left undischarged, as these are inherited proof obligations of the LAC and SS machines (according to the shared event composition approach [14]). The Event-B machine that specifies the LACE semantics as a whole is located in the bottom right cell of the table. One can observe that this machine is much larger and has much more proof obligations, than the conceptual machines, of which this machine is composed.

To make it convenient for a LACE user to work with Event-B we developed a graphical visualization of the LACE specification using the BMotion Studio plug-in [10]. This visualization runs together with the ProB animator and provides a GUI (graphical user interface) for a machine being animated. The GUI is based on the original LACE notation. By experimenting with a LACE program specification using this GUI a user can get better understanding of the DSL design and improve efficiency of her programs. Screen shots of the visualization can be found on the web-page of our project.[2]

## 5 Related Work

There are a number of studies in which Event-B has been applied to a specification of the dynamic semantics of a DSL. Aït-Sadoune and Aït-Ameur employ Event-B and Rodin for proving properties and animation of BPEL processes [5]. Hoang et al. use Event-B and Rodin to automate analysis of Shadow models [9]. In both studies, DSL program descriptions are translated into Event-B specifications. The translations are implemented in the Java programming language. These works do not use generic instantiation and composition techniques, but apply *refinement* of Event-B machines [4] to implement modularity of the programs. Based on our experience, refinement restricts semantics definition and can be rather complicated for automatic proving. Moreover, we use model transformations to implement the generation of Event-B specifications, which increases the abstraction level of the translation and therefore enhances its reuse.

Besides Event-B, other formalisms have been used as a target formal domain for specifying semantics of DSLs. Chen et al. propose transformational specification of dynamic semantics using Abstract State Machines (ASM) as a target formalism and explore specified behavior by means of the AsmL simulator tool [8]. Moreover, they introduce *semantic units* as an intermediate common language for defining dynamic semantics of DSLs and explore a technique for their composition [7]. Another approach, that supports reuse of the DSL analyses via intermediate specification modules, is proposed by Ratiu et al. [12]. They

[2] `www.win.tue.nl/mdse/COREF`

identify conceptually distinct *sub-languages*, shared by different DSLs, and transform these to different analysis formalisms. These works support modularity of a DSL specification by modularizing target formalisms. In this paper we describe how modularity of the DSL specification arises from the modularity of the DSL semantics, and apply model transformations to compose semantic modules.

## 6 Conclusion

In this paper we showed how the dynamic semantics of an industrial DSL can be defined using the Event-B formalism and model transformations. The Rodin platform and its plug-ins provide a broad spectrum of functionality and analysis tools for Event-B specifications. Our objective was to adopt Event-B for the industrial use cases for two major roles: DSL users and DSL developers. This was achieved by using MDE techniques – model transformations that define the semantics mapping from DSL domain to Event-B.

In order to specify semantics of LACE in a modular and scalable way we introduce semantic features and specify them in conceptual Event-B machines. The conceptual machines are verified on the metamodel level using automatic provers of Rodin. The LACE-to-Event-B transformation composes the conceptual machines into the LACE specification and instantiates this specification for concrete LACE programs. The resulting Event-B specifications can be validated and model checked by DSL developers and can be simulated by DSL users in the user-friendly GUI – all in the Rodin environment.

As future work we aim for applying the demonstrated techniques to other DSLs. For this we need to generalize LACE-to-Event-B transformation by distinguishing repetitive Event-B code, that can be combined into fine-grained specification patterns. This will allow not only for reuse of the demonstrated techniques of instantiation and composition, but also for reuse of already verified and visualized pieces of specification.

## 7 Acknowledgements

We are very grateful to Marc Hamilton and Wilbert Alberts (ASML, The Netherlands) for introducing us to the LACE world and providing very useful feedback on our experiments. We would like to thank Michael Butler and Colin Snook (University of Southampton, United Kingdom) for their help with using Event-B and Rodin. We also would like to thank Anton Wijs and Alexander Serebrenik (Eindhoven University of Technology, The Netherlands) for their useful comments on this paper.

## References

1. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. Object Management Group (OMG), July 2007.

2. J.-R. Abrial. *Modeling in Event-B: system and software engineering*, volume 1. Cambridge University Press, 2010.
3. J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: An Open Toolset for Modelling and Reasoning in Event-B. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(6):447–466, 2010.
4. J.-R. Abrial and S. Hallerstede. Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. *Fundam. Inform.*, 77(1-2):1–28, 2007.
5. I. Aït-Sadoune and Y. Aït-Ameur. Stepwise Design of BPEL Web Services Compositions: An Event-B Refinement Based Approach. In R. Lee, O. Ormandjieva, A. Abran, and C. Constantinides, editors, *Software Engineering Research, Management and Applications*, pages 51–68. Springer Berlin / Heidelberg, 2010.
6. D. A. Basin, A. Fürst, T. S. Hoang, K. Miyazaki, and N. Sato. Abstract Data Types in Event-B – An Application of Generic Instantiation. In *Workshop on the experience of and advances in developing dependable systems in Event-B*, volume abs/1210.7283 of *CoRR*, pages 5–16, 2012.
7. K. Chen, J. Porter, J. Sztipanovits, and S. Neema. Compositional Specification Of Behavioral Semantics For Domain-Specific Modeling Languages. *Int. J. Semantic Computing*, 3:31–56, 2009.
8. K. Chen, J. Sztipanovits, S. Abdelwalhed, and E. Jackson. Semantic Anchoring with Model Transformations. *European Conference on Model Driven Architecture - Foundations and Applications*, pages 115–129, 2005.
9. T. S. Hoang, A. McIver, L. Meinicke, C. Morgan, A. Sloane, and E. Susatyo. Abstractions of non-interference security: probabilistic versus possibilistic. *Formal Aspects of Computing*, pages 1–26, 2012.
10. L. Ladenberger, J. Bendisposto, and M. Leuschel. Visualising Event-B Models with B-Motion Studio. In M. Alpuente, B. Cook, and C. Joubert, editors, *Proceedings of FMICS 2009*, volume 5825 of *Lecture Notes in Computer Science*, pages 202–204. Springer, 2009.
11. M. Leuschel and M. Butler. ProB: A Model Checker for B. In A. Keijiro, S. Gnesi, and M. Dino, editors, *FME*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874. Springer-Verlag, 2003.
12. D. Ratiu, M. Voelter, Z. Molotnikov, and B. Schaetz. Implementing Modular Domain Specific Languages and Analyses. In *Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation*, pages 35–40, 2012.
13. R. Silva and M. Butler. Supporting Reuse of Event-B Developments through Generic Instantiation. In K. Breitman and A. Cavalcanti, editors, *11th International Conference on Formal Engineering Methods (ICFEM)*, volume 5885 of *Lecture Notes in Computer Science*, pages 466–484. Springer, 2009.
14. R. Silva and M. Butler. Shared Event Composition/Decomposition in Event-B. In B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Formal Methods for Components and Objects (FMCO)*, pages 122–141. Springer, 2010.
15. C. Snook, F. Fritz, and A. Illisaov. An EMF Framework for Event-B. In *Workshop on Tool Building in Formal Methods - ABZ Conference*, 2010.

# Ensuring OSGi Component Based Properties at Runtime with Behavioral Types

Jan Olaf Blech

RMIT University, Melbourne fortiss GmbH, Munich

**Abstract.** We present work on using automata based behavioral descriptions (behavioral types) of OSGi components for monitoring their specified behavior at runtime. Behavioral types are associated with OSGi components. We are focusing on behavioral types that specify protocols defined by possible orders of method calls of and between components and specifications based on the maximal execution time of these method calls. Behavioral runtime monitors for detecting deviations from a specified behavior are generated for components automatically out of their behavioral types. We sketch the integration of our behavioral runtime monitors into a behavioral types framework and present implementation and evaluation work on the behavioral runtime monitoring part.

## 1 Introduction

In our work, we are extending the basic typing concepts of traditional software component systems with means for specifying possible behavior of components. As with traditional types, like primitive datatypes and their composition, our *behavioral types* can be used for eliminating possible sources of errors at development time of software systems. This is analog to classical static type checks performed by a compiler. Furthermore, we can use behavioral types for eliminating possible sources of errors at runtime. This is analog to dynamic type checks performed when accessing pointers that reference data with types that can not be statically determined in some classical programming languages. Behavioral types also provide additional information about components which can be used for further tool based operations. Ensuring behavioral type correctness at runtime of an OSGi system is the main focus of this paper.

In this paper, we are using finite automata based descriptions of method call orders and maximal execution times of methods. Programmers even outside the academic community seem to be familiar with finite automata and thus, we believe that it is a good candidate for the acceptance of our specification formalism. We present a first version of an implementation [1] for the OSGi [14] framework [2]. OSGi allows dynamic reconfiguration of Java based software systems. In this paper, we concentrate on checking / ensuring of behavioral type safety at runtime of a system using *behavioral runtime monitors* generated from our behavioral types. We monitor a system's execution and throw behavioral types exceptions in case of deviations from the specified behavior.

[1] The parts of our behavioral types framework concerning behavioral runtime monitors as described in this paper are available at http://sourceforge.net/p/beht/wiki/Home

[2] Among other aspects of the framework, additional implementation details are described in [7]

Our work features the following highlights: 1) The use of multi-purpose automata based behavioral types. The same specification files can be used for other operations at compile time, e.g., static analysis of component compatibility, and runtime, e.g., discovery of components in a SOA like scenario, dynamic adaptation of components [6]. 2) The enforcement of these types at runtime by generating behavioral runtime monitors out of the types, using aspect oriented programming and an integration into Java by throwing runtime exceptions in case of deviation, and ensuring of maximal execution time of methods by using aspect oriented programming and runtime exceptions. There is no need to modify or add special comments to the source code files of the system. We think that this is highly beneficial for the acceptance of our approach since existing practices in Eclipse can still be used, people who are not interested in using behavioral types may still work on the same code base.

*Overview* Related work is discussed in Section 2. Our behavioral types are introduced in Section 3 together with behavioral runtime monitors. Section 4 describes the monitor integration with Java and AspectJ. Section 5 presents an example and a conclusion is given in Section 6.

## 2 Related Work

Interface automata [1] are one form of behavioral types. Like in this work, component descriptions are based on automata. The focus is on communication protocols between components which is one aspect that we also address in this paper. While the used formalism for expressing behavior in interface automata is more powerful (timed automata vs. automata vs. timing annotation per method), interface automata do not target the main focus of this paper: checking the behavior at runtime of a component by using some form of monitoring. They are especially aimed at compatibility checks of different components interacting at compile time of a system. The term behavioral types is used in the Ptolemy framework [12]. Here, the focus is on real-time systems.

The runtime verification community has developed frameworks which can be used for similar purpose as our behavioral type based monitors. The MOP framework [15] allows the integration of specifications into Java source code files and generates AspectJ aspects which encapsulate monitors. Compared to this work, the intended goals are different. While we keep the specification and implementation part separate, in order to be able to use the specification for different purposes at development, compile and runtime, a close integration of specification and code is often desired and realized in the runtime verification frameworks. A framework taking advantage of the trade-off between checking specifications at runtime and at development time has been studied in [9]. A framework that generates independent Java monitors leaving the instrumentation aspect to the implementation is described in [3]. Other topics explored in this context comprise, e.g., the efficiency and expressiveness of monitoring [2, 4] but are less focused on software engineering aspects compared to this paper.

Monitoring of performance and availability attributes of OSGi systems has been studied in [17]. Here, a focus is on the dynamic reconfiguration ability of OSGi. Another work using the .Net framework for runtime monitor integration is described in [11].

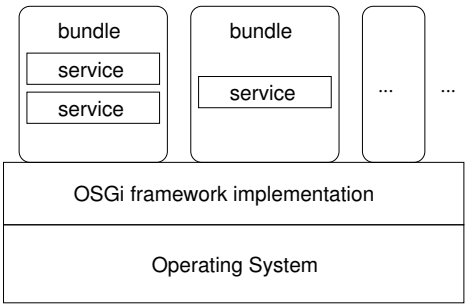


Fig. 1: OSGi framework

Runtime enforcement of safety properties was initiated with security automata [16] that are able to halt the underlying program upon a deviation from the expected behaviors. In our behavioral types framework, the enforcement of specifications is in parts left to the system developer, who may or may not take potential Java exceptions resulting from behavioral type violations into account.

Means for ensuring OSGi compatibility of bundles realized by using an advanced versioning system for OSGi bundles based on their type information is studied in [5]. Some investigations on the relation between OSGi and some more formal component models have been done in [13]. Aspects on formal security models for OSGi have been studied in [10].

## 3 Behavioral Types for OSGi

We present an overview on OSGi and describe our behavioral types. We present our vision for integrating behavioral types in the development and life-cycle of OSGi systems. Furthermore, we present the implemented generation of runtime monitors from behavioral types.

### 3.1 OSGi Overview

The OSGi framework is a component and service platform for Java. It allows the aggregation of services into bundles (cf. Figure 1) and provides means for dynamically configuring services, their dependencies and usages. It is used as the basis for Eclipse plugins but also for embedded applications including solutions for the automotive domain, home automation and industrial automation. Bundles can be installed and uninstalled during runtime. For example, they can be replaced by newer versions. Hence, possible interactions between bundles can in general not be determined statically.

Bundles are deployed as .jar files containing extra OSGi information. Bundles generally contain a class implementing an OSGi interface that contains code for managing the bundle, e.g., code that is executed upon activation and stopping of the bundle. Upon activation, a bundle can register its services to the OSGi framework and make it available for use by other bundles. Services are implemented in Java and typically realized

by registering a service object implementing a special interface. The bundle may itself start to use existing services. Services can be found using dictionary-like mechanisms provided by the OSGi framework. Typically one can search for a service which is provided using an object with a specified Java interface.

In the context of this paper, we use the term OSGi component as a subordinate concept for bundles, objects and services provided by bundles.

The OSGi standard only specifies the framework including the syntactical format specifying what bundles should contain. Different implementations exist for different application domains like Equinox[3] for Eclipse, Apache Felix[4] or Knopflerfish[5]. If bundles do not depend on implementation specific features, OSGi bundles can run on different implementations of the OSGi framework.

Services can run in parallel and are – if not explicitly synchronized – asynchronous. Method calls, even between objects in different bundles – are non-blocking. In the context of behavioral runtime monitoring using behavioral types, we are interested on how to monitor relevant semantics features of the runtime behavior rather than reasoning about the semantics features themself. For this paper, the monitoring of the order of method calls within and between components and their timing behavior and the dynamic creation and handling of monitors in accordance with the dynamic handling of bundles and objects are relevant.

### 3.2 Behavioral Types

Our behavioral types provide an abstract description of a components behavior and thus provide a way of formalizing specifications associated with the component. They can be used as a basis for checking the compatibility of components – for composing components into new ones, and interaction of different components – and for providing ways to make components compatible using coercion. Type conformance can be enforced at compile time (e.g., like primitive datatypes int and float in a traditional typing system) – if decidable and feasible – or at runtime of a system – e.g., like whether a pointer is assigned to an object of a desired type at runtime in a traditional typing system.

In our work behavioral types are realized as files that contain a description of (parts of the) behavior of an OSGi component. Typically, there should be one file per bundle, or class definition. But different aspects of behavior may also be realized using different files. In Eclipse the files are associated with an OSGi bundle by putting them in the same project folder in the Eclipse workspace. Here, behavioral types are formally defined using the following ingredients.

*Behavioral Type Automaton* A behavioral type automaton is a finite automaton represented as a tuple $(\Sigma, L, l_0, E)$ comprising an alphabet of labels $\Sigma$, a set of locations $L$, an initial location $l_0$ and a set of transition edges $E$ where each transition is a tuple $(l, \sigma, l')$ with $l, l' \in L$ and $\sigma \in \Sigma$. A consistency condition on our types is that all $\sigma \in \Sigma$ appear in some transition in $E$.

[3] http://www.eclipse.org/equinox/
[4] http://felix.apache.org/site/index.html
[5] http://www.knopflerfish.org/

In this paper, since we are interested in method calls, $\Sigma$ is the set of method names of components. The definition presented here can be used for specifying the behavior of single objects, all objects from a class, bundles and their interactions. It can be used for monitoring incoming method calls, outgoing method calls, or both.

*Maximal Execution Time Table* In addition to the protocol defined by the behavioral type automaton, we define the maximal execution time of methods as a mapping $\Sigma \rightarrow long \cup \perp$ from the set of method names $\Sigma$ to their maximal execution time in milliseconds. The specification of a maximal execution time is optional, thus, the $\perp$ entry indicates that no maximal execution time is set.

The behavioral type automaton together with the corresponding maximal execution time table form a behavioral type. Additional descriptive information is optionally available, but not used for the behavioral runtime monitoring aspects that are described here. Other representations such as $\Sigma$ comprising method signatures and timing information are possible future extensions.

### 3.3 Behavioral Types at Development and Runtime of a System

A potential major advantage of using behavioral types is the support of a seamless integration of behavioral specification throughout the development phase and the life cycle of a system. Our behavioral types can be used for different purposes (we proposed them in [8]) at development and runtime. A main idea of using behavioral types at development time is to derive them from requirements and use them for refinement checking of different forms of specification for the same entity that are supposed to have some semantical meaning in common. For example, the abstract specification, source code and compiled code of the same component represent different abstraction levels and should fulfill the same behavioral type. Checking this could be done by using static analysis at development time. At the end of a development process, a developed OSGi bundle is deployed including the behavioral type files. These can now be used for additional (dynamic) operations in the running system. Figure 2 shows two operations which can be carried out at runtime of a system: the registration and discovery of components using the OSGi framework, the compatibility, e.g., deadlock checking of bundle interaction protocols. Behavioral runtime monitors and their derivation from the development process are shown in Figure 3. The Figure shows the generation of the behavioral runtime monitor and its connection using aspects at development time and the actual monitoring at runtime. Up till now, we have implemented editors, registration of OSGi components, compatibility (deadlock freedom) of specifications, some form of dynamic adaptation as proposed in [6], and the behavioral runtime monitors which are new to this paper.

### 3.4 Monitor Generation

Regardless of what we intend to monitor, the monitor generation from a specification is the same. It is done automatically from a behavioral type file and generates a single Java file that defines a single monitor class.

Figure 4 shows a generated monitor. Monitors are generated as classes bearing a name derived from the original behavioral type. They comprise a map `maxtimes` that

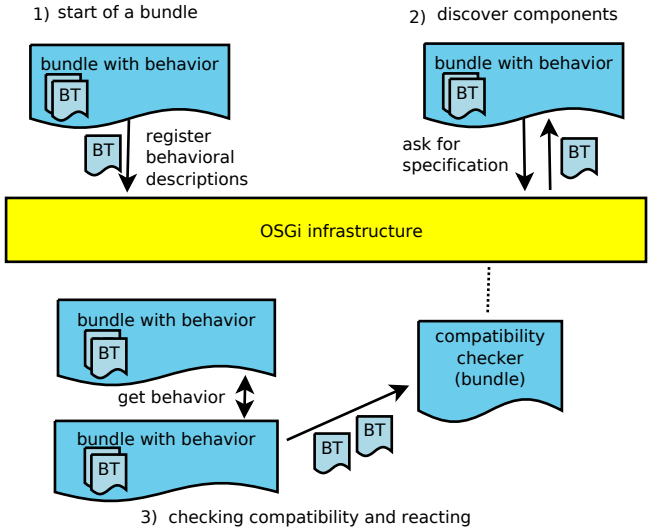


Fig. 2: Behavioral types at runtime

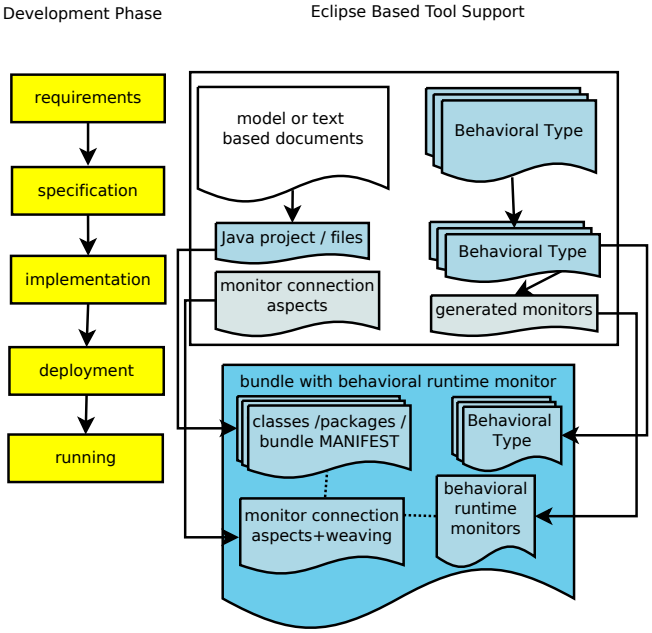


Fig. 3: Derivation of behavioral runtime monitors

maps method names to their maximal execution time in milliseconds. This entry is optional. If present, this map is initialized by the constructor –

`public clientinstance_out_realistic_simple_mon()`

in the example – of the monitor with the values specified for methods in the behavioral type file. Generated from an automaton from the behavioral type our behavioral runtime monitors comprise a static enumeration type with the location names of the automaton. In the automaton, the locations `LOCs0, LOCs1` are present. Using this type a state transition function generated from the transition relation is generated. The state transition function takes a string encoding a method name – event name – and updates a state field `protected LOCATION state` of the method. This field is initialized on object creation with the name of the initial state: `LOCs0` in the example.

```
package monitors;
import ....
public class clientinstance_out_realistic_simple_mon {
   public Map<String,Long> maxtimes = new HashMap<String,Long>();
   public clientinstance_out_realistic_simple_mon() {
        maxtimes.put("listFlights",new Long(1000)); }
   public static enum LOCATION {  LOCs0 , LOCs1 }
   protected LOCATION state = LOCATION.LOCs0;
   public boolean nextState(String event) {
      boolean rval = false;
      switch (state) {
         case LOCs0:
            ...
            break;
         case LOCs1:
            if (event.equals("listFlights")) {
               state = LOCATION.LOCs1;
               rval = true;
            }
            ....
            if (event.equals("listFlight")) {
               state = LOCATION.LOCs1;
               rval = true;
            }
            break;
      }
      return rval;
   }
}
```

Fig. 4: Generated example monitor

## 4 Behavioral Runtime Monitor Integration using AspectJ

The generated monitors are connected to the component that shall be observed using AspectJ aspects. AspectJ is an extension of Java that features aspect oriented programming. Aspects are specified in separate files and feature pointcuts that allow the specification of locations where Java code specified in the aspect shall be added to existing Java code. This weaving of aspect code into existing Java code is done on bytecode level.

Monitors are created and called from aspects. All extra code needed to integrate the monitors is defined in the AspectJ files or in libraries accessed through the AspectJ files. There is no need to touch the source code of a component. This independence of source code and specification is a design goal of our framework. We distinguish different kinds of monitor deployment. Each kind requires its own aspect template and its instantiation.

*Singleton monitors* In some cases it is sufficient to use a singleton instance of a monitor. This is the case when monitoring all the method calls that occur in a bundle, within all objects of a class, or within a singleton object. For monitoring method call orders, we use a `before` pointcut in AspectJ. Figure 5 shows an example aspect: Here, before the calls to methods – specified in the execution pattern after the ":" in the pointcut – of all objects of class `MiddlewareProc` an update on the state transition function – the `com.nextState` – is inserted. We extract the name of the called method using reflection and a helper method `AJMonHelpers.getMethodName` and pass it to the state transition function. In addition to updating the state field in the monitor we get a boolean value indicating whether the monitored property is still fulfilled. In case of a deviation the `BehavioralTypeViolationException` – a runtime exception is thrown. The implementation of the `MiddlewareProc` class may or may not catch this exception and react to it.

```
package bookingsystem.middleware;
import java.util.HashMap;
import java.util.Map;
....
import monitors.*;

public aspect CallincprotocolMiddlewareProc {
    ...
    pointcut myMethod(MiddlewareProc p): this(p) &&
       within(MiddlewareProc) && execution(* *(..));
    before (MiddlewareProc p): myMethod(p) {
        ...
     boolean verdict = com.nextState(
           AJMonHelpers.getMethodName(
              thisJoinPointStaticPart.getSignature()));
     if (!verdict) throw new BehavioralTypeViolationException();
    }
}
```

Fig. 5: Example aspect

*Multiple monitor instances* In same cases we want to monitor each object of a class with an independent monitor. Here, we create on call of the object's constructor an individual monitor for the object. It is added to a (hash)map (Object $\rightarrow$ Monitor). Since the AspectJ pointcuts are defined with respect to the static control flow information specified in the source code of a class, on each call of a method belonging to the class to be monitored, we use the same code in each object and chose the monitor for the particular object by looking it up from the map and advance the respective monitor state.

*Monitoring of time* Monitoring time is done using Java timers within the Java code associated with the pointcuts. On call of a method we create a timer that is scheduled to throw an exception after the specified maximal execution time. Using the `after` pointcut, the timer is canceled if the method's execution finishes on time and thus, no exception is thrown in this case.

The adaptation of an aspect for monitoring a particular component is simple. One has to take the appropriate AspectJ .aj file and adapt it, by inserting the names of the classes and packages that shall be monitored and the correct monitor names. Weaving of the aspects is done automatically on Java bytecode level and no additional configuration needs to be done.

## 5 Example and Evaluation

One example scenario regarded by us is the flight booking system (our set-up comprises only the functionality necessary for our monitoring experiments) shown in Figure 6. OSGi components and their interactions are shown. Clients are represented as proxy components in the system and served by middleware processes which are created and managed by a coordination process. Middleware processes use concurrently a flight database and a payment system which are represented by proxy OSGi components. We have investigated the communication structure between the components and investigated deployment of monitors. This comprises the following cases: 1) The use of multiple monitors running in parallel and being created at runtime for different objects which are created dynamically. In the example system this is the case for the middleware processes, where processes are created as separate objects on demand and are monitored independently of each other. 2) The monitoring of all objects of a single class using a single monitor and the monitoring of singleton objects and the monitoring of bundle behavior. This is, e.g., the case in the payment subsystem. 3) Furthermore, we have investigated the monitoring of maximal execution time of methods. In the example system this is the case in the payment subsystem and access to the flight database. We did not find any major problems in our approach.

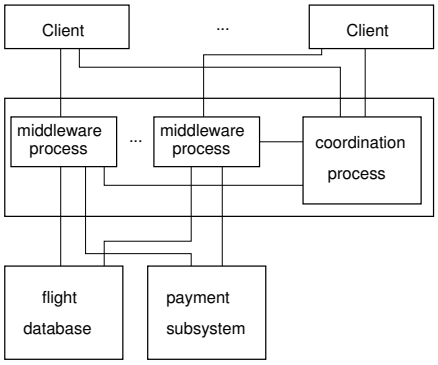


Fig. 6: Components of our flight booking system

## 6 Conclusion

We presented work on behavioral types for Java / OSGi components and the monitoring of behavioral type based specifications at runtime of a system. Our Eclipse based implementation allows the behavioral runtime monitoring of components without modifying their source code by using aspect oriented programming. In addition to the behavioral runtime monitoring work, the same behavioral types can be used for other operations at compile time, e.g., static analysis of component compatibility, and runtime, e.g., discovery of components in a SOA like scenario, dynamic adaptation of components [6].

## References


1. L. de Alfaro, T.A. Henzinger. Interface automata. Symposium on Foundations of Software Engineering, ACM , 2001.
2. H. Barringer, Y. Falcone, K. Havelund, G. Reger, D. Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. Formal Methods, vol. 7436 of LNCS, Springer-Verlag, 2012. (FM'12)
3. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Verification, Model Checking, and Abstract Interpretation, vol. 2937 of LNCS, Springer-Verlag, 2004. (VMCAI'04)
4. Bauer, A., Leucker, M.: The theory and practice of SALT. NASA Formal Methods, vol. 6617 of LNCS, Springer-Verlag, 2011.
5. J. Bauml and P. Brada. Automated Versioning in OSGi: A Mechanism for Component Software Consistency Guarantee. Euromicro Conference on Software Engineering and Advanced Applications, 2009.
6. J. O. Blech, Y. Falcone, H. Rueß, B. Schätz. Behavioral Specification based Runtime Monitors for OSGi Services. Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), vol. 7609 of LNCS, Springer-Verlag, 2012.
7. J. O. Blech, H. Rueß, B. Schätz. On Behavioral Types for OSGi: From Theory to Implementation. http://arxiv.org/abs/1306.6115. arXiv.org 2013.
8. J. O. Blech and B. Schätz. Towards a Formal Foundation of Behavioral Types for UML State-Machines. UML and Formal Methods. ACM SIGSOFT Soft. Eng. Notes, 2012.
9. E. Bodden, L. Hendren. The Clara framework for hybrid typestate analysis. Software Tools for Technology Transfer (STTT), vol. 14, 2012.
10. O. Gadyatskaya, F. Massacci, A. Philippov. Security-by-Contract for the OSGi Platform. Information Security and Privacy Conference, IFIP Advances in Information and Communication Technology, vol. 376, 2012.
11. K.W. Hamlen, G. Morrisett, F.B. Schneider. Certified in-lined reference monitoring on .NET. Programming languages and analysis for security, ACM 2006.
12. E.A. Lee, Y. Xiong. A behavioral type system and its application in ptolemy ii. Formal Aspects of Computing, 2004.
13. M. Mueller, M. Balz, M. Goedicke. Representing Formal Component Models in OSGi. Proc. of Software Engineering, Paderborn, Germany, 2010.
14. OSGi Alliance. OSGi service platform core specification (2011) Version 4.3.
15. P. O'Neil Meredith, D. Jin, D. Griffith, F. Chen, G. Roşu. An Overview of the MOP Runtime Verification Framework. Software Techniques for Technology Transfer, Springer, 2011.
16. F.B. Schneider. Enforceable security policies. ACM Transactions on Information and System Security, vol. **3**, ACM, 2000.
17. F. Souza, D. Lopes, K. Gama, N. Rosa, R. Lima. Dynamic Event-Based Monitoring in a SOA Environment. On the Move to Meaningful Internet Systems, vol. 7045 of LNCS, Springer-Verlag, 2011.

# Symbolic Execution of Satellite Control Procedures in Graph-Transformation-Based EMF Ecosystems

Nico Nachtigall, Benjamin Braatz, and Thomas Engel

Université du Luxembourg, Luxembourg
firstname.lastname@uni.lu

**Abstract.** Symbolic execution is a well-studied technique for analysing the behaviour of software components with applications to test case generation. We propose a framework for symbolically executing satellite control procedures and generating test cases based on graph transformation techniques. A graph-based operational symbolic execution semantics is defined and the executed procedure models are used for generating test cases by performing model transformations. The approach is discussed based on a prototype implementation using the Eclipse Modelling Framework (EMF), Henshin and ECLiPSe-CLP tool ecosystem.

**Keywords:** symbolic execution, graph transformation, test case generation, triple graph grammars, EMF henshin

## 1 Introduction

Symbolic execution [4] is a well-studied technique for analysing the behaviour of software components with applications to test case generation. The main idea is to abstract from possibly infinite or unspecified behaviour. Uninitialised input variables or external function calls are represented by symbolic variables with the sets of possible concrete input values as value domains. Consequently, symbolic program execution is rather based on symbols than on concrete values leading to symbolic expressions which may restrict the value domains of the symbols. For each execution path, a path constraint (PC) is defined by a dedicated boolean symbolic expression. Solving the expression, i.e., finding a valuation for all contained symbolic variables so that the expression is evaluated to true, provides concrete input values under which the corresponding path is traversable. An execution path is traversable as long as its path constraint is solvable.

In this paper, we propose a framework for symbolically executing satellite control procedures (SCPs) and generating test cases based on graph transformation techniques [6]. We successfully apply graph transformation techniques in an industrial project with the satellite operator SES (Société Européenne des Satellites) for an automatic translation of SCPs from proprietary programming languages to SPELL (Satellite Procedure Execution Language & Library) [8]. The safety-critical nature of SCPs implies that extensive testing is required after

translation. The presented approach allows us to generate tests without leaving this graph-transformation-based ecosystem. We define a graph-based operational semantics for symbolically executing SCPs for a subset of the SPELL language. The executed procedure models are used for generating test cases by performing model transformations. We discuss our approach based on a prototype implementation using the EMF Henshin [7] and ECLiPSe-CLP [9] tools.

Sec. 2 introduces the symbolic execution framework. In Sec. 3, the graph-based operational symbolic execution semantics is defined. Sec. 4 presents model transformation rules for test case generation and a prototype implementation for the approach. Sec. 5 concludes the paper and compares with related work.

## 2 Models & Symbolic Execution Framework

The symbolic execution framework in Fig. 1 uses the abstract syntax tree (AST) of a procedure, which defines a typed attributed graph [6]. The AST can be constructed by parsing the source code with appropriate tools (see Sec. 4). The AST graph is symbolically executed using two graph transformation systems (GTSs). A GTS is a set of graph transformation rules where each rule may create or delete nodes and edges or update attribute values. In phase one, the GTS $GTS_{Flow}$ is used to annotate $AST$ with execution flow information leading to graph $AF = AST + FLOW$. In phase two (symbolic execution), the GTS $GTS_{Sym}$ is exhaustively applied to $AF$ leading to graphs $State_i = AF + SYM_i, i = 1..n$ representing the status of the execution.

Fig. 2 shows the running example in a small subset of the SPELL language [8]. SCP "Charge Batteries" retrieves the state of charge (SC) of both batteries of a satellite, defines a minimal threshold min of 50%, and switches to the battery with higher SC if it exceeds min. Otherwise, an alert is issued. Meta-model SCP specifies the general syntax of ASTs for such procedures. A procedure Proc consists of a list of statements Stmnt (assignments Asg, function calls FnCall, definitions FnDef or branching If structures) with explicit next pointers. Function calls contain a list of arguments (arg) and function definitions contain a list of parameters (pm). An assignment contains a variable (var) and an assigned expression (ex). Expressions (Expr) are either numbers (Number), variables (Var) or Boolean expressions (Bool) with operator ($<, <=, >, >=, and, or$) and operands (left (le) and right (ri)). Complex statements (If,FnDef) contain a block (B) that references a list of statements. Furthermore, If statements have a boolean condition cond and may have else and ElIf structures (edge el). The $AST$ for the procedure is a graph typed over meta-model SCP. Graph $FLOW$ represents the flow annotation of $AST$. Places P are assigned (dotted edges asg) to nodes in $AST$ that should

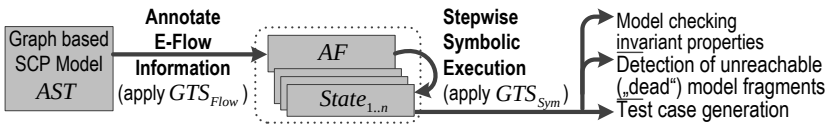


**Fig. 1.** Steps of Symbolic Model Execution

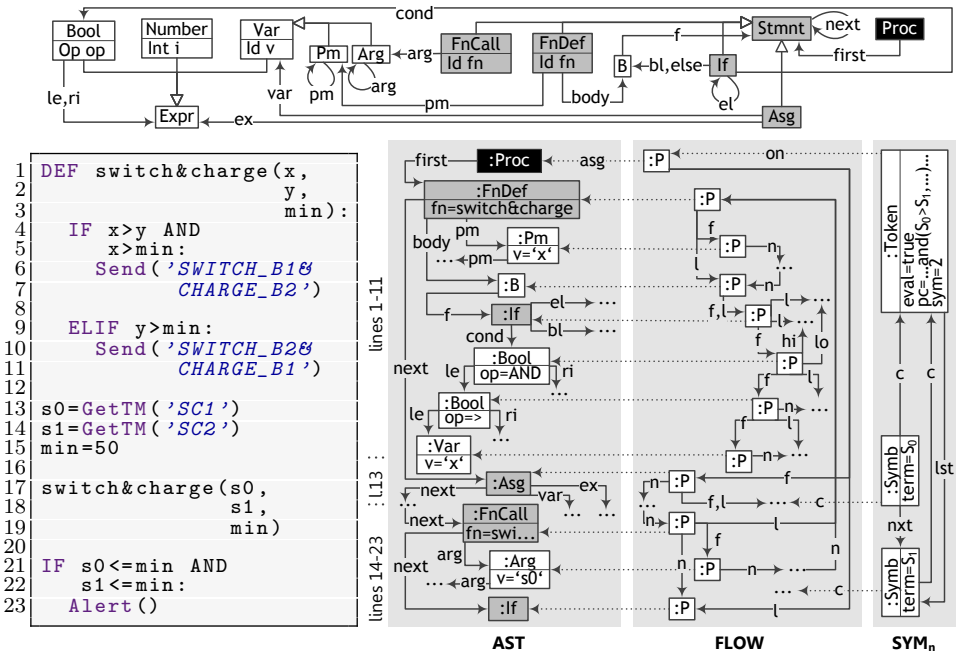


**Fig. 2.** SCP meta-model (top), SCP "Charge Batteries" (left), SCP model (AST), flow annotation (FLOW) and symbolic execution elements ($SYM_n$)

be executed. P nodes can be connected by f,l or n edges in order to indicate which other nodes need to be executed at first, next or last before finishing the execution of a node, e.g., in order to execute the procedure (node Proc), the assignment in line 13 (node Asg) needs to be executed first. For each execution path, a Token representing the current execution point with path constraint is created (in total six Tokens for the example). In graph $SYM_n$, the Token node on place P that is assigned to node Proc indicates that the procedure was evaluated (eval=true) with path constraint pc=and(and($S_0$>$S_1$,$S_0$>50),not(and($S_0$<=50,$S_1$<=50))) and symbolic variables $S_0, S_1$ (Symb) for GetTM('SC1') and GetTM('SC2') by entering line 6 but not line 23. The resulting graphs $State_{1..n}$ can be used for model checking invariants, detection of dead model fragments or test generation.

## 3 Operational Execution Semantics

The execution semantics is divided into the execution flow of the AST graph and the token semantics for traversing all flow paths. Fig. 3 shows the rules of $GTS_{Flow}$ and $GTS_{Sym}$ in short-hand notation, i.e., nodes and edges marked with <+> are created, those marked with <-> are deleted and attribute values of the form [x=>y] are updated from x to y when applying the rule. Nodes marked with <tr> have a "hidden" translation attribute that is updated [false=>true] during rule application so that the rule is only applied once. A more formal definition of graph transformation in general is given in [6].

Rule $\mathsf{Init}_1$ specifies that the first statement of a procedure needs to be executed first. An initial token is put on the first place with path constraint true and eval=false. Rule $\mathsf{Stmnt}_1$ defines that successive statements need to be executed successively. Note that the first statement in Fig. 2 is a FnDef. Therefore, another rule defines that the succeeding statement needs to be executed first until there is no more FnDef. Rule $\mathsf{Asg}_1$ defines that the expression of an assignment needs to be evaluated first before assigning the resulting value. The rule for blocks is defined analogously. Rule $\mathsf{Bool}_1$ defines that the left operand has to be evaluated before the right operand. Rule $\mathsf{If}_1$, branches the flow - the condition is evaluated before executing the block (hi - positive condition) or an "empty" place (lo - negative condition). Rule $\mathsf{EIIf}_1$ links the "empty" place of rule $\mathsf{If}_1$ to the alternative If. Rule $\mathsf{Else}_1$ is defined analogously.

Rule $\mathsf{TFst}_2$ moves the token to the first child place (edge f) as long as possible. Rule $\mathsf{TNxt}_2$ moves the token of an evaluated place to the next place (edge n) and changes attribute eval to false. The rules implement a left-most inner-most evaluation strategy. Rule $\mathsf{GetTM}_2$ evaluates each GetTM-FnCall to a path-wide unique symbolic variable (Symb) $S_i$ (uniqueness is given by token attribute sym which is increased by one). Note that Symbs are ordered in their occurence of evaluation which is important for a later test generation. Rule $\mathsf{GetTM}_2$ requires that a last Symb already exists. An analogue rule creates a last Symb if not existent. Rule $\mathsf{Asg}_2$ assigns the term of the evaluated expression to the variable Var and sets the assignment as evaluated by moving token edge on. Rule $\mathsf{BoolAnd}_2$ concatenates

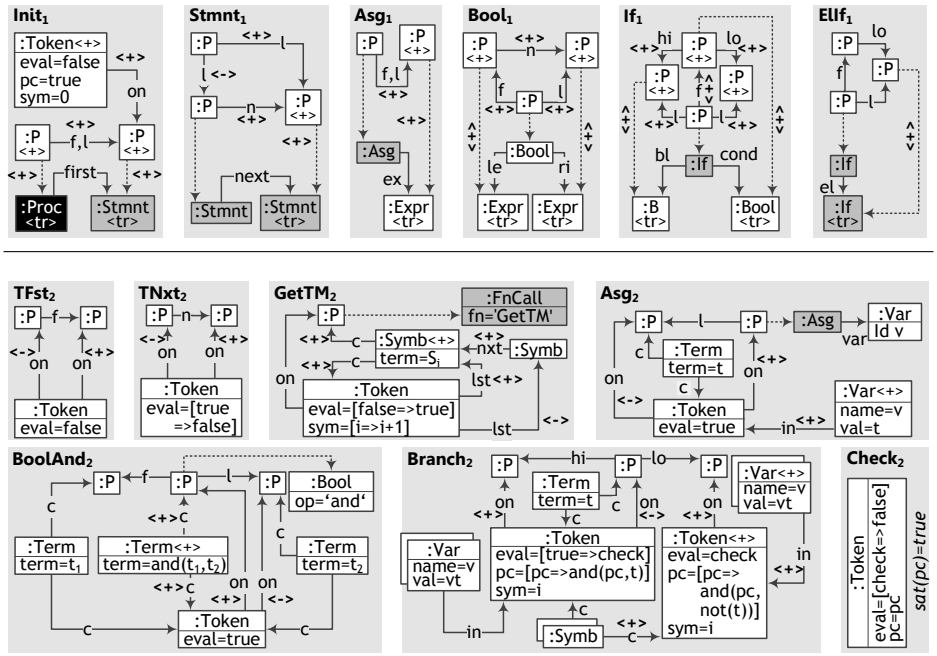


**Fig. 3.** Rules for annotation (top, $GTS_{Flow}$) and symbolic execution (bottom, $GTS_{Sym}$)

both evaluated operands with and. Analogue rules for boolean expressions with other operators (or,<,>,etc.) are defined. Amalgamated rule $\mathsf{Branch}_2$ duplicates the token with all connected variables and symbols, negates the condition (not(t)) for the lo path and concatenates the condition with the path constraint. Rule $\mathsf{Check}_2$ checks, if the path constraints of duplicated tokens are still satisfiable after a branch (attribute condition sat(pc)=true). If the path constraint of a token becomes unsatisfiable, the token status eval remains check and the token can not be moved any more. After simulating the example in Fig. 2, three tokens from six possible paths are assigned to node Proc with eval=true while the other three tokens remain at the last If statement with unsolvable path constraints. Only tokens that are assigned to node Proc with eval=true are considered during test case generation (they represent execution paths with solvable path constraints). Additional rules are defined for annotating and traversing function calls and definitions. A function is traversed every time it is called so that global side effects in execution can be respected, e.g., operations on call by reference arguments.

## 4 Implementation & Test Case Generation

A procedure is parsed with Xtext [5] to an EMF AST graph first. Then, the EMF Henshin tool [7] is used in combination with the ECLiPSe constraint solver [9] to execute the AST graph by automatic rule applications and satisfiability checking / solving constraints. The AST graph is completely preserved during execution. Correspondences between symbolic variables of path constraints (nodes of type Symb) and AST graph structures are used for test generation by applying the forward model transformation (FT) rules in Fig. 4. An FT rule [8] consists of a source graph $G_S$, correspondence graph $G_C$ and target graph $G_T$. While $G_S$ is parsed, nodes and edges in $G_C, G_T$ are created. Applying the rules yields a graph that is serialised to test case files with Xtext. Rule Proc2Test creates a Test suite for a procedure. Rule Token2Case creates a test case for each execution path with solvable path constraint. Rule LstSymb2KeyElem adds a key tm for each last (edge lst) evaluated GetTM(tm) function call of an execution path to the test case with a list containing a test input valuation for symbolic variable s as first (edge fst) element so that path constraint pc is satisfied (solve(s,pc)). A second rule handles all previous symbolic variables. Symbolic variables are ordered in order to reflect the sequential execution order of the represented GetTM function calls which is needed for proper test generation with test inputs in correct order.

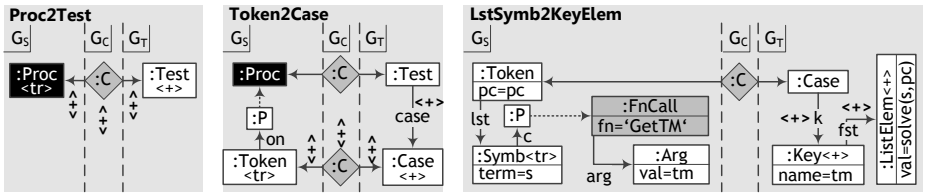


**Fig. 4.** Rules for test case generation

## 5 Conclusion & Related Work

We have presented a framework for symbolically executing simple satellite procedures. The approach preserves the correspondences between symbolic variables and the AST, so that the result graph can be used for test case generation by performing model transformations afterwards. A prototype implementation for the symbolic execution and test case generation framework has been presented.

In contrast to interpreter based symbolic execution engines [1,3] of programming languages, our graph-transformation-based approach allows symbolic execution on a more abstract level enabling its formal analysis and application to other languages and behavioural diagrams in future work. In [2], an abstract symbolic execution framework based on term rewriting is proposed. In contrast to our approach, the correspondences between symbolic variables and the program term are not preserved. Research on how to transfer and enhance results from term to graph rewriting approaches for symbolic execution is topic of future work. In [10], the execution of UML state machines is presented but diagram–path-constraint correspondences are not specified explicitly.

In future work, we plan to extend the symbolic execution semantics for applicability to industrial SPELL SCPs [8] and analyse its correctness w.r.t. a formal SPELL semantics that needs to be defined first. We will investigate important properties of symbolically executed models that should be preserved during model refactorings and how to ensure their preservation. Moreover, we will assess the scalability of our approach.

## References

1. Anand, S., Pasareanu, C.S., Visser, W.: JPF-SE: A symbolic execution extension to java pathfinder. In: TACAS. pp. 134–138 (2007)
2. Arusoaie, A., Lucanu, D., Rusu, V.: A Generic Approach to Symbolic Execution. Tech. Rep. RR-8189, INRIA (Dec 2012)
3. Cadar, C., Dunbar, D., Engler, D.: Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX conference on Operating systems design and implementation. pp. 209–224. OSDI'08, USENIX Association, Berkeley, CA, USA (2008)
4. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM 56(2), 82–90 (Feb 2013)
5. The Eclipse Foundation: Xtext (2013), `http://www.eclipse.org/Xtext/`
6. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation, vol. EATCS Monographs in Theoretical Computer Science. Springer (2006)
7. EMF Henshin (2013), `http://www.eclipse.org/modeling/emft/henshin/`
8. Hermann, F., Gottmann, S., Nachtigall, N., Braatz, B., Morelli, G., Pierre, A., Engel, T.: On an Automated Translation of Satellite Procedures Using Triple Graph Grammars. In: Proc. ICMT'13, LNCS, vol. 7909, pp. 50–51. Springer (2013)
9. Schimpf, J., Shen, K.: Eclipse - from lp to clp. Theory and Practice of Logic Programming 12, 127–156 (2012)
10. Zurowska, K., Dingel, J.: Symbolic execution of UML-RT State Machines. In: Proc. of SAC '12. pp. 1292–1299. SAC '12, ACM (2012)

# Research Questions for Validation and Verification in the Context of Model-Based Engineering

Catherine Dubois[1], Michalis Famelis[2], Martin Gogolla[3], Leonel Nobrega[4], Ileana Ober[5], Martina Seidl[6], and Markus Völter[7]

[1] ENSIIE, Évry, France
[2] University of Toronto, Canada
[3] University of Bremen, Germany
[4] University of Madeira, Funchal, Portugal
[5] University of Toulouse, France
[6] Johannes Kepler University Linz, Austria
[7] Völter Ingenieurbüro, Heidenheim, Germany

**Abstract.** In model-based engineering (MBE), the abstraction power of models is used to deal with the ever increasing complexity of modern software systems. As models play a central role in MBE-based development processes, for the adoption of MBE in practical projects it becomes indispensable to introduce rigorous methods for ensuring the correctness of the models. Consequently, much effort has been spent on developing and applying validation and verification (V&V) techniques for models. However, there are still many open challenges.
In this paper, we shortly review the status quo of V&V techniques in MBE and derive a catalogue of open questions whose answers would contribute to successfully putting MBE into practice.

## 1 Introduction

This paper is based on the discussions of a working group (whose members are mainly the authors of this paper) of the 2013 Dagstuhl seminar number 13182 on *Meta-Modeling Model-Based Engineering Tools*. The working group addressed a panel of important issues with *validation and verification* (V&V) of models and model transformations whose correctness is crucial for the successful realization of model-based engineering (MBE) projects.

Overall, models help to clarify and plan the development of software. Furthermore, they allow the use of methods to ensure quality and discover errors in conceptual designs. However, with the extensive use of models during the software development process, there is the danger of introducing defects into the models. For detecting or avoiding such defects, techniques of validation and verification (V&V) for models help.

Besides models, model transformations (MTs) are at the heart of model-based engineering. Different kinds of MTs underlie the engineering activities and their

associated tools. They can be used for example to refine models, to generate code, etc. This variety of purposes of MTs emphasizes the need for effective and dedicated V&V techniques for animating, testing, and proving them.

In a model-based development process, V&V techniques help to better understand models and to assess model properties which are stated implicitly in the model. V&V serves to inspect models and to explore modeling alternatives. Model validation answers the question 'Are we building the right product?' whereas model verification answers 'Are we building the product right?', similar as V&V does for code. Validation is mainly an activity done by the developer demonstrating model properties to the client, whereas in verification the developer uncovers properties relevant within the development process. Verification is closely related to testing and covers automatic and semi-automatic (static) analysis techniques like theorem proving and model-checking.

In this paper, we shortly review the status quo of V&V techniques in MBE and derive research questions whose answers will push the whole field forward. The methodology behind this paper is as follows. First, we started to list objectives, methods, tools, use cases and then highlighted difficulties, bottlenecks and pitfalls. On this basis, we derived some key research questions. This paper sums up these discussions and lists the different research questions.[8]

The main conclusion of our analysis of the actual situation is that specific V&V methods for models and MTs are required. Although numerous tools exist to verify and validate hardware and software, they do not apply straightforwardly to models and MTs. In [8], E. M. Clarke, E. A. Emerson and J. Sifakis mentioned abstraction techniques as one of the promising paths for the future advances in the field of verification. The use of V&V techniques in the context of MBE follows the direction of abstraction. However V&V for models is concerned by a more complete abstraction mechanism that covers the entire system, whereas in [8], abstraction is considered mostly at mathematical level and targets formal semantics.

The rest of the paper is organized as follows: In Section 2, we reflect on the situation and describe some challenges. Subsequently, the research questions which emerged are listed in Section 3. We conclude in Section 4.

## 2 Status and Challenges

In this section we describe the main observations regarding the state of theory and practice that informed the discussions of our working group and list proposed solutions found in the literature. The observations are organized thematically, based on the main areas in which V&V and MBE intersect.

### 2.1 Gap Existing between Models and V&V Formalisms

In the context of model-based development, V&V faces new challenges. The origin of most of them stays in the gap existing between the (mostly high-level)

[8] We are aware that for some of them proposals already exist. Because of lack of space, we are not able to reference them all, and so we cannot reach exhaustiveness.

specification mechanisms, used at design time both for specifying the model and the properties that are subject to V&V, and the underlying V&V engines that use *low-level* formalisms.

The gap between the two specification mechanisms can be (at least partially) filled by using model transformations and traceability mechanisms [11], which are definitely needed in this context. Yet it is only part of the answer as behavioral semantics often leads to non-bijective correspondences between design time and runtime artifacts.

The V&V frameworks, which are typically using back-end tools using specific formalisms need to be complemented with mechanisms allowing the user to interact with the back-end tools through their own modeling languages. They do offer the right environment to reason about the system under modeling. In this context we can mention existing initiatives that go on this direction. In [6] a high-level change-driven transformation language is proposed in order to capture changes, even those that concern low level transformations (as it may be in the case of some feedback (e.g. counter-examples) produced by V&V tools. Supporting the user during the error diagnosis phases can be done e.g. by allowing customizable simulation trace visualization [1].

In the same spirit of hiding the technicalities related to the V&V engine, we can mention here the issue of (semi-)automatically managing the V&V machine configuration. Although not necessarily a research challenge, this is a typical functionality required in order to improve the accessibility to V&V tools.

### 2.2 Need to Refine Existing Methodologies

V&V tools have the potential of helping the user during model development, by allowing early V&V, similar to debugging facilities offered by programming development environments. Most of the generic methodologies identify some possible points where V&V could be used. In a realistic setting, the model-based development methodology should be refined in order to better identify which V&V activities are meaningful at the various phases of design, in order to take full advantage of the existing V&V engines. Such a methodology would most likely vary depending on the nature of the project and on the application domain, but one can expect that methodology definitions can be capitalized within the same business domain.

### 2.3 Importance of Snapshots

In the context of V&V, it is important to be able to clearly and completely describe a dynamic model configuration (often called model snapshot, or global state of the system) - in terms of existing objects, values of their respective attributes, existing links, active states in the state machine (if any), content of the input queue, etc. Such a snapshot could correspond to the state of the dynamic model before an error or otherwise identified as meaningful. A particular case of model snapshot is the one specifying the model instantiation, arising in the context of describing the initial state of a model execution. Each commercial

tool that offers model simulation functionalities uses some (more or less documented) mechanism for describing model instantiation, however this snapshot description does not cover arbitrary snapshots.

There are some approaches that tackle the idea of model snapshot, such as for instance, the USE tool [12] which allows the user to generate/verify UML model snapshots. Yet, no modelling tool uses such elaborate techniques.

### 2.4 Properties

V&V focuses on verifying some properties with different objectives. Properties to be verified differ according to the nature of models (e.g. static or dynamic) and to the stage in the development process (e.g. specification or code generation time). Furthermore there are also limitations and synergies depending on the applied V&V techniques (static analysis, testing, model checking, runtime verification, formal proof). However there is no one-to-one association between properties and techniques, the latter being for the most part complementary.

More generally, we can distinguish structural properties (composition of models, consistency, redundancy), behavioral properties (ensure liveness of a model or safety properties) and also quantitative properties (such as Worst Case Execution Time (WCET) or schedulability). Properties are related with models but also to model transformations. Here the classification is different, it is discussed in more detail in the next subsection. We can also distinguish between static and dynamic properties and in both categories refine into language inherent, model specific, generic or user-defined properties.

We can find in the literature many approaches and tools to verify models. However it is quite confusing and it is difficult to draw a classification allowing the answer to the following question: *What kind of property to verify on which model at what stage with what kind of technique?* Some partial answers exist, e.g. using feature models [10] or ontologies [15].

### 2.5 Model Transformations

Model transformations (MTs) are an integral part of model-based software engineering. MTs are used for a broad variety of purposes, ranging from applications as diverse as maintaining inter-model consistency to defining the translational semantics of a domain-specific language. This ubiquity of model transformations emphasizes the need for effective V&V techniques, that focus specifically on MTs. This is especially true in contexts of use where MTs are critical, such as code generation. At the same time, V&V for MTs differs from traditional program verification, since MTs are defined at a higher level of abstraction than programs, thus creating opportunities for more effective application of formal techniques. These factors have generated considerable interest in the research community, with special-interest events such as the VOLT workshop ("Verification Of ModeL Transformations") being organized since 2012.

An important challenge in studying the verification of MTs is to understand how verifying MTs is different from traditional program verification. There has

been some work mapping the challenges that are specific to MTs; for example, [4, 5] discuss the challenges of testing MTs, whereas [16] focuses specifically on bidirectional transformations.

In the same vein, it is necessary to understand how to best reuse existing verification tools, techniques and methods. For example, in [2], the autors propose a tri-dimensional approach that takes into account (a) the kind of transformation involved, (b) the properties of interest, and (c) the available verification techniques. The purpose is to identify what verification method is most appropriate based on the transformation and desired properties. Similarly, [10] classifies approaches for verifying transformations based on five criteria: (a) verification goal (e.g. consistency, correctness), (b) representation of the domain, (c) representation used for the verification task, (d) specification language used, and (e) verification technique (theorem proving, static analysis, model checking).

In general, the correctness of MTs is of paramount importance and is generally the ultimate goal of verification of MTs. Correctness is tightly related with specification. The *Tracts* methodology [18] proposes a generalization of model transformation contract, based on the idea of *duck typing* MTs with OCL constraints, while also supporting test-case generation. Another approach is to reuse transformation primitives in order to build correct-by-construction transformations, typically with a trade-off in expressive power, such as the case of DSLTrans [3].

Finally, the identification of properties that are meaningful when verifying MTs is also important. Confluence and termination have been studied extensively for transformations based on graph rewriting [9, 14]. In [2], a more thorough categorization of properties is proposed, marking the distinction between properties related to transformations themselves (e.g. determinism) and properties related to the result of applying the transformation (e.g. conformance of the output). For each of these major categories, several minor sub-categories are identified.

### 2.6 Informal vs. Formal vs. Incomplete Modeling

Starting from informal use case sketches, which can be connected to formal model elements by trace links, there is need during V&V to concentrate on particular model parts and to switch on or off particular model inherent elements (in class diagrams, e.g., multiplicities, or aggregation and composition restrictions). Furthermore we may desire to explicitly configure constraints by negating, deactivating or activating them (in class diagrams, e.g., class invariants and operation pre- and postconditions; in state charts, e.g., state invariants and transition pre- and postconditions). Such configuration options must be offered with different types of granularity: (a) all model elements may be relaxed, (b) only a manual model element selection can be considered for relaxation, or (c) a semi-automatic element selection for relaxation may be offered (such a semi-automatic selection might depend on a user-determined, editable criteria catalogue). In the ultimate vision one might consider sliders on the user interface of the modeling tool which allows the developer to gradually go from a strict, formal model through various

intermediate levels to a totally relaxed and informal model. However, if formal test cases and test scenarios are desired, then a minimal frame for test case construction must be preserved. Such a minimal frame could consist, e.g., for class diagrams of central classes and associations and for state charts of central states and transitions, with all model elements in the minimal version without any implicit or explicit constraints. Thus, there will be a tradeoff between switching off or ignoring certain model elements (like constraints or classes) and the degree of formality: if more model elements are switched off, the model will become more informal and will accept more scenarios; however, this procedure only works to a certain degree, because one can only formulate scenarios if at least a minimum selection of formal model elements is present; for a completely informal model no formal scenario can be formulated. Thus, ignoring certain model elements or constraints and the degree of formality are closely linked.

In order to handle incomplete and partial models wrt V&V, these model configuration options must be recorded along with the various V&V test scenarios, test cases and their results with respect to the configured model. Test cases and test scenarios must be invoked repeatedly with different configuration options and test results must be recorded in a systematic way. It must be possible to query test results in order to retrieve the proper configuration settings. The mechanisms for playing around with model relaxation can also be employed to allow for model alternatives. Model relaxations and model alternatives will span up a graph of model versions. These model versions must be connected to a graph of proper model test cases and model scenarios.

### 2.7 Comparison and Benchmarking

In the V&V research area, not only theoretical results are important, but also the tools which implement novel approaches in order to benefit from these results. Expressive benchmarks are necessary to evaluate the maturity of the tools as well as compare the power of novel approaches to established techniques. At dedicated workshops like the Comparison and Versioning of Software Models [17] the benchmarking issue has been a specially discussed topic which is one approach to obtain commonly accepted benchmarks. Often community-organized competitions and evaluations are a valuable source for the benchmarks which serve as basis for evaluations in scientific publications illustrating the benefits of their contributions. Examples of such events are the SAT solving competition [13] or the Transformation Tool Contest (TTC) [7].

For the community the advantages of a centrally organized competition are manifold. First of all, evaluations are performed in an environment equal to all participants, thus allowing a fair comparison of the different tools. Second, the benchmarks are not selected by the tool developers, but by some impartial experts and covers the whole spectrum of interesting test cases. Third, a clear documentation of the outcome is provided such that the experiments performed in the context of a competition are repeatable. By this means the state-of-the-art of a research field becomes publicly available, it becomes clear where progress has been made and where more work has to be done. Ideally, new research questions

are derived from the results which might then be handed over to the community in order to get solutions. These research questions may be documented in terms of benchmarks which cannot be handled by current tools. Often, such benchmarks are provided from industry and allow the researchers to show that a new approach improved the state-of-the-art.

To the best of our knowledge, recently there are no community-organized evaluations and competitions for V&V approaches in the field of MBE (TTC goes in that direction), although such events would be extremely beneficial with respect to the same arguments discussed above. Their absence is somehow surprising as the research community is rather large playing a major role in all leading modeling conferences. However, there are several reasons which might explain why no V&V competitions for MBE-approaches are organized at the moment. In the following, we elaborate on three urgent problems, which have to be overcome for structured V&V research.

(i) *No common standards.* Many approaches use their own metamodel in order to focus on the language element relevant for their purposes as for example UML is too large to be completely implemented.

(ii) *No community platform.* In order to collect and document interesting benchmarks which are required for organizing a successful competition, the community needs a forum to gather relevant data as it is for example done with the TPTP repository.

(iii) *Metrics for improvements.* In automatic theorem proving the improvements are mostly measured by runtime reduction or by compactness of proofs. For V&V approaches it is not so easy to measure progress in tools: the size of encodings or the execution time could be compared as well as the size of the error traces. However, MBE tools are also confronted with other requirements like usability and this is much harder to evaluate.

### 2.8 Domain-Specific Languages

Most verification tools have quite specific, sometimes archaic and hard to use input languages that are optimized for the semantic paradigm used by the tool. These languages are often alien to standard developers and hence, verification tools are often not used. On the other hand, developers could benefit from easy-to-use verification techniques; even in non safety-critical domains, verification can be an additional means of quality assurance, even when it is not required by industry standards. Development approaches based on domain-specific languages (DSLs) are becoming more and more mainstream, through code generation this approach leads to improved productivity. Models expressed with DSLs also have the potential of simplifying analysis and verification, because of the higher degree of domain semantics they express. There seems to be a potential to exploit the two approaches synergistically: from the high-level models, we can automate the generation of the input to the verification tools.

## 3 Research Questions

In this section we list the research questions that emerged from the discussions of our working group. The purpose of the list is to document the important issues that, we believe, research in the intersection of MBE and V&V must address.

**Gap Existing between Models and V&V Formalisms:** How do we express properties at the level of models in a way understandable to clients? How do we formulate models and properties in a single language transparent to clients? How do we report the V&V results and diagnostics in an appropriate form to clients? How do we bridge the gap between formally expressed and verified properties on one side and client attention on the other side? Can modeling language extensions help in making explicit the needs of V&V machines?

**Need to Refine Existing Methodologies:** How do we integrate V&V in the overall development and modeling process? On the technical level of tool exchange? On the methodological level of using the right technique at the right time for the right task? When are techniques like animation, execution, symbolic evaluation, testing, simulation, proving or test case generation used efficiently during development?For which model and model transformation properties can they be employed?

**Design-time vs. Runtime:** How do we obtain during the V&V phase an initial model instantiation on the model runtime level which is determined by the model design time description? How do we obtain large and meaningful instantiations? How do we connect design time and runtime artifacts? How do we deal with the scalability issue in the context of V&V ? How do we handle time and space concerns wrt design time and runtime artifacts? How do we automatically or semi-automatically manage the V&V machine configuration?

**Properties:** How do we handle model and model transformation properties relevant in V&V like consistency, reachability, dependence, minimality, conformance, safety, liveness, deadlock freeness, termination, confluence, correctness? How do we search for such properties in models and model transformations? What are the benefits and tradeoffs between expressing these properties on more abstract modeling levels in contrast to expressing them on more concrete levels? How do we find the right techniques for uncovering static and dynamic model properties? Which techniques are appropriate for uncovering static modeling language inherent properties, which for static model-specific properties? Which techniques are appropriate for uncovering dynamic generic properties, which for dynamic model-specific properties? Which high-level features are needed in the property description language in order to query and to determine modeling level concepts?

**Model Transformation:** What verification techniques are meaningful for verifying model transformations? How do we analyse properties like confluence and termination for transformations which are composed from transformation units? How do we analyse correctness of model transformations wrt a

transformation contract? How do we infer a transformation contract from a model transformation?

**Informal vs. Formal vs. Incomplete Modeling:** How do we leverage informal assumptions found in sketches for exploratory V&V? Are informal sketches close enough to V&V at all? What are appropriate relaxation mechanisms for different degrees of formality? How do we handle incomplete or partial models wrt V&V? How do we deactivate and activate model units? How do we handle the exploration of model properties and alternatives?

**Comparison and Benchmarking:** How do we compare existing V&V tools employed for modeling wrt functionality, coverage, scalability, expressiveness, executing system (i.e., for models at runtime)? Which criteria are appropriate for comparison? Can the broad and diverse spectrum of V&V machines (like B, Coq, HOL/Isabelle, SAT, SMT, CSP solvers, Relational logic and enumerative techniques) be globally compared in a fair way at all?

**Domain-Specific Languages:** How can DSLs be defined so that they are close to the domain concepts on the one hand, but still allow the generation of meaningful input files for verification tools? How do we express the properties to be verified at the domain level in a user-friendly way? Can the property specifications be integrated with the same DSL and/or model used for describing the to-be- verified system without creating self-fulfilling prophecies? How can we lift the result of a verification (e.g. an example program execution that demonstrates the failure) back to the domain level and express it in terms of the DSL-level input? Can incremental language extensions help with making programs expressed in general-purpose languages more checkable? For example, the semantics of a specific extension construct may enable the generation of very rich inputs to the verification tool, which otherwise may have to be specified manually (program annotations or properties)?

## 4 Conclusion

On this paper we present some observations and list the research questions that emerged from them. This list of research questions spans a wide area of themes where MBE and V&V intersect: specification and feedback and its impact on stakeholder collaboration, development process, design-time vs runtime, properties, model transformations, informal vs formal vs incomplete modeling, comparison and benchmarking, and domain-specific languages.

Our hope is that the set of observations and questions presented here will spark further discussions and thus aid the community focus research on those areas where the synergy of MBE and V&V can yield the greatest benefits.

## References

1. El Arbi Aboussoror, Ileana Ober, and Iulian Ober. Seeing Errors: Model Driven Simulation Trace Visualization. In *Proc. of the Int. Conf. on Model Driven Engineering Languages and Systems (MODELS'12)*, volume 7590 of *LNCS*, pages 480–496. Springer, 2012.

2. Moussa Amrani, Levi Lucio, Gehan Selim, Benoit Combemale, Jürgen Dingel, Hans Vangheluwe, Yves Le Traon, and James R. Cordy. A Tridimensional Approach for Studying the Formal Verification of Model Transformations. In *Proc. of the IEEE Fifth Int. Conf. on Software Testing, Verification and Validation (ICST'12)*, pages 921–928. IEEE Computer Society, 2012.
3. Bruno Barroca, Levi Lúcio, Vasco Amaral, Roberto Félix, and Vasco Sousa. DSLTrans: A Turing Incomplete Transformation Language. In *Software Language Engineering*, volume 6563 of *LNCS*, pages 296–305. Springer, 2011.
4. Benoit Baudry, Trung Dinh-trong, J.-M. Mottu, Devon Simmonds, Robert France, Sudipto Ghosh, Franck Fleurey, and Yves Le Traon. Model transformation testing challenges. In *Proc. of the IMDT workshop @ ECMDA06*, 2006.
5. Benoit Baudry, Sudipto Ghosh, Franck Fleurey, Robert France, Yves Le Traon, and Jean-Marie Mottu. Barriers to systematic model transformation testing. *Commun. ACM*, 53(6):139–143, June 2010.
6. Gábor Bergmann, István Ráth, Gergely Varró, and Dániel Varró. Change-driven model transformations - change (in) the rule to rule the change. *Software and System Modeling*, 11(3):431–461, 2012.
7. Pieter Van Grop Christian Krause, Louis Rose. Transformation tool contest 2013. `http://planet-sl.org/ttc2013`.
8. Edmund M. Clarke, E. Allen Emerson, and Joseph Sifakis. Model checking: algorithmic verification and debugging. *Commun. ACM*, 52(11):74–84, 2009.
9. Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of algebraic graph transformation*, volume 373. Springer, 2006.
10. Sebastian Gabmeyer, Petra Brosch, and Martina Seidl. A Classification of Model Checking-Based Verification Approaches for Software Models. In *Proceedings of VOLT'13*, 2013.
11. Ismênia Galvão and Arda Goknil. Survey of traceability approaches in model-driven engineering. In *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, pages 313–326. IEEE Computer Society, 2007.
12. Martin Gogolla, Fabian Büttner, and Mark Richters. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming*, 69:27–34, 2007.
13. Matti Järvisalo, Daniel Le Berre, Olivier Roussel, and Laurent Simon. The International SAT Solver Competitions. *AI Magazine*, 33(1), 2012.
14. Jochen M. Küster. Definition and validation of model transformations. *Software and Systems Modeling*, 5(3):233–259, 2006.
15. Mounira Kezadri and Marc Pantel. First steps toward a verification and validation ontology. In *Proc. of Int. Conf. on Knowledge Engineering and Ontology Development (KEOD'10)*, pages 440–444, 2010.
16. Perdita Stevens. Generative and transformational techniques in software engineering ii. chapter A Landscape of Bidirectional Model Transformations, pages 408–424. Springer, 2008.
17. Jan Oliver Ringer Udo Kelter, Piet Pietsch. Comparison and versioning of software models. `http://pi.informatik.uni-siegen.de/CVSM2013/`.
18. Antonio Vallecillo, Martin Gogolla, Loli Burgueño, Manuel Wimmer, and Lars Hamann. Formal specification and testing of model transformations. In *Proc. of the 12th Int. Conf. on Formal Methods for the Design of Computer, Communication, and Software Systems: formal methods for model-driven engineering (SFM'12)*, pages 399–437. Springer, 2012.

# An Approach to Analyzing Temporal Properties in UML Class Models

Mustafa Al-Lail, Ramadan Abdunabi, Robert B. France, and Indrakshi Ray
Colorado State University
Computer Science Department
{mustafa, rabdunab, france, iray}@cs.colostate.edu

**Abstract.** The Unified Modeling Language (UML) Class Models are widely used for modeling the static structure of object-oriented software systems. Temporal properties of such systems can expressed using TOCL, a temporal extension to the Object Constraint Language (OCL). Verification and validation of temporal properties expressed in TOCL is non-trivial and there are no automated tools that can aid such analysis. Existing approaches rely on transforming the UML models to another language that supports automated analysis. Such transformation is complex and can introduce errors. Towards this end, we propose an approach for directly analyzing temporal properties expressed in TOCL. We present a case study based on the Steam Boiler Control System to demonstrate the applicability of the approach.

**Keywords:** Analysis, Verification, Class Model, Temporal Properties

## 1 Introduction

The Unified Modeling Language (UML) Class Models are probably the most common specification diagrams used in the software industry. Automated analysis of class models often uncovers design problems. Detecting design problems in a timely manner saves time and effort. Specifying and analyzing temporal properties in class models are non-trivial. Consider the following temporal property in the Steam Boiler Control System [1]: "when the system is in the initialization mode, it remains in this mode until all physical units are ready or a failure of the water level measurement device has occurred." It is hard to express such property using Object Constraint Language (OCL) in a class model. TOCL [2], however, is a temporal logic extension of OCL and can specify such temporal properties. Once a property is specified, the class model must be analyzed to check for the satisfaction of such properties. To the best of our knowledge, we are unaware of any class model-based techniques for directly analyzing TOCL properties.

There are a number of model-checking based techniques for specifying and analyzing temporal properties in UML behavioral models, such as state machines and activity diagrams (e.g., see [3, 4]). These techniques involve developing an exogenous transformation, in which the source and target models are expressed

in different languages. Typically, the UML behavioral models are transformed to languages that are supported by model checking tools. There are three major challenges associated with these approaches: (1) effective use of these heavy-weight techniques requires developers to have specialized skills, (2) one has to prove that the transformations preserve the semantics of the source UML models, and (3) the results of the analysis performed by the back-end analysis tool must be presented to developers in UML terms, thus requiring another exogenous transformation.

However, temporal properties can also be expressed in class models that must be subsequently verified. One option is to transform them into other languages supporting automated analysis, as is done for the temporal properties specified on the behavioral models. But such an approach will have similar problems to those mentioned earlier. Another option is to develop model-checking support for verifying TOCL properties in UML class models with operation specifications. Given the complex state spaces that have to be codified and analyzed, this is a very challenging research problem.

Existing tools of UML/OCL such as USE [5] and OCLE [6] can be used to analyze structural properties, but they provide little support for temporal analysis. For example, the USE tool allows a user to interactively simulate the behavior of an operation by entering commands that change the states of objects and then the user checks if the operation's postconditions hold. Interactive simulation of operation behavior is useful, but can be tedious, time-consuming, and error-prone when manually simulating a scenario involving many interactions. Towards this end, researchers have demonstrated how scenarios can be modeled as a sequence of snapshots, which, in turn, can be verified using USE and OCLE [7]. However, adapting such an approach for verifying temporal properties is still an ongoing challenge. Our current work aims to fill this gap.

In this paper we propose a lightweight class model-based analysis approach that checks temporal properties against a non-exhaustive set of behavioral scenarios. The approach neither requires the use of exogenous transformations nor specialized skills other than those related to UML modeling. Our approach builds upon our previous work [8], where we described a temporal analysis approach that leverages the USE Model Generator [9] to produce a subset of the class model state space. A TOCL property is then checked against this state space. The approach described in this paper improves upon the earlier version in the following manners. First, the new version of the approach is based on the USE Model Validator which significantly outperforms the Model Generator [9]. The Model Validator uses boolean satisfiability (SAT) solvers to perform the analysis task. This results in a larger set of behavioral scenarios that can be checked and hence increases the confidence that a temporal property holds on a class model. Second, in the previous version, a procedure for creating non-exhaustive set of scenarios is defined manually. This task is tiresome, error-prone, and can be difficult to non-experts. In the current approach, this step is automated. Lastly, a complementary step is added to make the analysis results easier to exam to find the error. We applied our approach in specifying and analyzing real temporal

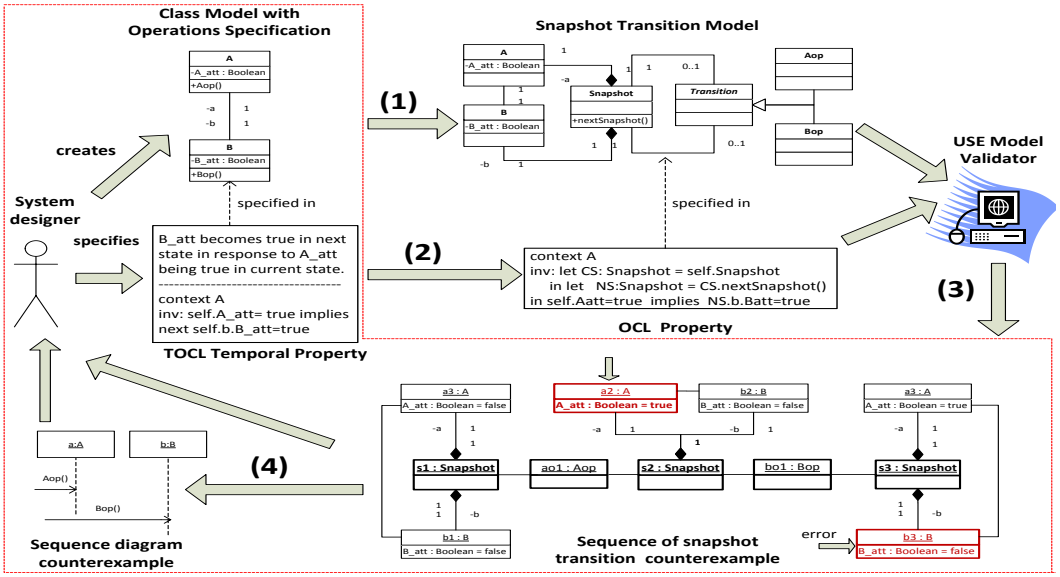


Fig. 1: An Overview of the Approach

properties of the Steam Boiler System. Note that, checking such properties is non-trivial using our earlier approach [8].

The rest of the paper is organized as follows. In Section 2, we give an overview of the proposed analysis approach. Section 3 presents the specification Steam Boiler System properties and Section 4 illustrates the analysis of these properties. In Section 5, we discuss related work, and in Section 6 we summarize our contributions and give pointers to future directions.

## 2 An Overview of the Approach

The research that led to this approach focused on answering the following question: "Given a UML class model, and a temporal property, is there a scenario supported by the class model that violates the property?." Figure 1 presents an overview of the approach. At the front-end of the approach, a system designer is responsible for 1) creating a design class model, and 2) specifying a temporal property in TOCL. A class model specifies application states and includes OCL specifications of operations. Then, the USE Model Validator is used at the back-end to generate behavioral scenarios against which the temporal property is checked. The tool produces a *scenario*, an object diagram of snapshot transition, that violates the temporal property. The back-end processing is transparent to the system designer.

The approach consists of four major steps. A transition-based class model of behavior is produced in Step 1. The model, called a *Snapshot Transition*

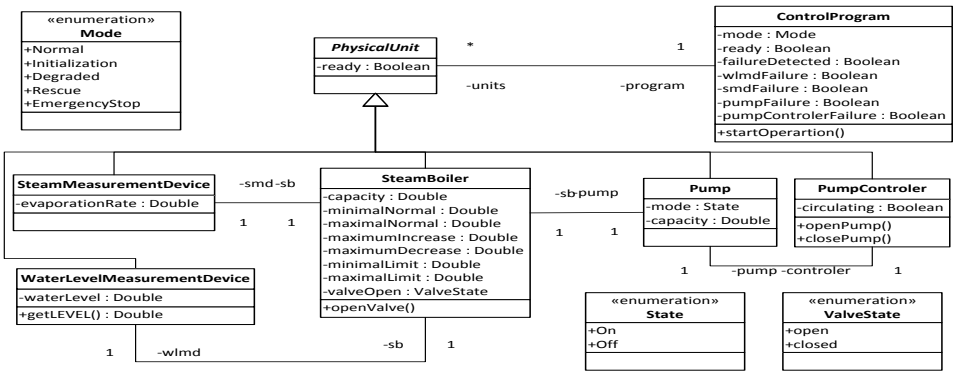


Fig. 2: The Design Class Model for the Steam Boiler Control System

*Model* ($STM$), is a class model that characterizes the valid sequences of state transitions caused by executions of operations specified in the class model. A state is modeled as a configuration of objects called a snapshot. The $STM$ is mechanically generated from the class model.

In Step 2, the temporal property to be checked is converted to an OCL property defined in the context of the $STM$. The temporal property is specified in TOCL, a temporal logic extension to OCL [2]. The TOCL property and its OCL representation are instances of temporal property specification patterns that enable the UML modelers to apply reusable solutions to specify temporal properties in object-oriented notation, more details in our paper [8].

In Step 3, the USE Model Validator tool is used to produce instances of the $STM$ (scenarios) and check the $STM$ constraint generated in Step 2 against the scenarios. Specifically, the tool checks if there is a scenario that violates the temporal property.

The analysis results for scenarios that have a large number of snapshots and transitions might be difficult to interpret. For ease of examining, this result is also visualized by a UML sequence diagram in Step 4.

## 3 The Steam Boiler Control System Problem

We use the Steam Boiler Control System described in [1] to illustrate the proposed approach. The system works correctly when the water level is within two normal limits (minimalNormal and maximalNormal) and can not pass over two critical limits (minimalLimit and maximalLimit). Otherwise the steam-boiler can be seriously damaged.

Figure 2 shows a design class model of the Steam Boiler Control System.

The class model has five operations that change the state of the system. The operation getLEVEL() reads the water level and stores it in the variable waterLevel and getSTEAM() reads the evaporation steam rate and writes it in the

Table 1: TOCL and OCL specification of the steam boiler temporal properties

| Temporal Property | Pattern | TOCL Specification on Class Model | OCL Specification on the Snapshot Transition Model |
|---|---|---|---|
| **TP1**: As soon as the program recognizes a failure of the water measuring device unit it goes into the rescue mode. | Response-global | **context** *ControlProgram*<br>**inv:** *self.wlmdFailure implies next self.mode=# Rescue* | **context** *ControlProgram*<br>**inv:** *inv: let CS: Snapshot= self.snp in NS: Snapshot= CS.getNext()*<br>**in** *self.wlmdFailure implies NS.program.mode= # Rescue* |
| **TP2**: Failure of any physical units except the water measuring device puts the program into degraded mode | Response-global | **context** *ControlProgram*<br>**inv:** *(smdFailure or pumpFailure or pumpControlerFailure) implies next self.mode=# Degraded* | **context** *ControlProgram*<br>**inv:** *let CS: Snapshot= self.getCurrentSnapshot()*<br>in let NS: Snapshot = CS.$getNext()$<br>**in** *(self.pumpControlerFaiure or self.pumpFailure or self.smdFailure) implies NS.program.mode =# Degraded* |
| **TP3**: If the water level is risking to reach one of the limit values (e.g., greater than maximalNormal or less than minimalNormal) the program enters the mode emergency stop. | Response-global | **context** *SteamBoiler*<br>**inv:** *(self.wlmd.waterLevel > self.maximalNormal or self.wlmd.waterLevel < self.minimalNormal) implies next self.program.mode = # EmergencyStop* | **context** *SteamBoiler*<br>**inv:** *let CS: Snapshot = self.snp*<br>in let NS: Snapshot = CS.$getNext()$<br>**in** *(self.wlmd.waterLevel > self.maximalNormal or self.wlmd.waterLevel < self.minimalNormal) implies NS.program.mode = # EmergencyStop* |
| **TP4**: when the valve of the steam boiler is open, then eventually the water level will be lower or equal to the maximal normal level. | Response-global | **context** *SteamBoiler*<br>**inv:** *self.valveOpen = # open implies sometime (self.wlmd.waterLevel < = maximalNormal)* | **context** *SteamBoiler*<br>**inv:** *let CS: Snapshot = self.snp*<br>in let FS: Set(Snapshot) = CS.$getPost()$<br>**in** *self.valveOpen = # open implies FS → exists (s:Snapshot \| s.WLMD.waterLevel < = maximalNormal)* |
| **TP5**: when the program is in the initialization mode and a failure of the water level measurement device is detected it puts the program in the emergency stop mode. | Response-global | **context** *ControlProgram*<br>**inv:** *(self.mode = # Initialization and self.wlmdFailure ) implies next self.mode =# EmergencyStop* | **context** *ControlProgram*<br>**inv:** *let CS: Snapshot = self.snp in let NS: Set(Snapshot) = CS.getNext()*<br>**in** *(self.mode = # Initialization and self.wlmdFailure) implies NS.program.mode = # EmergencyStop* |
| **TP6**: when the system is in initialization mode, it remains in this mode until all physical unites are ready or a failure of the water level measurement device has occurred. | Universality-between Q and R | **context** *ControlProgram*<br>**inv:** *self.mode = # Initialization implies always self.mode = # Initialization until (PhysicalUnit.allInstances→ forAll( u: PhysicalUnit \| u.ready))* | **context** *ControlProgram*<br>**inv:** *let CS: Snapshot = self.snp in let $FS_1$: Snapshot = CS.getPost() → select(s:Snapshot \| s.boiler.ready and s.SMD.ready and s.pump.ready and s.PC.ready and s.WLMD.ready)→ first() in let $PreFS_1$=Set(Snapshot) = $FS_1$.getPre() in let BTS: Set(Snapshot)=$PreFS_1$ → excluding(CS.getPre())*<br>**in** *self.mode = # Initialization implies BTS → forAll (s1:Snapshot \| s1.program.mode= # Initialization)* |

vaporationRate variable. The openPump(),closePump(), and openValve() operations open the pump, close the pump, and open the boiler valve, respectively. The OCL specifications of getLEVEL() and openPump() are defined bellow.

```
context WaterLevelMeasurementDevice::getLEVEL(): Double
pre: self.program.mode= #Normal
post: self.waterLevel = result

context PumpControler::openPump()
pre: self.pump.mode = # Off
post: self.pump.mode = # On
```

The system has a number of temporal requirements that need to be verified. We resorted to the use TOCL to specify the temporal properties in the boiler system. Table 1 presents the TOCL specifications of some of these properties. In our previous paper [8], we explain how to use reusable solution patterns to specify temporal properties in TOCL.

# 4 Case Study: Specifying and Analysing Temporal Properties of Steam Boiler

The following discusses the steps in Figure 1 in the context of the Steam Boiler Control System.

## 4.1 Step1: Generation of the *STM*

Step 1 takes the steam boiler class model (see Fig. 2) as input and produces a $STM$ model [7]. The $STM$ model characterizes a sequence of state transitions of the boiler system, where each transition is triggered by an operation invocation. The $STM$ is formed by (1) creating a Snapshot class, (2) creating a hierarchy of transition classes representing operation invocation, and (3) converting operation specifications to invariants of the transition classes. Everything else ( class invariants, associations ect.) remains intact in the $STM$ model. Figure 3 shows the $STM$ model that is produced from the boiler class model.

Each instance of the Snapshot class represents a state in a transition system. A snapshot is a configuration of one object of each of the concrete classes inf Figure 2. In this system, a snapshot has only one object of each class. The approach can also be used to specify snapshots that have many objects of each class, and it distinquishes between these objects using identifiers [7].

To create the hierarchy of transition classes, we generate a subclass of the abstract $Transition$ class from each operation. In the Steam Boiler class model, we only consider five modifier operations and thereby we create five subclasses of the class $Transition$, one for each operation (see Fig.2 and Fig. 3). For each parameter of an operation, we generate two references (shown as attributes of the Transition subclasses) that represent the value of the parameter before and after the execution of the operation. In boiler class model, none of the operations has a parameter, so we do not create any references. We define two references for each operation that point to the object's states before and after an operation invocation. A reference is also created for the return value of an operation.

We define the before and after state conditions in the $STM$ as invariants based on the pre and post conditions of the operations in the initial class model.

The following illustrates how the getLevel() operation in the WaterLevelMeasurmentDevice class is defined in the $STM$ model. We generate two references (wlmdPre and wlmdPost of type WaterLevelMeasurmentDevice) that point to the object that the operation is invoked on. Because this operation has a return value of type Double, a ret:Double reference is created to point to the returned value of the operation. The pre and post conditions of getLevel() operation ( presented above) are converted to invariants in $STM$ as follows:

```
context WaterLevelMeasurmentDevice_getLevel
inv: self.wlmdPre.program.mode=# Normal
inv: wlmdPost.waterLevel= ret
```

Similarly, we generate invariants from the pre and post conditions for all the other operations.

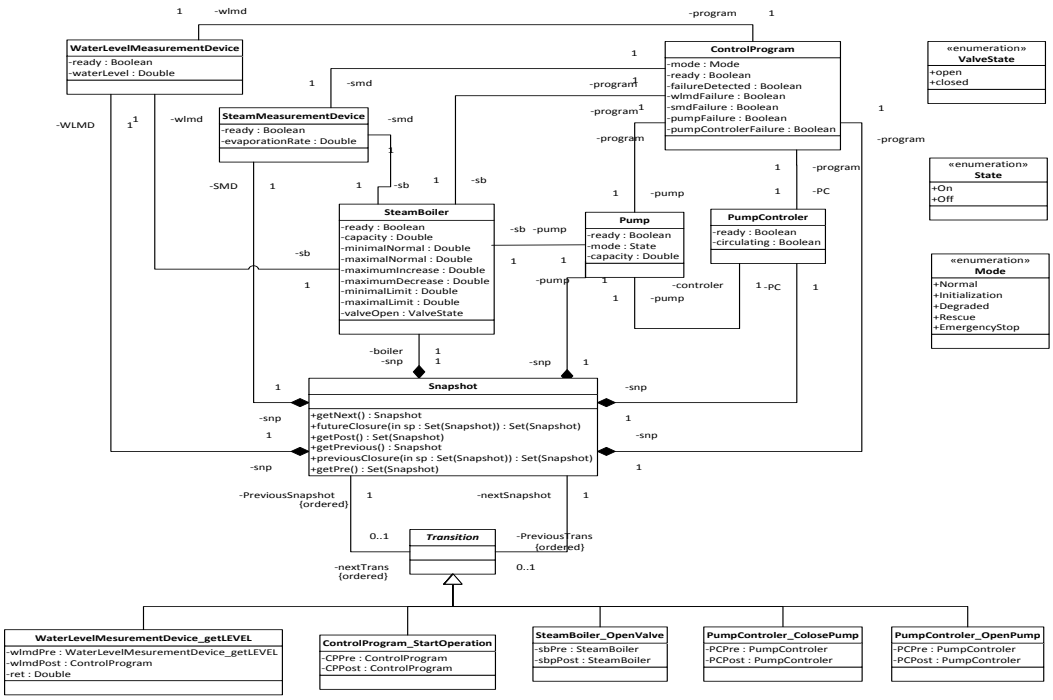


Fig. 3: The Steam Boiler Snapshot Transition Model

### 4.2 Step2: Converting TOCL to OCL properties

In this step, the steam boiler class model is unfolded as a sequence of snapshot transitions represented by the $STM$ in order to express TOCL properties as OCL constraints. We first specify the temporal properties in the Steam Boiler Control System using TOCL. Table 1 shows some of the boiler system TOCL properties. These TOCL properties are specified in the context of the steam boiler class model. Then, OCL constraints are systematically produced in the context of the steam boiler $STM$ model (see Fig. 3) using these TOCL properties. Each OCL constraint captures the semantics of the corresponding TOCL constraint in the context of the $STM$ model.

Consider the TOCL and OCL expressions of the temporal property TP1 in the Table 1. The TOCL states that if the water measuring device fails *(self.wlmd Failure=true)* then the program goes into the rescue mode ($nextself.mode = \#Rescue$). In the corresponding OCL expression, the next state ($NS$) is returned by first getting the current *snapshot(CS)* and navigating to the next state by the operation *getNext()*. Then the OCL asserts that if the water measuring device fails, then the program in the next state is in the Rescue mode.

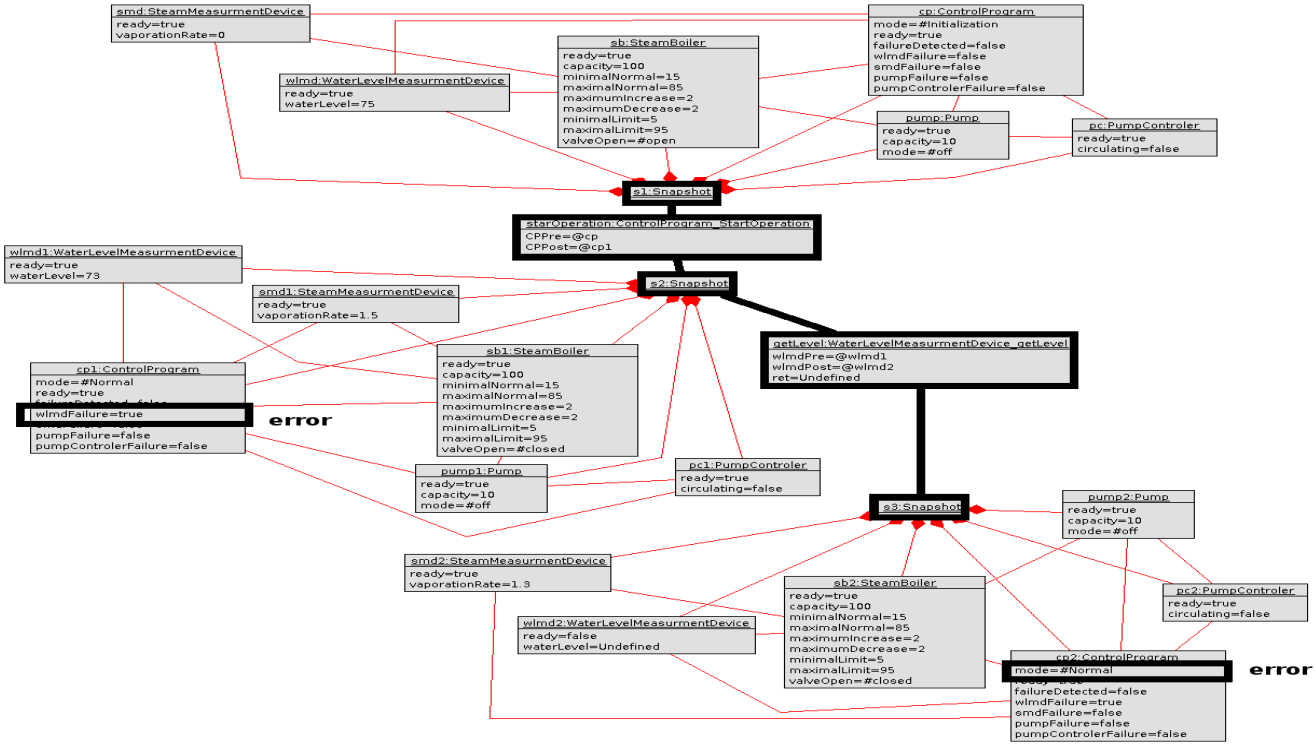


Fig. 4: Counterexample: Scenario violating the temporal property TP1

### 4.3 Step 3: Analysis

Now, we apply a UML/OCL structural analysis tool to perform the analysis task of the boiler system temporal properties. In this case study, we used the USE Model Validator to check the temporal property expressed in OCL in the steam boiler $STM$ model. For each property, the Validator attempts to produce a scenario (i.e., an instance of the $STM$) that violates the property. The Validator takes the $STM$ model and an OCL property and provides a relational logic specification. Then the tool employs of-the-shelf SAT solvers to check if there exist an instance of the $STM$ model that violates the OCL expression.

The approach checks a property based on the small-scope hypothesis [10]. That is, when a property does not hold in a model, it is more likely that there is a small scenario that violates the property. Therefore, the approach does not enumerate all possible scenarios, but a constrained number. A scope restricts the number of instances that each class can have in a snapshot and limits the number of transitions in a scenario. As such, the Model Validator enumerates all possible scenarios within the defined scopes and check the given property.

We analyzed the temporal properties in Table 1 on scopes that have one object of each class and 10 transitions. The Validator uncovered a scenario that

**Algorithm 1** Snapshot Transitions to Sequence Diagram

**Input:** Sequence of Snapshot Transition
**Output:** Sequence diagram
Algorithm Steps: For every object of the Class transition do
**Step 1.** Get the class name and the operation name that associated with transition.
**Step 2.** Get the object on which the operation is invoked on.
**Step 3.** Get the operation parameters from the transition object attributes.
**Step 4.** Get the return value from the ret attribute of the transition object.
**Step 5.** Draw a timeline for the object that the operation is invoked on from step 2.
**Step 6.** Draw an operation invocation on the object using the name of the operation and its attributes from steps 1, and 3 above .
**Step 7.** Draw a return message of the operation with the value from step 4

violates the first temporal property in Table 1, TP3. Figure 4 shows the counterexample that violates property TP1. To uncover the fault, the verifier must examine the counterexample.

### 4.4 Step 4: Sequence diagram extraction

The results of Step 3 might be complicated and difficult to present and examine. In this step, we provide support for extracting a sequence diagram from a sequence of snapshot transition. Algorithm 1 provides a systematic way to achieve this objective. We do not show the generated sequence diagram for the lack of space.

## 5 Related Work

A number of model-checking based techniques have been proposed for specifying and analyzing temporal properties in UML behavioral models, such as statemachines and activity diagrams (e.g., see [3, 4]). In order to apply such techniques, the UML models must be transformed to the tool-specific input languages. For example, the vUML [3] tool automatically converts UML statemachines to PROMELA specifications and then invokes SPIN model checker to verify the desired properties. Although the system is modeled as UML statemachines, the temporal properties are specified in LTL, but not in the UML notation. Eshuis [4] applied symbolic model checking to analyze the data integrity constraints of UML activity diagram and class models. The activity models are formalized and transformed to the input language of the NuSMV model checker. Unlike these techniques, the analysis approach described in this paper neither requires transformation nor requires that the verifier be familiar with notations other than UML and TOCL/OCL.

UML/OCL analysis tools, such as OCLE [6] and USE [11] provide support for validating structural properties. However, OCLE and USE are limited in analyzing temporal properties. The approach described in this paper enables a system designer to analyze TOCL temporal properties using OCLE and USE.

The Scenario-based Design Analysis approach [7] checks whether a given scenario is supported by a design class model. The analysis results depend on the quality of the selected scenarios, which is challenging for complex models . While this approach checks one scenario at a time, the approach in this paper builds on the Scenario-based analysis to check a temporal property within a scope of automatically generated scenarios.

## 6 Conclusions and Future Work

In this paper, we proposed a lightweight and rigorous approach that uses UML notations for specification and analysis of temporal properties without the need for transformation. The use of TOCL object-oriented temporal logic with specification patterns makes the approach accessible to UML modeling community. As a pointer to future work, we plan to provide a system-development process through which a system designer is able to design complex systems in incremental and iterative manner. Our future work also includes deploying the approach for specifying and analyzing a real-world healthcare Dengue Decision Support System (DDSS) requirements.

## References

1. Abrial, J.R., Börger, E., Langmaack, H.: The Stream Boiler Case Study: Competition of Formal Program Specification and Development Methods. In: Formal Methods for Industrial Applications. (1995) 1–12
2. Ziemann, P., Gogolla, M.: OCL Extended with Temporal Logic. In: Ershov Memorial Conference. (2003) 351–357
3. Lilius, J., Porres, I., Paltor, I.P., Centre, T., Science, C.: vUML: a Tool for Verifying UML Models. (1999) 255–258
4. Eshuis, R.: Symbolic model checking of uml activity diagrams. ACM Trans. Softw. Eng. Methodol. **15** (January 2006) 1–38
5. Gogolla, M., Bohling, J., Richters, M.: Validating UML and OCL Models in USE by Automatic Snapshot Generation. Journal on Software and System Modeling **4** (2005) 2005
6. Chiorean, D., Paşca, M., Cârcu, A., Botiza, C., Moldovan, S.: Ensuring UML Models Consistency Using the OCL Environment. Electron. Notes Theor. Comput. Sci. **102** (November 2004) 99–110
7. Yu, L., France, R.B., Ray, I., Ghosh, S.: A Rigorous Approach to Uncovering Security Policy Violations in UML Designs. In: ICECCS. (2009) 126–135
8. Al-Lail, M., Abdunabi, R., France, R., Ray, I.: Rigorous Analysis of Temporal Access Control Properties in Mobile Systems. In: ICECCS. (July 2013)
9. Soeken, M., Wille, R., Kuhlmann, M., Gogolla, M., Drechsler, R.: Verifying uml/ocl models using boolean satisfiability. In: MBMV. (2010) 57–66
10. Jackson, D.: Alloy: A Lightweight Object Modeling Notation. ACM Transactions on Software Engneering Methodology **11**(2) (2002) 256–290
11. Gogolla, M., Büttner, F., Richters, M.: USE: A UML-based Specification Environment for Validating UML and OCL. Sci. Comput. Program. **69**(1-3) (2007) 27–34