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.

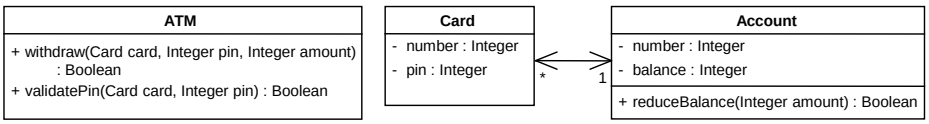

Fig. 1: Classes of the ATM system

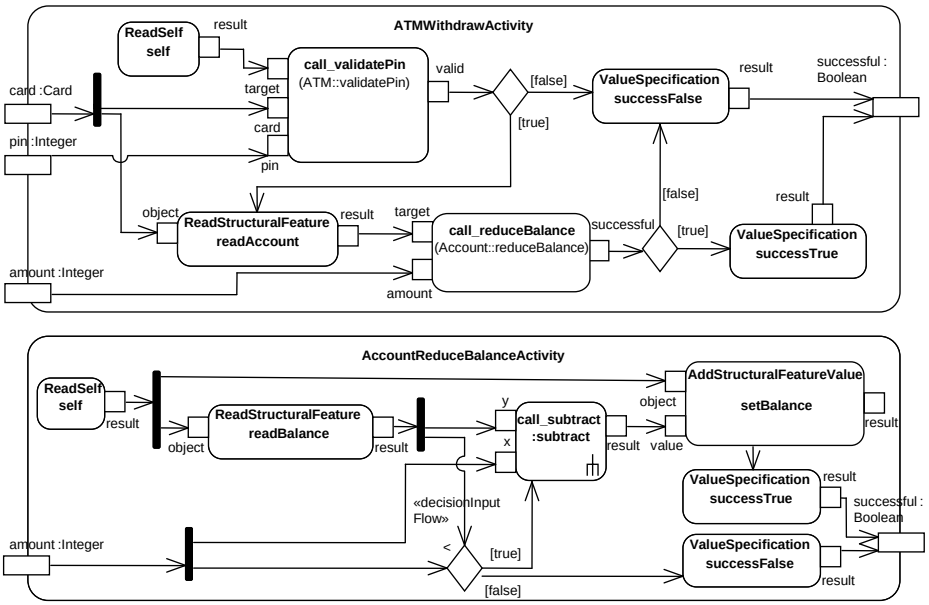

Fig. 2: Activities of the ATM system

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/`

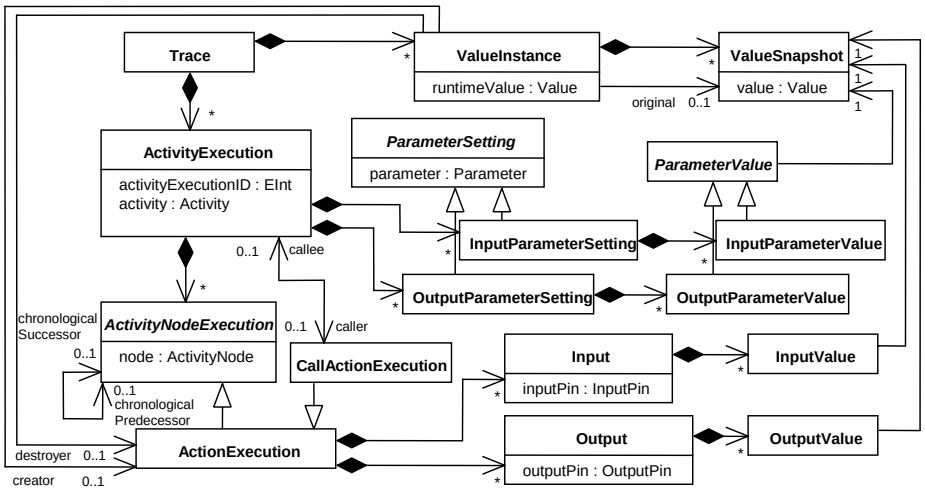

Fig. 3: Trace metamodel for fUML

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

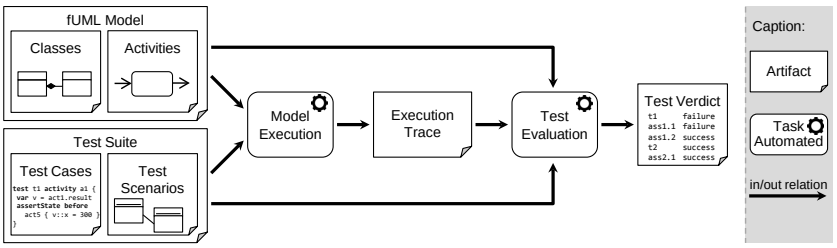

Fig. 4: Test interpreter

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.