Testing of UML Activity Diagrams

Stefan Mijatov

Vienna PhD School of Informatics
Vienna University of Technology, Austria
mijatov@big.tuwien.ac.at

Abstract. In model-driven development, modeling languages provide the means for software development on a higher level of abstraction than traditional general purpose languages. However, compared to general purpose languages, these modeling languages often lack the proper tool support, such as tools for debugging and testing. Especially testing is essential to achieve a high quality of the final software product. In this paper, we present an approach for testing UML models on the model level to ensure the validation of their quality before the executable code is produced out of these models. In particular, we propose a dedicated testing language for specifying and executing test scenarios of UML activity diagrams based on OMG's fUML standard.

1 Problem

Modeling languages can be used by software developers for writing a program using concepts that are at a higher level of abstraction than with general purpose programming languages (GPLs) [1]. Using state-of-the-art techniques in the realm of model-driven development (MDD), the executable code of a software system can be produced automatically using model-to-code transformations.

Once the code is produced from models, any existing error in the models gets transferred to the code and hence becomes much more expensive to detect and correct there. Thus, adequate means for detecting and correcting errors already on the model level are necessary to improve the development process when using a model-driven approach. To ensure a high quality of models, they have to be tested continuously, as is done in code-centric software development.

In the context of GPLs, several kinds of testing approaches are adopted, such as unit testing, integration testing, and system testing. Especially unit testing is frequently adopted in practice. Unit tests evaluate assertions concerning the expected output and result of an operation invocation of some part of the software program. Specified tests can be executed continuously during the development life cycle to ensure that the introduction of new functionality into the system or re-factoring of existing code has not caused a bug somewhere else in the system. Unfortunately, in the domain of model-driven software development, unit testing of models has not gained enough attention yet.

The Unified Modeling Language (UML) [11], a standard by the Object Management Group (OMG), constitutes the most popular modeling language for specifying structural and behavioral aspects of a software system [5]. However,

the specification of the operational semantics of UML, which is a prerequisite for testing UML models unambiguously, is scattered across available documentation expressed in natural English language, which leads to different interpretations of the same specification by different subjects. This issue has been addressed recently with the introduction of the OMG standard called Foundational Unified Modeling Language (fUML) [10], which defines the operational semantics for a subset of UML. This subset contains the most relevant part of class diagrams for modeling the data structure and activity diagrams for specifying the behavior of the system.

Based on this precise and standardized specification of the operational semantics, we aim at addressing the lack of testing facilities for UML models. In particular, we plan to adopt the concepts of unit testing for empowering users to maintain a high quality of UML activity diagrams systematically and continuously during the modeling process. Therefore, we propose to develop a dedicated testing language for specifying test cases for activity diagrams. We argue that such a language may improve efficiency and effectiveness when specifying test cases, compared to using JUnit or similar code testing frameworks. Note that this work is part of the research project *Moliz*[1], which aims at providing an observable virtual machine for executing UML models based on fUML.

2 Related Work

In order to realize unit testing for UML models, the means for executing them is essential. Therefore, we first review the state of the art in model execution before we discuss related work in the area of model testing.

Model execution. Approaches for specifying the operational semantics to realize executable models can be assigned to one of the following three categories [3].

In a *rule-based approach* the semantics is specified using a rewriting system. Such a system consists of a set of rewriting rules. The execution of a rewriting system is based on the repeated application of the rewriting rules to an existing configuration, such as a model. Wachsmuth et al. [15] present an approach where such a transition system is completely based on the modeling means standardized by OMG. Metamodels are used to model configuration sets and model transformations are used to specify transition relations.

The mapping approach represents an approach where the concepts of a modeling language are mapped to respective concepts of a formal language for which a specified semantics is available. Wu et al. describe this kind of approach in [16] where an existing GPL debugger is reused to build a DSL debugging framework. Therefore, a mapping of the correspondences between the DSL and the generated GPL code is produced, which is composed of a source code mapping, a debugging methods mapping and a debugging results mapping.

The imperative approach is realized by providing a virtual machine for executing the models. Fitting in this category, Scheidgen and Fisher [14] present an

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

approach where operations of classes from the abstract syntax of a DSL are specified using activity diagrams. Thus, the authors combine MOF, OCL and UML activities to create models that cover the abstract syntax, runtime configurations and the behavior of runtime elements.

Model testing. When it comes to testing in the context of MDD, we can distinguish between two different concepts: *model-based testing* and *model testing*. Model-based testing is an approach where test cases and executable test scripts are automatically generated from models describing behaviour of the system under test (SUT) [2]. Grigorijevs [4] presents the generation of test models from UML sequence diagrams applying the standard UML testing profile [13].

In contrast to model-based testing, in this work model testing is defined as an approach to specify tests that are used to validate the quality of the *models themselves*. For this approach, model execution is essential producing model execution traces as sequences of executed steps and sequences of object models representing system states after each executed step. Wu et al. describe an approach to testing and debugging in the context of DSLs [16]. The idea is based on a translator which generates GPL code from DSL code, generates GPL unit test cases from DSL test cases, and generates the source code mapping information for both GPL and GPL unit test generation components, such that the controls and results may be mapped between DSL and GPL.

The execution of models based on the fUML standard is done by an interpreter, thus, no mapping is required. However, specifying tests for UML activity diagrams based on fUML is still unclear and undefined. Thus, we aim at providing the means to define and run tests for validating such models based on fUML.

3 Proposed Solution

The overall goal of this proposal is to establish the means for testing UML activity diagrams by leveraging the precise specification of the operational semantics provided by fUML. Thereby, users should be empowered to ensure the quality of their models efficiently and continuously.

Challenges. UML can be used for sketching and planning a system, as well as for specifying a system precisely enough to generate executable code. Thus, the UML models under test may be specified at *diverse levels of abstraction*. For instance, an action within a UML activity may simply be named `Check Application`, whereas no more details are available. In contrast, users may also use low-level actions, such as `Create Object` and `Set Feature Value`. Depending on the level of abstraction, the models under test vary significantly thus impacting the specification of the test cases. Moreover, we plan to enable continuous testing of models. Thus, we need to provide techniques for testing also *incomplete and under-specified models*. Due to these characteristics of the models under test, users need to be empowered to define assertions regarding the result of executing an activity, i.e., the resulting object diagram, but also concerning the trace of the activity execution. However, it is unclear how to represent traces of activities,

how to specify the scenarios and events of an execution, and how to specify and evaluate assertions.

Testing language. Enabling to test UML activities precisely and efficiently entails the need for a proper language for specifying the test cases. From a general point of view, the envisioned structure of the test could be composed of the following. First, users may define a *test scenario* in terms of an input object model with specific values for attributes and links between objects. Next, the declaration of the test is composed of a test name, an activity under test, and optionally an event until which activity should be executed. Finally, a set of *assertions* is specified in the body of the test. Each assertion in the body of the test has an event as part of its definition that specifies where in the trace this assertion should be valid. For defining assertions concerning the result of an activity execution, we may reuse OMG's Object Constraint Language (OCL) [12], which is intended to describe additional structural constraints on elements of UML models. OCL considers only a single state of the system, described as an instance of the UML model. However, during the execution, we also have to deal with transitions between states of model instances. In order to express such behavioral constraints, extensions of OCL have to be defined. Therefore, we plan to adopt concepts of ongoing research for extending OCL with operators from linear temporal logic [17]. The basic idea of linear temporal logic is to consider not only single states, but also arbitrary state sequences. By doing so, it is possible to characterize valid system behavior by specifying the allowed sequences of system states. Operators, such as *after*, *before*, and *initially* are used to specify constraints which express temporal dependencies of different states of objects.

Architecture. A high-level view on the basic architecture of the proposed solution is given in Figure 1. As can be seen in the figure, the starting point is the specification of a test case in the aforementioned testing language. Here we may specify the initial state of the system, e.g., `given x1 and x2`, what actions are

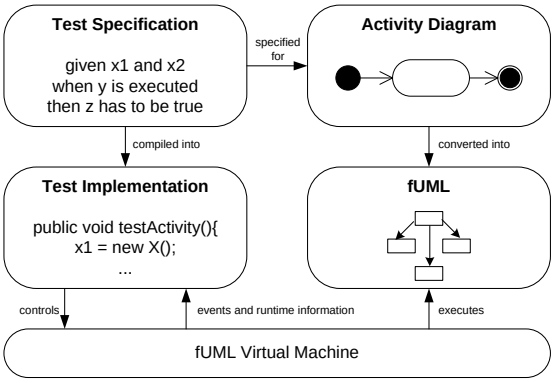

Fig. 1. Basic architecture of proposed solution

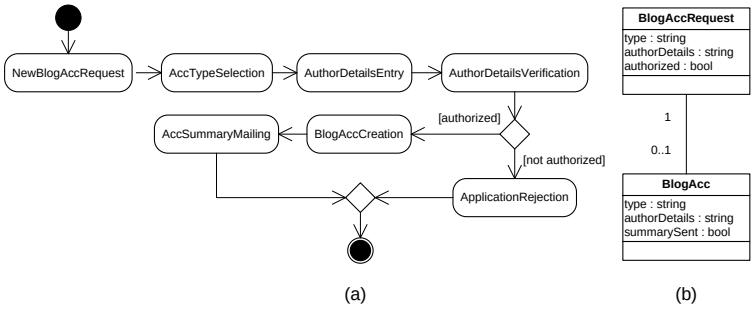

Fig. 2. Blog account creation activity and class diagram

executed and what assertions should be evaluated, as well as their dependency on the execution order of actions, e.g., `when y is executed then z has to be true`. For each test case or group of test cases, an activity diagram under test is specified. Another part of the architecture is a generator which translates test cases into an implementation of the test cases in the form of some standard programming language testing framework, such as JUnit [6]. Additionally, the UML activity diagram under test is converted into a corresponding fUML representation, which is then executed using the fUML virtual machine [8]. Controls for creation and execution of both activity diagram and the specified tests are passed onto the fUML virtual machine and assertions are made upon. Events and runtime information is returned to the testing framework to evaluate the test cases.

Example. To exemplify the envisioned solution, we present briefly an example of a blog account creation process taken from [9], which is depicted in Figure 2(a). In this process, a user creates a new blog account request, selects the type of the account to be created, and enters some details. In the next step, these details are verified. If the request is authorized, the blog account is created and a summary of the creation is sent to the user by email. If the request is not authorized, the request application is rejected and the blog account is not created.

Figure 2(b) depicts the corresponding class diagram, which describes the data that is modified during execution of the activity diagram in Figure 2(a). The class `BlogAccRequest` represents the user's request for a new blog account and contains the attributes `type`, `authorDetails`, and a Boolean value indicating whether the request was authorized. For each request, there may be at most one blog account created, if the request was authorized. The class `BlogAcc` contains the attributes `type`, `authorDetails`, and a Boolean value indicating whether a summary of the created blog account was sent by email to the user.

In Listing 1.1, we depict an example for two test cases in terms of the envisioned testing language. Please note that this is just an initial idea, presenting what possible concepts of UML and temporal OCL operators could be necessary for specifying test cases for UML activities accordingly.

Listing 1.1. Test specification for the blog account creation activity

```
test BlogAccTest1 Activity: BlogAccountCreationActivity
execute until AuthorDetailsVerification {
  initially BlogAccRequest = null and BlogAcc = null;
  BlogAccRequest != null before AccTypeSelection;
  BlogAccRequest.type != null before AuthorDetailsEntry;
  BlogAccRequest.authorDetails != null before AuthorDetailsVerification;
}

test BlogAccTest2 Activity: BlogAccountCreationActivity execute all {
  BlogAcc.authorDetails != null before AuthorDetailsVerification;
  BlogAccRequest.authorized implies next BlogAccCreation;
  BlogAcc.summarySent after AccSummaryMailing;
}
```

In the specification of the test, it should be possible to specify which activity diagram is under test and until which action the execution should be performed. Inside the body there are several assertions on the involved objects. By using the keyword `initially` we can validate the state at the beginning of the activity execution. Moreover, we may use temporal operators such as `after` and `before` to specify state changes between execution of certain actions. We should also be able to express what should be the path of execution depending on the already executed trace using the keyword `implies next` (cf. second assertion of `BlogAccTest2` in Listing 1.1).

4 Preliminary Work

As mentioned in Section 1, within the project Moliz, we work on the development of an fUML virtual machine, which allows controlling the execution of fUML activity diagrams and obtaining runtime information from that execution [7]. This represents the basis for specifying and executing tests as proposed in this paper. The testing framework will be based on model traces where sequences of executed actions and appropriate class diagrams describing data modified by those actions are stored. An approach to store and access such model traces from the testing framework is under development. In order to enable interactions between the testing framework and the Moliz fUML virtual machine, extensions that have to be introduced into the Moliz fUML virtual machine are investigated.

5 Expected Contributions

The main contribution of the proposed work is an executable testing language and environment for UML activity diagrams. With this testing language and environment, we aim at establishing the means for specifying test cases precisely and efficiently to support users in maintaining a high quality of their UML models. Therefore, we leverage the operational semantics described in the fUML standard, which constitutes a well understood and widely accepted approach to specifying the operational semantics of UML. Having the means for standardized testing of UML models on the model level, we hope to be able to foster the promised move from code-centric to model-centric development and to contribute

to establishing a more complete set of tools for model-driven development of software using the UML standard.

The expected contributions of this proposed work may be summarized as follows. We expect to contribute *(i)* a design of a dedicated testing language for UML activity diagrams, *(ii)* a development environment enabling to create test cases more efficiently using a dedicated editor providing syntax checking, syntax completion, and syntax highlighting, *(iii)* a framework for executing and validating the test cases, as well as *(iv)* the tight integration of the testing facilities in a popular modeling environment, such as the Eclipse Modeling tools.

6 Plan for Evaluation

As the goal of this proposal is concerned with creating a testing language and environment for testing UML activity diagrams precisely and efficiently, we aim at investigating the following research questions: What is the coverage in terms of expressiveness of the testing language compared to unit testing frameworks at the code level? Is it more efficient to use the testing language for specifying tests on models than using a unit testing framework at the code level? Is the learning curve when using the testing language reduced in comparison to specifying test cases using a unit testing framework at the code level?

For answering these research questions, we will perform two studies. First, we will evaluate the expressiveness of the proposed testing language by conducting a case study in which we specify a broad set of tests and assess how concise and complete these tests may be specified. Second, we plan to perform a comprehensive user study with students from our Master degree studies, who enroll at the Model Engineering course at the Faculty of Informatics, Vienna University of Technology. In particular, students will be provided with ready-made UML models and requirements for specifying tests. One group of students will be asked to fulfill the test specifications using the developed testing language and environment, and the other group will have to implement the same test specifications using a GPL and a state-of-the-art testing framework, such as JUnit. After the students fulfill these requirements, they will be asked to fill out a questionnaire about how intuitive and user-friendly it was to accomplish the tasks. Based on this data, we plan to assess the efficiency of using our proposed testing facilities in comparison to using a GPL testing framework. In a second step, we will ask the students to realize a different but comparable set of requirements. After this second experiment, students will be asked again to give their opinions regarding the learning curve when using the testing language. The gained knowledge will be the basis for further improvements of the design of the testing language and environment.

7 Current Status

The proposed research is in its initial phase. The first step is to extend the fUML virtual machine in order to realize integration of the proposed testing language and its environment. In parallel, gathering of requirements that need to be fulfilled by a testing language will be performed. The next step contains a survey of relevant technologies and literature for realizing the testing environment as well as specifying the testing language itself. Next two steps are to design the testing language and subsequently to implement the testing language environment. Finally, we plan to evaluate the developed tools as described in Section 6. The expected completion of the PhD thesis is in two years from now.

References

1. Bézivin, J. On the Unification Power of Models. *Software and Systems Modeling 4*, 2 (2005), 171–188.
2. Bouquet, F., Grandpierre, C., Legeard, B., Peureux, F., Vacelet, N., and Utting, M. A subset of precise UML for model-based testing. In *A-MOST* (2007), pp. 95–104.
3. Bryant, B. R., Gray, J., Mernik, M., Clarke, P. J., France, R. B., and Karsai, G. Challenges and directions in formalizing the semantics of modeling languages. *Comput. Sci. Inf. Syst. 8*, 2 (2011), 225–253.
4. Grigorjevs, J. Model-driven testing approach based on UML sequence diagram. *J. Riga Technical University 44* (2011), 85–90.
5. Hutchinson, J., Whittle, J., Rouncefield, M., and Kristoffersen, S. Empirical Assessment of MDE in Industry. In *ICSE* (2011), pp. 471–480.
6. Massol, V., and Husted, T. Junit in action, 2003.
7. Mayerhofer, T. Testing and debugging UML models based on fUML. In *ICSE* (2012), pp. 1579–1582.
8. Mayerhofer, T., Langer, P., and Kappel, G. A Runtime Model for fUML. In *MoDELS Workshops, Accepted for publication in 7th International Workshop on Models@run.time* (2012).
9. Miles, R., and Hamilton, K. *Learning UML 2.0*. O'Reilly Media, Inc., 2006.
10. OMG. Foundational Unified Modeling Language 1.0 (fUML). `http://www.omg.org/spec/FUML/1.0`, 02 2011.
11. OMG. Unified Modeling Language 2.4.1 (UML). `http://www.omg.org/spec/UML/2.4.1/`, 08 2011.
12. OMG. Object Constraint Language. `http://www.omg.org/spec/OCL/2.2`, 04 2012.
13. OMG. UML Testing Profile. `http://www.omg.org/spec/UTP/1.1`, 04 2012.
14. Scheidgen, M., and Fischer, J. Human comprehensible and machine processable specifications of operational semantics. In *ECMDA-FA* (2007), pp. 157–171.
15. Wachsmuth, G. Modelling the operational semantics of domain-specific modelling languages. In *GTTSE* (2007), pp. 506–520.
16. Wu, H. Grammar-driven generation of domain-specific language testing tools. In *OOPSLA Companion* (2005), pp. 210–211.
17. Ziemann, P., and Gogolla, M. An Extension of OCL with Temporal Logic. In *Critical Systems Development with UML* (2002), pp. 53–62.