Testing Functional Requirements in UML Activity Diagrams

Stefan Mijatov, Tanja Mayerhofer(✉), Philip Langer, and Gerti Kappel

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

Abstract. In model driven engineering (MDE), models constitute the main artifacts of the software development process. From models defining structural and behavioral aspects of a software system implementation artifacts, such as source code, are automatically generated using model transformation techniques. However, a crucial issue in MDE is the quality of models, as any defect not captured at model level is transferred to the code level, where it requires more time and effort to be detected and corrected. This work is concerned with testing the functional correctness of models created with a subset of UML called fUML comprising class and activity diagrams. We present a testing framework for fUML, which enables modelers to verify the correct behavior of fUML activities.

Keywords: Functional testing · UML activity diagrams · fUML

1 Introduction

In model driven engineering (MDE), models are the main artifacts of the development process. Using model transformations, code and other implementation artifacts are automatically produced from models improving the productivity of the software development process, as well as the quality, portability, and maintainability of the developed system [2]. As the development process shifts from being code-centric to being model-centric, the quality of the models used in an MDE-based software development process becomes essential. Any defect not captured at model level will be propagated to the code level, where it will require more time and effort to be detected and corrected.

This work is concerned with verifying the *functional correctness* of models created with UML [14], which is the most widely adopted modeling language in MDE. More precisely, we focus on fUML [16], which is an executable subset of UML (cf. also xUML [10]) comprising class diagrams for defining the structure of systems and activity diagrams for defining the behavior of systems. For fUML, a standardized virtual machine exists that gives precise operational semantics to the included subset of UML. The standardization of fUML's semantics provides the basis for developing model analysis techniques and tools for UML models.

In general, it can be distinguished between two main analysis techniques for verifying the functional correctness of software artifacts, namely formal analysis and testing techniques. These two techniques are not mutually exclusive,

J.C. Blanchette and N. Kosmatov (Eds.): TAP 2015, LNCS 9154, pp. 173–190, 2015.
DOI: 10.1007/978-3-319-21215-9_11

but instead complement each other. While several approaches applying formal analysis techniques on fUML have been proposed in the past, to the best of our knowledge only first ideas and intents on applying testing techniques on fUML have been published (cf. Sect. 6).

In this paper, we present a fully functional and implemented *testing framework for fUML*, which is based on first ideas and an early prototype presented in [12]. The framework comprises a test specification language, which enables modelers to express assertions on the behavior of a system defined in fUML, as well as a test interpreter, which evaluates these assertions. Besides giving an overall overview of our testing framework, we present *three newly developed testing features*. These new features address three requirements on testing fUML models: *(i)* Specifying assertions on the behavior of a system requires the capability to evaluate complex conditions on the system's runtime state, such as iterations over existing objects and calculations over their feature values. *(ii)* Temporal expressions allowing precise selections of the runtime states to be asserted are required. *(iii)* Because fUML models can be used to specify concurrent behavior, the existence of a potentially large number of possible execution paths has to be considered in the test evaluation. To address these requirements, we extended our initial testing framework with *(i)* support for OCL [15] allowing the specification of complex assertions on the runtime state of a system, *(ii)* a set of temporal operators and temporal quantifiers allowing a more precise selection of the runtime states to be asserted, and *(iii)* an improved test evaluation algorithm taking concurrent behavior into account. We evaluated our testing framework with these newly introduced features in a *user study* concerning the properties *ease of use* and *usefulness*. The evaluation results on the one hand indicate that the testing framework is both easy to adopt and useful for testing fUML models, and on the other hand enabled us to identify potential for improvement.

The remainder of the paper is structured as follows. In Sect. 2, we introduce an example for motivating and illustrating the newly developed features of our testing framework. In Sect. 3 and Sect. 4, we provide an overview of our testing framework and describe its new features in detail. The results of our user study and related work are discussed in Sect. 5 and Sect. 6, respectively. In Sect. 7, we conclude the paper and outline future work.

2 Motivating Example

In this section, we want to motivate our testing approach based on the example of an automatic teller machine (ATM) system. The structure of the ATM system is depicted in Fig. 1. The ATM can be used to perform withdrawals from a bank account. The process of performing a withdrawal (operation *ATM.withdraw*) is realized by the activity *ATM.withdraw* shown in Fig. 2. For starting a withdrawal, the user has to provide an ATM card, the pin assigned to the card, and the amount of money to be withdrawn from the user's account. Once the withdrawal is started, first a new transaction is created and set as current transaction (action *startTransaction*). Next, the provided pin is validated (action *validatePin*). If the pin is valid, the withdrawal is performed (action *makeWithdrawal*).

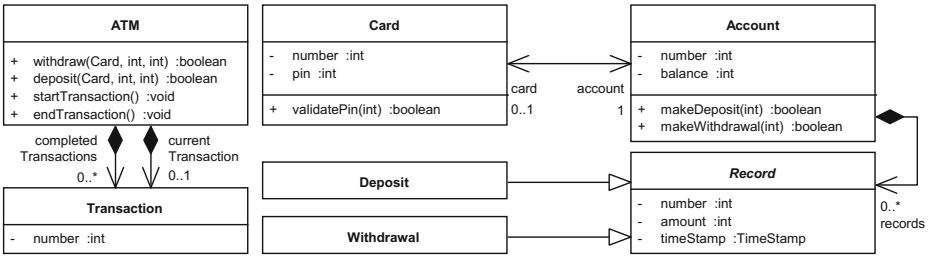

Fig. 1. Class diagram of the ATM system

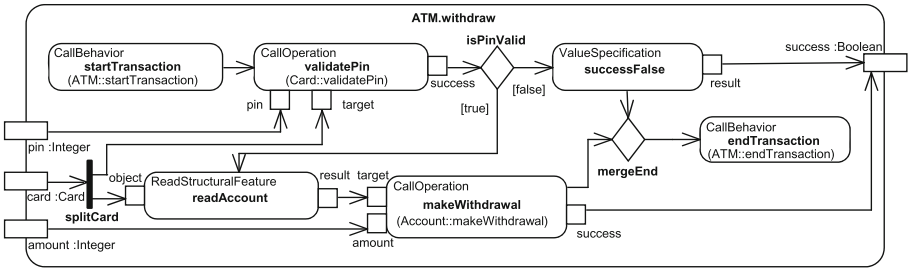

Fig. 2. Activity diagram of the operation *ATM.withdraw*

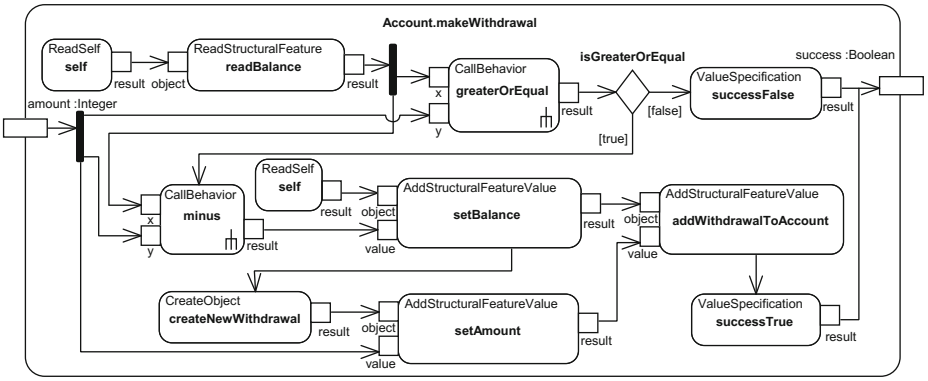

Fig. 3. Activity diagram of the operation *Account.makeWithdrawal*

This in turn causes the balance of the account to be updated and a corresponding withdrawal record to be created. Once the withdrawal has been completed, the transaction is ended and recorded (action *endTransaction*).

Please note that the actions *startTransaction*, *validatePin*, *makeWithdrawal*, and *endTransaction* are call actions calling the declared operations. The explained functionality of these operations are implemented by dedicated activities. In the following, we discuss the implementation of the operation *makeWithdrawal*. The remaining activities are omitted due to space limitations.

Figure 3 shows the activity implementing the operation *Account.makeWithdrawal*. This activity first retrieves the account's balance (action *readBalance*)

and compares it with the amount of money to be withdrawn (action *greaterOrEqual*). If the balance exceeds the amount of money to be withdrawn, the new balance is calculated (actions *minus*) and set (actions *setBalance*). Finally, a new withdrawal record is created (action *createNewWithdrawal*), its amount is set to the withdrawn amount of money (action *setAmount*), and it is associated to the bank account (action *addWithdrawalToAccount*). In the case that the withdrawal was performed, the value *true* is provided as output of the activity (action *successTrue*), otherwise *false* is provided (action *successFalse*).

2.1 Functional Requirements of the ATM System

In the following, we consider the functional correctness of the ATM's withdrawal functionality. For correctly handling withdrawals in case a *correct pin* was provided, the ATM system has to fulfill the following functional requirements.

FR1 The pin has to be validated before the actual withdrawal is performed.
FR2 The account's balance has to be reduced by the provided amount of money.
FR3 After the completion of the withdrawal, the balance of the account should be equal to the difference between the sum of all recorded deposits and the sum of all recorded withdrawals.
FR4 A new withdrawal record has to be created for the account.
FR5 The activity should return *true* indicating a successful withdrawal.
FR6 When the withdrawal is started, a new transaction should be created; once it is completed, the transaction should be ended and recorded.

2.2 Requirements of the Testing Framework

To verify that the fUML model of the ATM system fulfills the specified functional requirements, a testing framework is needed providing the following capabilities.

1. The testing framework shall provide the possibility to test the *chronological order* in which nodes of an activity are executed. Thereby, the framework ensures that the specified order is correct for every possible execution of the activity, taking concurrency into account. (Required for *FR1*)
2. The testing framework shall provide support for testing whether an activity produces the correct *output* for a given input. Also, checking the output of actions within the activity shall be supported. (Required for *FR2* and *FR5*)
3. The testing framework shall provide the possibility to test the *runtime state* of a system during the execution of an activity. Therefore, it has to enable the selection of the relevant runtime states, as well as the evaluation of expressions on these runtime states. (Required for *FR3*, *FR4*, and *FR6*: For *FR6* it has to be tested whether after the execution of the action *startTransaction* of the activity *ATM.withdraw* a new transaction has been created; for the final runtime state it has to checked whether the account's balance and records are consistent (*FR3*), a new withdrawal record has been created (*FR4*), and the started transaction has been recorded (*FR6*))

```
scenario atmTestData [
    object atmTD: ATM {}
    object cardTD: Card {pin = 1985;}
    object accountTD: Account {balance = 100;}
    object depositTD: Deposit {amount = 100;}
    link card_account {source card = cardTD; target account = accountTD;}
    link account_record {source account = accountTD; target records = depositTD;}
]
```

Listing 1. Test scenario for testing the ATM system

```
test atmTestSuccess activity ATM.withdraw(card=cardTD, pin=1985, amount=100) on
    ↪atmTD {
    assertOrder *, validatePin, *, makeWithdrawal, *; // FR1
    finally {
        readAccount.result::balance = 0; // FR2
        check 'BalanceRecords' on readAccount.result; // FR3
        check 'NumOfWithdrawalsSuccess' on readAccount.result; // FR4
        success = true; // FR5
    }
    assertState eventually after constraint 'TransactionCreated' { // FR6
        check 'TransactionEnded', 'TransactionAdded';
    }
}
```

Listing 2. Test case for testing the ATM system

```
context ATM
 exp TransactionCreated: currentTransaction <> null
 exp TransactionEnded: currentTransaction = null
 exp TransactionAdded: completedTransactions -> size() = 1
context Account
 exp NumOfWithdrawalsSuccess: records -> select(oclIsTypeOf(Withdrawal)) -> size()=1
 exp BalanceRecords:
  (records -> select(oclIsTypeOf(Deposit)) -> collect(amount) -> sum()) -
  (records -> select(oclIsTypeOf(Withdrawal)) -> collect(amount) -> sum()) = balance
endpackage
```

Listing 3. OCL constraints for testing the ATM system

The early prototype of our testing framework presented in [12] only partially supported these capabilities. In this work, we introduce new testing features that significantly extend the framework's capabilities and enable a more precise and thorough verification of the functional correctness of fUML activities.

3 Overview of the Testing Framework

Our testing framework is composed of a *test specification language* enabling the definition of assertions on the behavior of fUML activities and a *test interpreter* evaluating these assertions. In the following, we briefly introduce these two components and discuss their limitations as presented in [12].

3.1 Test Specification Language

The test specification language enables modelers to define *test suites* composed of test scenarios and test cases.

Test scenarios allow the specification of objects and links, which can be used both as input values and expected output values of activities under test. The

definition of a test scenario is composed of the keyword `scenario`, a scenario name, arbitrary many object definitions, and arbitrary many link definitions.

Listing 1 shows the test scenario defined for testing the ATM's functional requirements presented in Sect. 2. The test scenario is called *atmTestData* and defines four objects (keyword `object`), namely one *ATM* object, one *Card* object with the pin *1985*, one *Account* object with the balance *100*, and one *Deposit* object with the deposit amount *100*. Furthermore, it defines two links (keyword `link`), namely between the specified *Card* and *Account* objects, as well as between the *Account* and *Deposit* objects.

A *test case* tests the behavior of an activity. Its definition consists of the keyword `test`, a test name, the keyword `activity`, the name of the activity under test, an optional list of input parameter value assignments for the activity, an optional declaration of a context object for the activity, and a body. In the body an arbitrary number of order assertions and state assertions can be declared.

Listing 2 shows the test case *atmTestSuccess* asserting the functional requirements of the ATM's withdraw functionality defined by the activity *ATM. withdraw*. For the input parameters, the *Card* object *cardTD* defined in the test scenario (cf. Listing 1), the correct pin *1985*, and the amount to be withdrawn of *100* are provided. The activity is executed for the *ATM* object *atmTD* also defined in the test scenario. The test case consists of one order assertion (line 2) and two state assertions (lines 3-8 and 9-11), which are explained in the following.

Order assertion can be used to test the order in which the nodes of the activity under test are executed. To specify an order assertion, the keyword `assertOrder` is used followed by the list of nodes in their expected execution order. It is also possible to specify a relative order of nodes by the use of jokers for skipping exactly one ('_') or zero to many ('*') nodes.

The order assertion of the test case *atmTestSuccess* (cf. Listing 2, line 2) tests whether the action *validatePin* is executed before the action *makeWithdrawal* with arbitrary many nodes being executed before, in between, or after them.

State assertions can be used to check the runtime state of the tested system during the execution of the activity under test. The definition of a state assertion consists of the keyword `assertState`, a temporal expression selecting the runtime state to be checked, and arbitrary many state expressions defining the expected properties of the selected runtime state. The temporal expressions provided by the test specification language have been substantially extended and improved compared to our early prototype presented in [12]. They will be extensively discussed in Sect. 4.2. In line 4 of the test case *atmTestSuccess* (cf. Listing 2) we see an example of a state expression. It checks in the final runtime state of the ATM system whether the account's balance has been updated to 0. Please note that the test input for the activity *ATM.withdraw* defines that a withdrawal of the amount 100 should be performed for the account associated with the card *cardTD*. The card *cardTD* and its associated account *accountTD* have been defined in the test scenario shown in Listing 1. Because the initial balance of the account is specified to be 100 (cf. Listing 1, line 4), it is asserted whether after the withdrawal of 100, the account's balance is equal to 0.

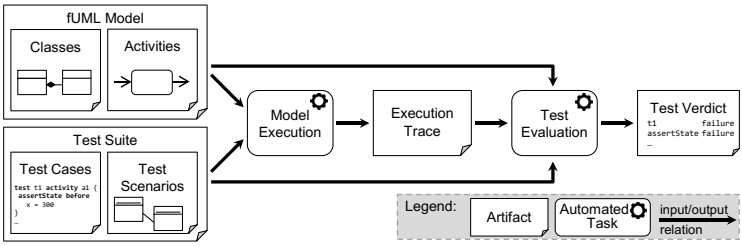

Fig. 4. Test interpreter

The test case shown in Listing 2 tests the fulfillment of all functional requirements defined in Sect. 2. The correspondences are provided in the comments in Listing 2. Please note that the test case uses the objects defined in the test scenario shown in Listing 1 as input values for the tested activity *ATM.withdraw*, and the OCL expressions shown in Listing 3 for several state assertions. We will discuss the test case in more detail in Sect. 4. A thorough discussion of the test specification language is also provided on our project website [13].

3.2 Test Interpreter

The test interpreter is responsible for evaluating test cases specified in the presented test specification language. The process of evaluating test cases is shown in Fig. 4. The input provided to the test interpreter consists of the fUML model to be tested and the test suite to be evaluated on this model. Each test case in the test suite is evaluated by executing the activity under test for the input values defined by the test case using the extended fUML virtual machine elaborated in previous work [9]. This extended fUML virtual machine captures execution traces reflecting the runtime behavior of the executed activity. In particular, an execution trace provides detailed information about the execution order of activities and activity nodes, inputs and outputs of activities and activity nodes, as well as the runtime state of the system at any point in time of the execution. The execution traces are analyzed by the test interpreter for evaluating every assertion defined by a test case. The output of the evaluation is a test report providing the information which assertions succeeded, which assertions failed, and further information on failing assertions, such as invalid execution orders of activity nodes and invalid system states.

3.3 Limitations

While the early prototype of our testing framework as presented in [12] supports assertions of a system's runtime state and the correct execution order of activity nodes, it has the following major limitations:

1. State assertions are restricted to simple equality checks of objects and their feature values. Complex expressions, such as iterations over a set of objects

or calculations over their feature values are not supported. Furthermore, the selection of the runtime states to be checked by state assertions can be only defined by referring to the execution of particular activity nodes, but not by defining conditions that should be fulfilled in the states to be selected.
2. Temporal expressions for selecting the states to be checked in a state assertion are limited to the temporal operators *after* and *before*, as well as the quantifiers *always* and *exactly*. They are insufficient for expressing more complex state assertions, such as that some property of the state eventually becomes *true* or that a certain property is valid in only some states.
3. Furthermore, order assertions are evaluated on a single execution path of the activity under test, which is insufficient in the presence of concurrency.

4 Extensions of the Testing Framework

To overcome the aforementioned limitations of the early prototype of our testing framework, we have extended it with support for OCL, additional temporal operators and quantifiers, as well as a new test evaluation algorithm accounting for concurrent behavior. These extensions are subject of this section.

4.1 OCL Expressions

With state expressions it is tested whether the runtime state of a tested system fulfills certain properties. In the early prototype of our testing framework, state expressions were restricted to simple equality checks. With this restriction, complex properties, such as needed for verifying the consistency of an account's balance with its deposit and withdrawal records (*FR3*), are not possible.

Supporting the definition of complex properties in state expressions requires the extension of our testing framework with a suitable expression language. Thereby, concepts allowing iterations over objects existing in a system's runtime states, calculations over these objects' feature values, and comparisons of values are of particular interest. This includes especially operations for the predefined types of fUML, such as *Collection* operations (e.g., *select()*, *forAll()*).

The integration of these concepts requires an extension of our test specification language with complex grammar concepts, as well as an extension of our test interpreter for evaluating expressions defined with these concepts. Both extensions are expensive to achieve without using an already existing expression language with supporting infrastructure. Thus, we decided against building our own expression language and interpreter, but instead integrated OCL with our testing framework. OCL [15] is a formal language providing concepts for defining expressions on UML models. Like UML, it is standardized by OMG and most of the experts in the modeling domain are familiar with OCL.

We integrated OCL with our testing framework, such that OCL expressions can be used for defining *complex conditions* on a system's runtime state as state expressions in state assertions, as well as for specifying *temporal expressions* selecting the runtime states to be asserted. This integration was achieved using

the DresdenOCL framework [7], which provides extension mechanisms that allow the integration of OCL into the abstract and concrete syntax of an existing modeling language, such as our test specification language, as well as the evaluation of OCL expressions on any model instances, such as the runtime state of a model represented with fUML as required by our test interpreter. Details about how such an integration of OCL may be achieved can be found in [7].

The OCL expressions used in the test cases for the ATM system are given in Listing 3. For instance, the OCL expression *BalanceRecords* (lines 7-9) specifies that the balance of an account should be equal to the difference between the sum of all recorded deposits and the sum of all recorded withdrawals. This OCL expression is used in the test case (cf. Listing 2, line 5) to test that the account's balance and its withdrawal and deposit records are consistent.

4.2 Temporal Expressions

Temporal expressions are used in state assertions for selecting the runtime states of a tested system that have to be checked for expected properties. Thereby, runtime states are generated during the execution of activities and capture the system's state after a certain action has been executed. For instance, as illustrated in Fig. 5, the states *S1*, *S2*, *S3*, and *S4* resulted from the execution of the actions *actionA*, *actionB*, *actionC*, and *actionD*, respectively. The values a, b, and c in each state represent the results of evaluating conditions on these states.

Temporal expressions are composed of temporal operators, temporal quantifiers, and actions or alternatively OCL constraints. In the early prototype of our testing framework, OCL constraints were not supported for selecting runtime states, and important temporal quantifiers, such as *eventually*, were missing. As part of our extensions, we also refined the supported temporal operators. In the following, we discuss the temporal operators and quantifiers based on Fig. 5.

Temporal operators in combination with the specification of actions are used for selecting the runtime states to be considered in a state assertion. We support the temporal operators *after* and *until* defining that all runtime states after or until an action has been executed shall be considered. If OCL constraints are used instead of actions, they are evaluated in each runtime state starting with the first one. Those states in which the constraints are evaluated to *true* for the first time are select, as well as all runtime states between them.

Temporal quantifiers are used for specifying in which of the selected runtime states the state expressions of a state assertion should evaluate to *true*. Our test specification language provides the temporal quantifiers *always*, *eventually*, *immediately*, and *sometimes* described in the following.

The temporal quantifier *always* defines that the state expressions should evaluate to *true* in all selected runtime state. For instance, the temporal expression *(1a)* specifies that in each state starting from the first one until the state produced by *actionB*, the value of the state expression c should evaluate to *false*.

The temporal quantifier *eventually* defines that each state expression should evaluate to *true* in one of the selected runtime states and should remain *true* in all of the following selected runtime states. For instance, the temporal expression

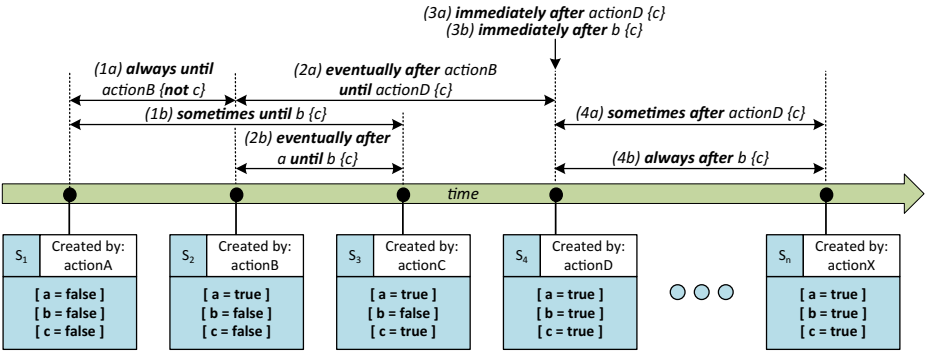

Fig. 5. Combinations of temporal operators and temporal quantifiers in state assertions

(2b) specifies that from the first state in which *a* becomes *true*, until the first state in which *b* becomes *true*, the value of the state expression *c* should become *true* in one state and remain *true* in each of the following selected states.

The temporal quantifier *immediately* specifies that each state expression should be *true* in either a runtime state created by the specified action or the one right before this state, depending on whether the temporal operator *after* or *until* is used. If an OCL constraint is used instead, the state expression should be fulfilled in the first state where the specified constraint evaluates to *true* or the state right before it. For instance, the temporal expression *(3a)* specifies that the value of the state expression *c* should be *true* in the state caused by *actionD*.

The temporal quantifier *sometimes* defines that each state expression should evaluate to *true* in at least one of the selected runtime states. The temporal expression *(1b)* specifies that the state expression *c* should evaluate to *true* in at least one of the states from the first state until the state where *b* becomes *true*.

We also introduced the keyword *finally* as a shorthand for `always after actionX` where *actionX* is the last executed action of the activity under test.

Looking back at the test case for the ATM system in Listing 2, the state assertion in lines 9-11 specifies that after the constraint *TransactionCreated* (cf. Listing 3) is evaluated to *true*, the constraints *TransactionEnded* and *TransactionAdded* should eventually evaluate to *true*. Thus, it is tested whether during the execution of the activity *ATM.withdraw*, a transaction is created, which is afterwards ended and recorded.

With the newly introduced and improved temporal operators and temporal quantifiers, and the capability to use OCL conditions in temporal expressions, a system's runtime states can be much more precisely selected for testing purposes than has been possible with the early prototype of our testing framework.

4.3 Concurrency

Concurrency in an activity leads to the existence of a potentially large or even infinite number of possible execution paths of that activity, which have to be

	st	sc	vp	ipv	ra	mw	me	et
startTransaction (st)			T					
splitCard (sc)			T		T			
validatePin (vp)				T				
isPinValid (ipv)					T			
readAccount (ra)						T		
makeWithdrawal (mw)							T	
mergeEnd (me)								T
endTransaction (et)								

Fig. 6. Adjacency matrix for evaluating order assertions on the activity *ATM.withdraw*

considered in the test evaluation. In particular, order assertions checking the correct execution order of activity nodes have to be evaluated for every possible execution path of the activity under test.

The early prototype of our testing framework did not account for concurrent execution paths of activities and thus reported false positive evaluation results for order assertions. This is because the fUML virtual machine executes concurrent paths sequentially, and thus the trace used for evaluating order assertions reflects only one possible execution order of activity nodes lying on concurrent paths. The early prototype checked order assertions only on this single sequential execution order. To overcome this limitation, we implemented a new evaluation algorithm for order assertions, which correctly deals with concurrent paths.

As a first step, the algorithm transforms the execution trace of the activity under test into an adjacency matrix. The execution trace from which we construct the matrix is like in the former version of the evaluation algorithm obtained from a single execution of the activity under test for the given input defined by the test case. However, the new algorithm also takes the input/output dependencies between executed activity nodes into account, which are also captured by the execution trace. Thereby, an activity node B depends on an activity node A, if B received an object token or control token from A as input. In this case, B is added as being adjacent to A in the adjacency matrix.

Figure 6 shows the adjacency matrix constructed for the execution of the activity *ATM.withdraw* (cf. Fig. 2) with the input values defined in our test case *atmTestSuccess* (cf. Listing 2). For instance, the activity node *validatePin* is adjacent to the activity node *startTransaction*, because it received a control token from *validatePin* via the defined control flow edge. Thus, the matrix contains a *true* value (abbreviated with *T*) in the first row and third column.

Based on the constructed adjacency matrix, order assertions can be evaluated efficiently by analyzing the dependencies between activity nodes specified in the order assertions. For instance, to evaluate an order assertion `assertOrder *, A, B, *`, we have to check whether B depends on A, i.e., whether a *true* value in the adjacency matrix indicates B as being adjacent to A. If this is not the case, there exists no input/output dependency between A and B and, hence, they may

be executed in reverse order. Furthermore, we have to check that there are no other nodes independent of both A and B, i.e., nodes that lie on parallel paths.

For the evaluation of the jokers '_' and '*', also indirect input/output dependencies between activity nodes have to be considered, which can also be efficiently calculated from the adjacency matrix. For instance, to evaluate an order assertion `assertOrder A, _, B`, we have to check whether an arbitrary activity node X exists on which B depends and which itself depends on A, i.e., X provided input to B and received input from A.

Looking back at our test case defined for the ATM system (cf. Listing 2), the order assertion defined in line 2 is evaluated by checking in the adjacency matrix (cf. Fig. 6) whether *makeWithdrawal* is directly or indirectly adjacent to *validatePin*. Because this is the case, indicated by the underlined *true* values in the matrix, the order assertion evaluates to *true*.

4.4 Implementation

We provide an open source implementation of our testing framework integrated with EMF. The testing framework is part of the larger project *moliz* [13], which is concerned with executing, testing, and debugging models based on the fUML standard. For implementing the grammar and editor of our test specification language, we have used the Xtext framework. The test interpreter is implemented in Java and based on an extended version of the reference implementation of the fUML virtual machine elaborated in previous work [9]. For the integration of OCL with our testing framework, we have used the DresdenOCL framework [7].

5 Evaluation

We evaluated our testing framework with the presented new functionality concerning *ease of use* and *usefulness* by carrying out a user study. In the following, we present the user study setup, as well as the results and lessons learned.

5.1 User Study Setup

The user study consisted of the following four steps, which were carried out with each participant individually.

(i) Introduction. At the beginning of the user study, the participant was given an introduction to fUML and our testing framework. This included the most important concepts of fUML comprising fUML's class concepts, activity concepts, and action language. Furthermore, a simple exemplary fUML model was introduced and used to explain the main concepts of our test specification language.

(ii) Skills questionnaire. The target group of users of our testing framework are practitioners in the MDE domain using UML activity diagrams to define the behavior of systems. Thus, in order to obtain relevant results, our selection of participants was based on their background in UML, OCL, and unit testing.

The participants' skills in these languages were collected using a questionnaire. Most of the participants had a good background in UML being slightly more experienced with class diagrams than with activity diagrams. The knowledge of the UML action language was balanced from having no experience to being an expert. Most of the participants declared their experience with OCL at the beginner level, while unit testing knowledge was declared as average by most participants. The participants of the user study consisted of post-doctoral researchers, PhD students, and master students of the Vienna University of Technology.

(iii) Testing tasks. The participants were asked to complete two tasks with our testing framework. Therefore they used our implementation of the testing framework, including an editor for writing test cases and the test interpreter running test cases and providing the results as console output.

The aim of the *first task* was to evaluate the *ease of use* of our testing framework. Therefore, the participants had to define a test suite implementing predefined functional requirements for two given and correct activities. For the first activity, the participants had to specify a test scenario with one object, and two test cases with two different order assertions and two different state assertions. The activity comprised nine nodes and included simple fUML action types, such as *value specification action*. For the second activity, the participants had to specify a test scenario with several objects and links, two state assertions, and one OCL expression. The activity was composed of fourteen activity nodes and included slightly more complex fUML concepts, such as *expansion regions*.

With the *second task*, we aimed at evaluating the *usefulness* of our testing framework, in particular, the usefulness of test results for detecting and correcting defects in UML activity diagrams. In this task, the participant was given a defective activity diagram, two test cases testing the activity diagram, and the test results. Based on the test cases and test results, the participant had to locate the defects and suggest corrections. The activity consisted of nine activity nodes and included simple fUML action types. Two defects were introduced into the UML activity diagram. One defect consisted of wrong guards for a decision node, which led to the execution of a wrong path. This defect was detectable from the test result of a failing order assertion. The second defect consisted of a missing merge node, which led to an activity node not being executed. This defect was detectable from the test result of a failing state assertion.

(iv) Opinion questionnaire. Finally, the participants rated the ease of use and usefulness of the testing framework in a questionnaire.

More details about the case study setup including the used fUML models, test suits, and task descriptions may be found at our project website [13].

5.2 Results and Lessons Learned

We observed the participants during performing the given tasks to find out *(i)* how easy our testing framework is to use for testing UML activity diagrams (first task), and *(ii)* whether test results are useful for detecting and correcting defects in UML activity diagrams (second task).

(i) Ease of use. For the first task, where the participants had to define test cases, we made the following observations.

Test scenarios. Most of the participants had at the beginning problems to understand the purpose of test scenarios, because they tried to define the test scenarios before thinking about and writing the actual test cases. However, after having defined the first test case, the participants understood how to use test scenarios for providing input to the activities under test.

Order assertions. Another frequently observed problem encountered by the participants was to correctly specify order assertions. Several participants specified the expected order of activity nodes incorrectly, as they forgot to use jokers for allowing arbitrary nodes to be executed between two nodes of interest. However, after running the order assertion and reading the failing test result, all participants were able to correct the order assertion.

State assertions. A third recurring issue was related to understanding the relation between temporal expressions and state expressions. More precisely, several participants specified each state expression separately in a distinct state assertion, even though the temporal expressions of these state assertions were identical (i.e., only one state assertion would have been required).

OCL expressions. Several participants had issues with specifying the OCL expression required for one of the test cases. However, this was due to the fact that these participants had little experience with OCL. Connecting the OCL expression with a test case was not an issue for any of the participants.

Overall, we observed that after each written test case, the participants were making fewer mistakes in specifying the next one. By the time they got to the second task, all participants had a clear understanding about all the concepts provided by the test specification language. From this observation, we conclude that our test specification language has a gentle learning curve. One of the possible improvements that we discovered during the user study is that some concepts of the test specification language, such as the specification of links in test scenarios, could be improved. Furthermore, additional validations by the editor would significantly improve the specification of test cases, as it prevents defects in the test cases themselves.

(ii) Usefulness. In the second task, the participants had to detect and correct defects in a UML activity diagram based on test cases and test results. We made the following observations for this task.

Understanding test cases. The participants had no problems in understanding the given test cases and their purpose. They were able to correctly explain the functional requirements tested by the test cases.

Understanding test results. Out of the eleven participants, five were able to locate both introduced defects, three were able to locate the first defect only, and three were not able to locate any of the defects.

For identifying the first defect, we provided the participants with a test case testing the expected execution order of activity nodes with an order assertion, as well as the test result of running the test case on the defective activity. The test result listed the actually executed path, which allowed all of the participants to

detect that a wrong path was executed. Eight of the participants were also able to identify the related defect, namely wrongly defined guard conditions. Three participants were not able to locate this defect, because they were not familiar with how guard conditions are evaluated in UML based on object flows.

The second defect was a missing merge node, which impeded the execution of an activity node and consequently resulted in a wrong final runtime state of the tested system. For identifying this defect, we provided a test case checking the final runtime state with a state assertion, as well as the test result. The test result showed both the actually last executed activity node and the actual final runtime state. Neither of them was as expected by the state assertion. For identifying the causing defect, the participants had to detect that the activity node leading to the expected final runtime state was not executed and that the reason for this was the missing merge node. This was not as obvious as in the former example, where the result of an order assertion clearly showed which nodes of the tested activity were executed and which were not. Furthermore, identifying that a merge node has to be introduced to correct the defect requires the knowledge that alternative branches in UML activities always have to be explicitly merged by a merge node. Thus, the participants who did not have this knowledge were not able to identify the missing merge node as defect.

With this task, we aimed at evaluating how useful test results are for detecting and resolving defects in UML activity diagrams. We define the property *usefulness* as the average percentage of defects resolved by participants based on test results. Let D be the number of defects introduced into an activity, and RD_i the number of defects resolved by participant i. Then, the percentage of resolved defects by user i is $X_i = RD_i/D * 100$. The metric for measuring usefulness is $U = \sum X_i/n$, where n is the number of participants. According to this metric, the usefulness measured through the user study is $U = (5 * 100\% + 3 * 50\% + 3 * 0\%)/11 = 59.09\%$. This measure indicates a positive result for the usefulness of test results for detecting defects in activities. However, it also indicates that further improvements are needed.

Our conclusion drawn from these observations is that the visualization of test results is crucial for making them useful for locating defects. Therefore, providing more effective means for visualizing test results have to be investigated in future work. For instance, we intend to investigate the integration of the visualization of test results with UML modeling editors, such that test results can be presented on the tested activity diagrams themselves. Furthermore, presenting the states of a system occurred during the execution of an activity under test in the form of UML object diagrams could be useful, as it may provide more insight into the cause of failing test cases. Furthermore, for localizing a defect and deriving valid corrections, debugging is essential. Providing users with the possibility to step through the execution of an activity and observe the state of the system after each step may facilitate the localization of defects causing failing test cases.

Table 1 shows the results of the opinion questionnaire filled in by the participants to rate how difficult it was to accomplish the given tasks. As can be seen from the results, our observations and conclusions correspond to the participants' opinion.

Table 1. Results of the opinion questionnaire

Task	very easy	easy	medium	hard	very hard
Read class diagrams	7	4			
Read activity diagrams	3	7		1	
Write test cases		8		3	
Read test cases	3	4	2	2	
Read test results	3	4	2	2	
Correct activity diagrams	1	3	2	2	3

Threats to Validity. There are several threats to the validity of the evaluation results. First, in order to make the evaluation feasible in the described setup, the examples given to participants were of low complexity. Having more complex examples might give better insights into the ease of use of the test specification language and the usefulness of test results for detecting defects. Another threat to validity is the selection of participants. The participants consisted of researchers and students, but participants from industry were missing. Furthermore, also the fairly low number of participants influences the validity of the results. As future work, we intend to perform a larger user study with more participants having different background and knowledge, as well as with more complex examples.

6 Related Work

Until now, testing UML activity diagrams conforming to the fUML standard has not been investigated intensively. We are only aware of the work by Craciun *et al.* [4], who propose to develop a virtual machine for fUML models using the K-framework for efficiently testing fUML models. However, this work is still in its early stage and there is yet no information about an existing implementation.

For UML 2 activities and actions, Crane and Dingel [5] present an interpreter, which offers several dynamic analysis capabilities, such as reachability and dead lock analysis, as well as assertions on objects during the execution of activities. The latter capability is similar to the state assertions provided by our testing framework. However, only some simple expressions on objects are supported. In contrast, our testing framework supports the full power of OCL.

Another interesting line of work related to state assertions is temporal OCL. It is an extension of OCL with temporal operators and quantifiers (e.g., [3]) enabling not only the evaluation of OCL expressions on a single state of a system but also on its evolution. Thus, temporal OCL could be used in a similar way as our state assertions for testing purposes. However, our testing framework does not extend OCL with temporal expressions, but rather uses it as is and instead provides temporal expressions as part of the test specification language.

Contrary to testing techniques, several approaches applying formal analysis techniques on fUML activities have been proposed. Romero *et al.* [18] show how the standardized formal semantics of fUML can be utilized to perform formal

verification through theorem proving. Abdelhalim *et al.* [1] developed a framework that automatically formalizes fUML models as CSP processes and analyzes them for deadlocks. Laurent *et al.* [8] define a first-order logic formalization for a subset of fUML and apply model checking techniques for verifying the correctness of process models defined with fUML. Their formalization covers control and data flows, as well as resource and timing constraints. Properties that are verified include termination and dead lock freeness. Planas *et al.* [17] propose a verification method for fUML models, which focuses on the property *strong executability*. This property guarantees that every time an activity is executed, the system's state is changed in a way consistent with all defined integrity constraints. Micskei *et al.* [11] propose a transformation chain from UML models to formal verification tools using fUML and Alf as intermediary languages. In particular, they propose to translate UML state machines into the formal language of the UPPAAL tool environment, which provides a model checker allowing the formal verification of the modeled behavior.

Further approaches dealing with the formal analysis of UML activity diagrams exist, which, however, do not consider the full power of fUML. For instance, Eshuis and Wieringa [6] present a formalization of workflow models specified as UML activity diagrams for verifying functional requirements. In their approach, activity diagrams are translated into transition systems, functional requirements are defined as LTL formulas, and these LTL formulas are evaluated on the obtained transition systems using the NuSMV model checker.

7 Conclusion and Future Work

In this paper, we have presented a testing framework for fUML models, which allows modelers to verify the correct behavior of fUML activities. Besides giving an overview of the testing framework, we have explained three newly introduced features in detail, which significantly extend the framework's testing capabilities. In particular, we introduced support for OCL allowing the evaluation of more complex conditions on the expected runtime state of a system under test. Furthermore, our testing framework now provides additional temporal operators and quantifiers for more precisely selecting the runtime states to be asserted by test cases. Finally, we developed a new algorithm for verifying the correct execution order of activity nodes in the presence of concurrent behavior.

Based on the lessons learned from evaluating our testing framework in a user study, we intend to improve the ease of use of our test specification language by adapting its textual syntax, as well as the usefulness of test results by investigating more effective visualization techniques. Furthermore, we plan to further improve the evaluation of assertions taking into account concurrency. In particular, concurrent paths also have to be considered when evaluating state assertions, as actions modifying and accessing the same values concurrently might lead to nondeterminism. Another interesting feature that we have identified for future work is the support of comparisons between distinct runtime states of the tested system, i.e., the comparison of runtime states at different points in time.

References

1. Abdelhalim, I., Schneider, S., Treharne, H.: An Integrated Framework for Checking the Behaviour of fUML Models using CSP. International Journal on Software Tools for Technology Transfer **15**(4), 375–396 (2013)
2. Bézivin, J.: On the unification power of models. Software and Systems Modeling **4**(2), 171–188 (2005)
3. Bill, R., Gabmeyer, S., Kaufmann, P., Seidl, M.: OCL meets CTL: Towards CTL-Extended OCL model checking. In: Proc. of 14th Int. Workshop on OCL, OCL 2013. CEUR WS, vol. 1092, pp. 13–22. CEUR-WS.org. (2013)
4. Craciun, F., Motogna, S., Lazar, I.: Towards better testing of fUML models. In: Proc. of 6th Int. Conf. on Software Testing, Verification and Validation, ICST 2013, pp. 485–486. IEEE Computer Society (2013)
5. Crane, M.L., Dingel, J.: Towards a UML virtual machine: implementing an interpreter for UML 2 actions and activities. In: Proc. of 2008 Conf. of the Center for Advanced Studies on Collaborative Research, CASCON 2008, pp. 8:96–8:110. ACM (2008)
6. Eshuis, R., Wieringa, R.: Tool Support for Verifying UML Activity Diagrams. IEEE Transactions on Software Engineering **30**(7), 437–447 (2004)
7. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Thiele, M., Wende, C., Wilke, C.: Integrating OCL and textual modelling languages. In: Dingel, J., Solberg, A. (eds.) MODELS 2010. LNCS, vol. 6627, pp. 349–363. Springer, Heidelberg (2011)
8. Laurent, Y., Bendraou, R., Baarir, S., Gervais, M.-P.: Formalization of fUML: an application to process verification. In: Jarke, M., Mylopoulos, J., Quix, C., Rolland, C., Manolopoulos, Y., Mouratidis, H., Horkoff, J. (eds.) CAiSE 2014. LNCS, vol. 8484, pp. 347–363. Springer, Heidelberg (2014)
9. Mayerhofer, T., Langer, P., Kappel, G.: A runtime model for fUML. In: Proc. of 7th Workshop on Models@run.time, MRT 2012, pp. 53–58. ACM (2012)
10. Mellor, S.J., Balcer, M.: Executable UML: A Foundation for Model-Driven Architectures. Addison-Wesley Longman Publishing Co., Inc. (2002)
11. Micskei, Z., Konnerth, R., Horváth, B., Semeráth, O., Vörös, A., Varró, D.: On open source tools for behavioral modeling and analysis with fUML and Alf. In: Proc. of 1st Workshop on Open Source Software for Model Driven Engineering, OSS4MDE 2014. CEUR WS, vol. 1290, pp. 31–41. CEUR-WS.org. (2014)
12. Mijatov, S., Langer, P., Mayerhofer, T., Kappel, G.: A framework for testing UML activities based on fUML. In: Proc. of 10th Int. Workshop on Model Driven Engineering, Verification and Validation, MoDeVVa 2013. CEUR WS, vol. 1069, pp. 1–10. CEUR-WS.org. (2013)
13. Moliz Project. http://www.modelexecution.org
14. OMG: OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1 (August 2011). http://www.omg.org/spec/UML/2.4.1
15. OMG: OMG Object Constraint Language (OCL), Version 2.3.1 (January 2012). http://www.omg.org/spec/OCL/2.3.1
16. OMG: Semantics of a Foundational Subset for Executable UML Models (fUML), Version 1.1 (August 2013). http://www.omg.org/spec/FUML/1.1
17. Planas, E., Cabot, J., Gómez, C.: Lightweight verification of executable models. In: Jeusfeld, M., Delcambre, L., Ling, T.-W. (eds.) ER 2011. LNCS, vol. 6998, pp. 467–475. Springer, Heidelberg (2011)
18. Romero, A., Schneider, K., Gonçalves Vieira Ferreira, M.: Using the base semantics given by fUML for verification. In: Proc. of 2nd Int. Conf. on Model-Driven Engineering and Software Development, MODELSWARD 2014, pp. 5–16. SCITEPRESS Digital Library (2014)