# Execution of UTP test cases using fUML

Marc-Florian Wendland
Fraunhofer FOKUS
Berlin, Germany
marc-florian.wendland@fokus.fraunhofer.de

Niels Hoppe
Fraunhofer FOKUS
Berlin, Germany
niels.hoppe@fokus.fraunhofer.de

## ABSTRACT

The UML Testing Profile (UTP) is a standardized modeling language that offers concepts relevant to specify test cases, test data and even entire test automation architectures including test environments. Just recently in June 2018, the OMG adopted the official version 2.0 of UTP. It was primarily designed to support both manual and automated activities of the dynamic test process, in particular the design and specification of test cases, test data and test suites. Basically, such UTP-based test specifications remain on a platform-independent level and leave it open how the test cases shall be executed in the end. In this paper, we describe an approach to executing UTP test cases via the executable UML standards fUML and PSCS. Therefore, we map platform-independent UTP test cases, expressed as Interactions, into executable fUML test cases for eventual execution against fUML systems. It is a first step towards an executable representation of UTP.

## KEYWORDS

UML Testing Profile, UTP, executable UML, executable UTP, automated test execution, fUML, fUML-based test automation, architecture

## 1 INTRODUCTION

In system engineering, executable specifications serve the purpose of validating the feasibility of system architecture as well as the consistency of systems/software requirements specifications (SRS) through simulation. A simulation of systems requirements specifications enables engineers to evaluate the efficiency of competing architectural solutions as well as the correctness and completeness of requirements. Furthermore, systems simulations help understanding how the systems behaves under varying (simulated) environmental conditions or in situations that are deemed critical but costly or even not possible to provoke in a real environment (e.g. failure of a flight or train control system while operating, validation of functional countermeasure in a nuclear power station etc.). With the upcome of the executable UML that currently consist of Semantics of a Foundational Subset of Executable UML (fUML), Precise Semantics of Composite Structures (PSCS) and Precise Semantics of State Machines (PSSM), it is possible to build executable specifications with UML.

From a tester's point of view, testing an executable SRS is the earliest point in time where dynamic testing is feasible. Dynamic testing mainly consists of designing and executing test cases against the system under test (SUT). This can be a simulation of the system or its eventual implementation, however, testing both kinds of realizations does not vary much. In both cases, test cases are (usually) derived from an SRS and executed against the SUT. It is expected that both, the simulated and implemented SUT, basically behave functionally equivalent. Thus, test cases designed for testing an executable specification of the system ought to be reused for testing the eventual implementation of the system in a dedicated test environment or in the field.

Ideally, a methodology for continuous dynamic testing fosters reuse of test cases across target technologies and test interfaces. A prerequisite for such a continuous test design approach is a test specification language that enables the specification of platform-independent test models, consisting of a test architecture and test cases, for later generation of executable test cases for the respective testing target environment. The UML Testing Profile (UTP) [10] is such a test modeling language.

As an extension to UML, it offers test-specific concepts on top of UML, used in particular to support dynamic testing. The intention of using UTP for testing is, similar to using UML for system analysis and design, to facilitate communication and easy comprehensibility of test specifications. It does not claim to be an executable modeling language, even though, it allows for specifying entire test automation architectures, including support for test design, test execution, verdict calculation and test log analysis.

In this paper, we concentrate on a mapping from a platform-independent UTP test model to a fUML test automation solution and executable test cases for testing fUML systems. Therefore, we assign an executable semantics to a subset of UTP concepts necessary for our methodology by mapping the respective concept to executable fUML concepts. We assume that the reader has basic knowledge about UML, UTP and fUML. The contributions of this paper are:

- Defining the structures for executable, yet platform-independent UTP test models; and
- Describing a mapping of structural and behavioral aspects of such UTP test models to fUML test models for eventual test case execution.

The remainder of this paper is outlined as follows: Section 2 summarizes existing work that relates to or influenced our work. Section 3 introduces the UTP concepts relevant for fUML-based test execution. The executable semantics for these concepts is provided by a mapping to their fUML counter parts. Section 4 describes mapping UTP test architectures to executable test environments, whereas section 5 explains the mapping from UTP test case behavior to fUML test cases. Section 6 briefly describes the tools and technologies we used to implement and run the approach. Section 7 eventually concludes this paper and outlines future work.

## 2 RELATED WORK

UTP has been subject to publication for over a decade now. Baker et al. wrote a comprehensive book that accompanied the first version of the standard [1]. UTP has been used for model-based generation of test cases for service-oriented architectures [11] as well as for testing product lines [4]. The work from Zander et al. [14] is closest to our work, for it specifies a mapping from UTP test cases to executable TTCN-3 test cases. Besides being based on the earlier UTP version 1.0 as opposed to our work, which is based on the current version 2.0, Zander's et al. work targets solely the mapping of test cases without addressing aspects like test sets, scheduling or arbitration. These parts of the test automation architecture were assumed to be realized by an existing TTCN-e execution engine. Thus, the work described by Zander et al. concentrated on the mapping of Interactions to TTCN-3 in the first place, whereas our work addressed also the generation of (parts of) the components of the fUML test automation architecture, such as scheduling and logging.

To the best of our knowledge, there is no previous work that targets a mapping from UTP test models to fUML models for the sake of black-box testing of fUML models. However, testing of fUML models was subject to previous work. Mijatovic et al. [5] proposed a framework for testing fUML models that is based on a proprietary domain-specific language (DSL) for describing assertions on the execution flow of fUML Activities. Apart from not being based on UTP, this approach is different to our work, because it is capable of white-box testing e.g., by asserting the execution order of the executed *ActivityNodes* or the input/output values of single *ActivityNodes* within an executed Activity. This is not possible and not intended by our approach that concentrates on a pure black-box approach to continuous test design.

Craciun et al. [2] addressed testing of fUML models using a rewrite-based executable semantic framework called K. The information obtained from that early work makes it hard to relate the work to our approach. Unfortunately, the work was not continued so that no further information about the approach can be stated here.

The executable UML standard PSCS [9] contains a conformance test suite written in fUML and the Action Language for Foundational UML (ALF) [7], that resembles the assertions-oriented unit test frameworks such as JUnit or NUnit[1]. Additionally, the executable UML standard PSSM [8] defined another conformance testing framework based on fUML and ALF, which compares string collections of expected events with a captured string collection of executed event sequences from a set of input events. Both approaches utilize fUML and ALF to specify test cases for fUML models. Our approach, however, utilizes fUML only for test execution. Test design, logging and verdict calculation happens on a higher level of abstraction, i.e., on UTP test model level, independent from any executable language to facilitate continuous test design.

Wendland [12] described an approach towards the definition of a precise semantics for UML Interactions by mapping them to fUML Activities. Even though that work was independent of UTP or testing in general, it influenced and inspired the mapping of UTP test cases to an fUML-based representation described in the current paper.

Summarizing the related work it can be stated, that different parts of our approach (e.g., fUML, PSCS, UTP, testing of fUML models) were subject to past research work. Yet the approach to provide an executable semantics for a subset of UTP by mapping the respective concepts to fUML and finally execute those executable-made UTP test models against a fUML-based system model for the sake of early validation of its requirements is new.

[1]See https://junit.org and http://nunit.org respectively.

## 3 UTP- AND FUML-BASED TEST MODELS

UTP is a standardized graphical modeling language that enhances UML with test-specific concepts for designing, visualizing, specifying, analyzing, constructing, and documenting artifacts commonly used in and required for expressing test specifications. As an open ended standard, UTP neither prescribes the test modeling methodology, application domain, testing tool, nor the eventual target technology that is used for carrying out testing activities. In June 2018, a new version of UTP has been adopted by OMG, i.e., UTP 2.0, called UTP 2. The work described in this paper uses UTP 2 test models as input for the mapping to an executable representation.

In our approach, two kinds of test models are distinguished, the UTP test model and fUML test model. Inspired by the principles of the Model-Driven Architecture (MDA) [3], the UTP test model is henceforth called *platform-independent test model* (PITM), the fUML test model *platform-specific test model* (PSTM). The *PITM* specifies test cases and test architectures on a higher level of abstraction and makes no assumptions on the eventual target environment or implementation of the SUT. It provides a set of logical test cases formalized as sequence diagrams, which are used to generate executable test cases for different target environments (e.g., JUnit, TTCN-3, fUML) or test levels (e.g., simulation testing, component testing, system testing). In contrast, the *PSTM* merely represents an executable version of the *PITM* test cases. Comparable to the bytecode of a JUnit test case, the *PSTM* is, in theory, of less interest to the test engineer, as the semantics of the test case is defined on *PITM* level, whereas the *PSTM* is completely derived and thereby rendered transient and transparent. Thus, there is usually no need to generate a user-friendly variant of the *PSTM* , e.g. by using ALF instead of fUML, as test engineers are not required to understand the specifics of the execution. This is another benefit of continuous test design.

Since UTP is not per se an executable language, it is necessary to define an executable semantics for a set of required concepts. This executable semantics can either be specified in the same way as fUML, PSCS or PSSM was composed, in which case, a dedicated runtime environment would be required to actually run the UTP test cases. As a more concise and convenient way, we chose to rely on the already existing executable semantics of fUML concepts (PSSM and PSCS base their executable semantics on fUML, too) and provide a mapping for every necessary UTP concept to an executable fUML concept. This mapping approach is also consistent with mapping *PITM* test cases to other executable test languages such as JUnit or TTCN-3.

Similar to fUML, which represents an executable subset of UML, not all UTP concepts are deemed necessary for execution. For the definition of the mapping rules from *PITM* to *PSTM* it is necessary to identify those concepts from UTP for which an executable semantics is required. Figure 1 illustrates a (simplified) *PITM* featuring a subset of UTP concepts that are necessary for test execution and will be used for describing the mapping to the *PSTM* . These required UTP concepts are explained in greater detail subsequently. The SUT shown in Figure 1 represents an fUML model of an elevator system. Internals of the elevator systems are not important for this paper, since it concentrates on technical aspects of the mapping from UTP test models to an fUML test automation architecture and test cases instead of actual verification of the elevator system.

The essential concept in our approach is the test case (*Component* with *«TestCase»* applied), which shall be contained by test sets (*Package* with *«TestCase»* applied). Test sets serve as containers for a set of test cases sharing a certain purpose (e.g., regression testing, smoke testing etc.). The behavior of a test case is given as an *Interaction*, usually visualized by a sequence diagram, but not required to for any UML behavior can be used to specify test cases. Consequently, only the elements *Lifeline*, *MessageOccurrenceSpecification* (MOS), *Message* and *GeneralOrdering* are of interest.

UTP introduces a dedicated abstract procedural language based on stereotypes that abstract from the underlying UML metaclasses in order to simplify the construction and analysis of test cases without the need to know about the underlying UML behavior. The central elements of that procedural language are so called *test actions*. In this paper, we will focus on test actions for sending stimuli to (*Message* with *«CreateStimulusAction»*), and expecting responses (*Message* with *«ExpectResponseAction»*) from the SUT. See the UTP specification [10], Clause 8.5.2 ProceduralElements and Clause 8.5.3 Test-specific actions for further details.

A test case must always be executed on a composite structure called *test configuration* (i.e., *Components* stereotyped with *«TestConfiguration»*), which defines communication channels between its parts, who are called *test configuration roles*. They represent either test components or test items, distinguished by the stereotypes *«TestItem»* and *«TestComponent»*. A test item represents the SUT, whereas the test components belong to the test environment and drive a test case's behavior by sending stimuli to or expecting responses from the SUT. Test configurations can be shared across multiple test cases. Test cases are then linked with a respective test configuration by establishing a *Generalization* dependency among themselves and the test configuration.

## 4 GENERATING THE EXECUTABLE TEST ENVIRONMENT

### 4.1 Mapping test sets

As opposed to *PITM* test sets, *PSTM* test sets are executable entities, for they schedule the execution and logging of the test cases they contain. Therefore, each *PITM* test set is translated into a fUML *Class*, whose classifier behavior, i.e., *test set scheduler*, is responsible for instantiating and eventually executing the respective test cases as illustrated in Figure 2. Test cases are represented as nested classes of the *PSTM* test set and executed by the test set scheduler, whose behavior consists of the two phases *initialization* and *execution of test cases*.

During *initialization* of the test set, an instance of a test set log is created. This test set log links all the test case logs of the executed test cases in the end. The created object is then passed as an argument to the invoked (i.e., executed) test case.

During *execution of test cases*, each test case that belongs to the *PITM* test set is scheduled for execution. This scheduling manifests in a generated flow consisting of a *CreateObjectAction* and a *StartObjectBehaviorAction* for each test case. Since *PITM* test cases are currently unordered in a *PITM* test set[2], there is no possibility to specify the eventual execution order of test cases in the *PSTM* , leaving it up to the transformation engine.

### 4.2 Setting up the executable test environment

*4.2.1 Mapping test components.* On *PITM* level, test components are represented as *Components* without any classifier behavior. The behavior of a test component is determined by every *Lifeline* that represents the test component within a test case. On *PSTM* level, however, a test component's behavior must be precisely and unambiguously defined, but since a test component must only exists during the execution of a test case, our mapping towards *PSTM* test components is fairly simple: For each test configuration role with *«TestComponent»* applied, a dedicated fUML *Class* is generated that represents the executable test component for a single test case. It is represented by an active *Class* with an *Activity* as its classifier behavior. This classifier behavior is derived from the covering *InteractionFragments* of the *Lifeline* representing the test component in the corresponding *PITM* test case and is described later to a greater extent.

*4.2.2 Mapping the test configuration.* On *PITM* level, reuse of test components and test configuration is encouraged by our methodology to ensure high maintainability of the test cases. On *PSTM* level, maintainability is not important for the executable artifacts are generated entirely from the *PITM* . Each test case is translated into a dedicated *Class* that directly contains the *PSTM* representation of the test configuration by processing the *Generalization* between the *PITM* test case and the *PITM* test configuration. Thus, the structure of the test configuration is replicated for each separate test case in the *PSTM* (Figure 5). The test item, however, is not generated, for it already exists as fUML *Class* that will merely be integrated into the test configuration.

The instantiation and activation of the generated *PSTM* test cases and their test configuration roles takes advantage of the *CS_DefaultConstructStrategy* from PSCS, making explicit creation, assembly and activation of objects obsolete. Anyhow, preparing the execution of a *PSTM* test case consists also of actions for preparing for logging, starting the test item and coordinating the test components.

Since the test components are active *Classes*, i.e., they possess their own thread of control and run in parallel once they are instantiated, so coordination of the test component behaviors is required. In particular, the test components must not send signals to the test item before the test item is itself instantiated and its behavior started. This coordination is achieved on *PSTM* level by generating another *Class*, the so called *(test case) coordinator*, which is integrated into a test case's test configuration and connected with the test components through dedicated synchronization *Ports* and an n-ary *Connector*, the so called *synchronization bus*, which is responsible for the transmission of synchronization messages to the test components. The coordinator also serves as an intermediary for exchanging data between the test case and the test components.

[2]There is a concept in UTP called *«TestExecutionSchedule»* that enables ordering of test case execution, however, this concept is not part of our work yet.

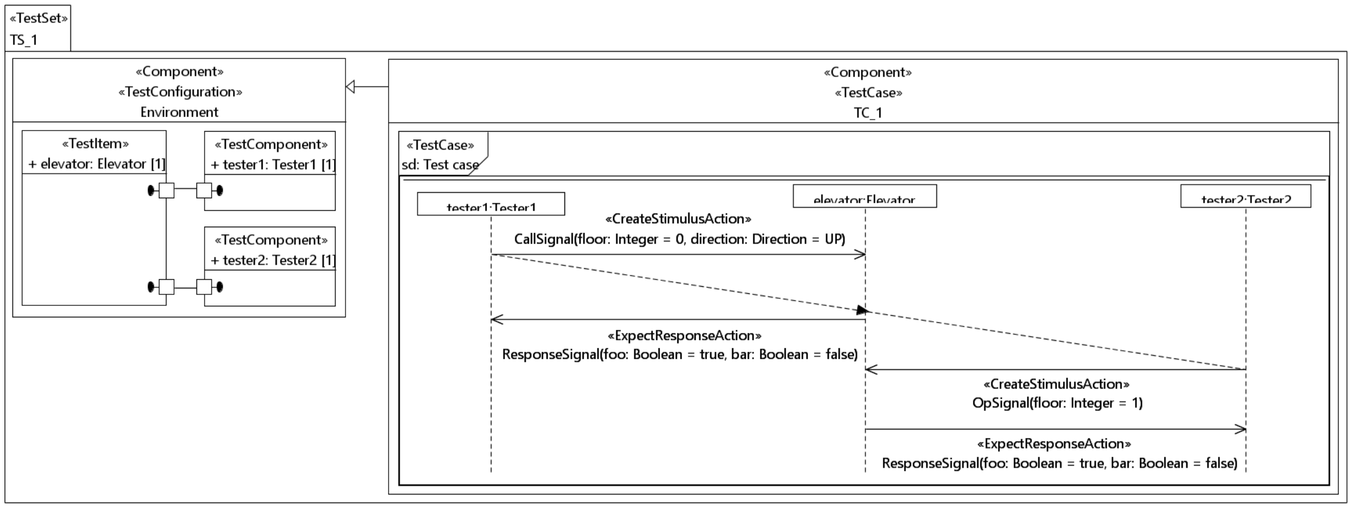

**Figure 1: *PITM* test model example**

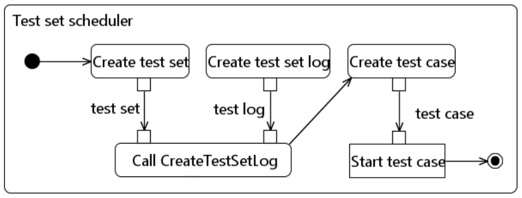

**Figure 2: Test set (scheduler) behavior**

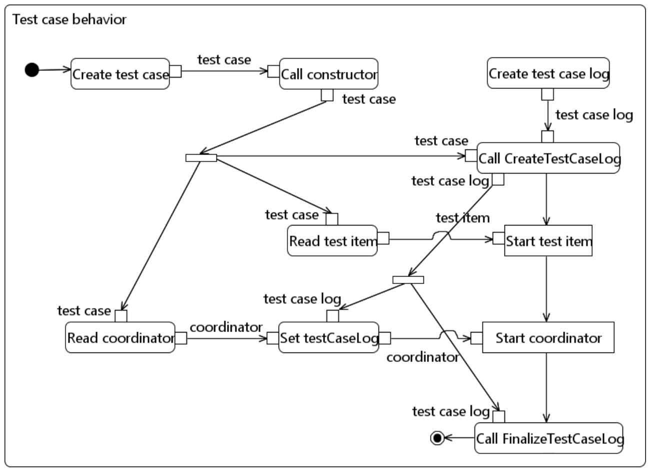

**Figure 3: Test case behavior**

*4.2.3 Preparing the test case execution.* Figure 3 illustrates the preparation of a test case execution prior to running the actual *PITM* test case behavior. In this regard, the classifier behavior of the *PSTM* test case creates a test case log by means of a *CreateObjectAction* 'Create test case log' and a *CallBehaviorAction* 'Call CreateTestCaseLog' and distributes it to the test components through the coordinator. The mechanism of creating test case logs is out of scope of this paper.

In order to prevent a potential loss of signals to or from the test item, it is important to ensure the complete setup of the test environment before the test item behavior commences. Therefore we require the test item to be modeled as a passive (not active) class, even though this is in violation with fUML.[3] Consequently, the test item must be started manually by a *ReadStructuralFeatureAction* 'Read test item', which passes the resulting object to a *StartObjectBehaviorAction* 'Start test item'.

*4.2.4 Coordinating the test case components.* Starting and synchronizing the test components is the responsibility of the *(test case) coordinator* (see Figure 4). For this purpose, it sends a start signal over the *synchronization bus.* The test component behaviors wait for this start signal before they commence their actual behavior. Added as an argument to the start signal is a reference to the test case log. This log object is required to capture the test actions executed by the test components.

Following this initial broadcast is a flow consisting of parallel *AcceptEventActions* for all *SignalEvents* corresponding to the completion signals of the involved test components. Only after all completion signals are received, the coordinator behavior completes and the test case can enter the finalization phase.

*4.2.5 Finalizing the test case execution.* After the completion of all test components and the associated termination of the action 'Start coordinator', the initially created test case log is finalized by a *CallBehaviorAction* calling the *OpaqueBehavior* 'FinalizeTestCaseLog'. Again, the mechanism of finalizing test case logs is out of scope of this paper.

[3]See fUML [6], Clause 7.2.2.2.3 Class, Additional Constraint 1: Only active classes may have classifier behaviors. We found out that the fUML engine we used allows for setting classifier behaviors for passive classes, too, but those passive classes are not started by the instantiation of the composite structure parts. We are aware that we need to find a better solution for this purposes in the future.

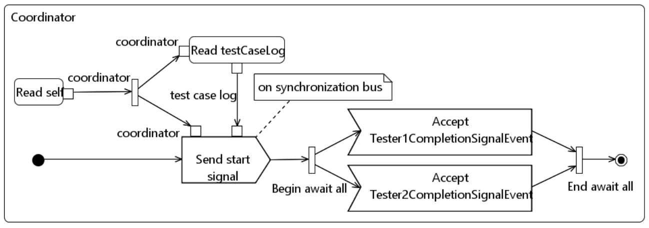

**Figure 4: Coordinator behavior**

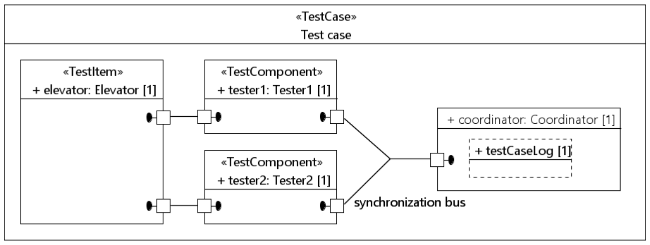

**Figure 5: Compiled *PSTM* executable test configuration**

## 5 MAPPING THE TEST CASE BEHAVIOR

The *Interactions* that represent the behavior of a *PITM* test case undergo the most extensive transformation. The resulting *PSTM* behavior is distributed among the test case itself for setting up the test environment including the *coordinator*, and the individual test components. Since the first mapping has already been described in the previous section, the subsequent sections deal with the generation of the individual test component behaviors.

Examples of the *Activities* implementing the test case and the coordinator behavior are shown in figures 3 and 4. The individual phases of the displayed behavior are described subsequently.

### 5.1 Mapping test actions

Each *PITM Lifeline* yields an ordered list of associated *MessageOccurrenceSpecifications* (*MOS*), which, together with the corresponding *Messages*, represent the test actions to be taken.

In order to derive a test component's behavior from a *Lifeline*, these covering *MOS* are iterated over and each test action is transformed into a flow. The sequence of such flows, enclosed by an initializing and a concluding flow for synchronization and logging, defines the ultimate behavior of the test component. Depending on the *messageSort* and whether the *MOS* represents a *sendEvent* or *receiveEvent*, different mappings are applied to generate the appropriate actions. Each flow representing a test action can be divided into four phases, each represented by a sub-flow:

(1) Signaling for synchronization (where applicable)
(2) Communication with the test item
(3) Logging
(4) Signaling for synchronization (where applicable)

Details on the different phases are given in the subsequent sections. Section 5.1.1 covers the communication with the test item (2), section 5.1.2 covers the logging phase (3) and section 5.1.3 covers the synchronization mechanism in (1) and (4).

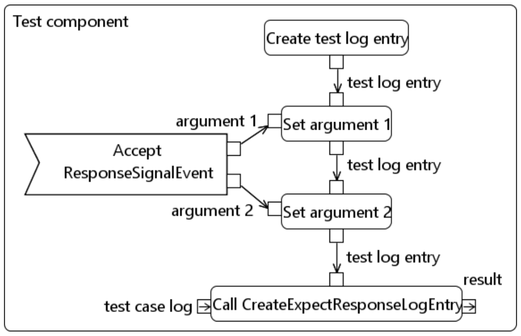

**Figure 6: Execution and logging of a CreateStimulusAction**

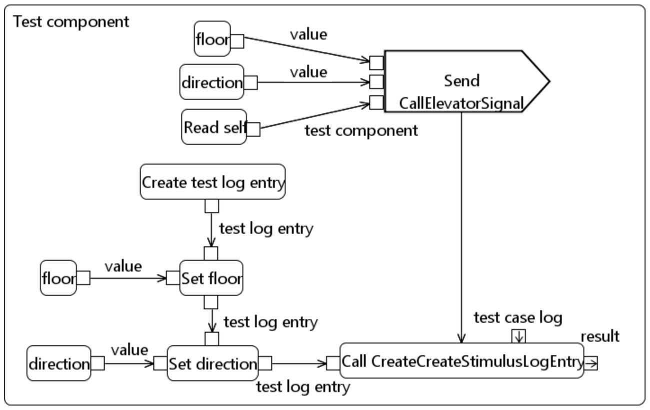

**Figure 7: Execution and logging of an ExpectResponseAction**

#### 5.1.1 *Sending and receiving Signals.*

For *sendEvents* of *Messages* that are the sort of *asynchSignal*, the transformation results in a *SendSignalAction* with one *InputPin* for every argument of the respective *Message* as well as appropriate *ValueSpecificationActions* connected by *ObjectFlows*. For *receiveEvents* of such *Messages*, the transformation results in an unmarshalling *AcceptEventAction* with one *OutputPin* for every argument of the respective *Message*. The values of the latter will be used to create an appropriate log entry as described in section 5.1.2. Examples of both flows are shown in figures 6 and 7.

#### 5.1.2 *Logging.*

Whenever a test action of the types *«CreateStimulusAction»* and *«ExpectResponseAction»* is being executed, a matching test log entry should be created. The implementing flow starts with a *CreateObjectAction*, creating an object of the type of the TestLogEntryStructure that belongs to the test action. Subsequently, *AddStructuralFeatureValueActions* populate the fields of the test log entry. In case of a *«CreateStimulusAction»*, the values are specified by *ValueSpecificationActions* in accordance with the sent Signal (see figure 6). In case of an *«ExpectResponseAction»*, the values are read from the actually received Signal (see figure 7). Finally, the object is passed to a *CallBehaviorAction*, calling the appropriate opaque behavior to persist the log entry.

*5.1.3 Synchronization of GeneralOrderings.* In order to reproduce the effect of *GeneralOrderings* (see Figure 1) in the test case behavior, a signaling mechanism was implemented. For that purpose, each *GeneralOrdering* is transformed into a corresponding *Signal* and *SignalEvent*. Test components whose lifelines are source or target of *GeneralOrderings* can then send or wait for such *Signals* and *SignalEvents* on the *synchronization bus* in order to synchronize their actions. When transforming a test action, the *toBefore* and *toAfter* associations of the corresponding MOS are evaluated to determine whether the test action must be synchronized. In case there are one or more *GeneralOrderings* found on the *toBefore* association, a flow consisting of parallel *AcceptEventActions* for all *SignalEvents* corresponding to the respective *GeneralOrderings*, is inserted at the beginning of the test action flow. Accordingly, in case there are one or more *GeneralOrderings* found on the *toAfter* association, a flow consisting of parallel *SendSignalActions* for all *Signals* corresponding to the respective *GeneralOrderings*, is inserted at the end of the test action flow.

## 6 IMPLEMENTATION AND EXECUTION

We implemented the described mapping with Eclipse QVTo. As execution engine, we utilized Eclipse Papyrus Moka. As Moka implements the CS_DefaultConstructStrategy from PSCS, it only offers support for binary *Connectors* by default. In order to support also n-ary *Connectors* as they are an integral part of our synchronization mechanism, we implemented a custom construction strategy.

We successfully executed the *PITM* test cases against the elevator use case. Test logs of the test case execution were successfully captured during execution. Details of UTP test logs were subject to our previous work [13]. Verdict calculation was done using an external Java implementation of the default UTP arbitration specifications, but had to be excluded from this paper due to space limitations.

## 7 CONCLUSION

In this paper, we described an approach to generate fUML test models from UTP test models for eventual testing of fUML systems. We addressed both, mapping rules to setup the test environment as well as mapping rules to execute the test case behavior. Therefore, we described a mapping from *PITM Interactions* to *PSTM Activities*. Using UTP for testing fUML systems is a new approach.

Even though the result demonstrates the feasibility of our approach towards executable UTP, a few aspects need closer discussion and further improvement.

We currently do not distinguish between test and system interfaces. The test configuration relies on the interfaces offered by the fUML system. This is not necessarily a shortcoming, however, continuous test design should rather define dedicated test interfaces to keep independence of any technical details of the SUT to ensure easier maintenance of the test cases. Maintenance is a key success factor in continuous test design. The reason why we do not utilize test interfaces is the lack of an adaptation layer, that is capable of mapping logical test interfaces to technical system interfaces. Supporting test interfaces and providing an adaptation layer has a high priority for our approach in the future.

Furthermore, we only support Signals and Receptions for communication between test components and the test item. Signal sending was for a long time the only supported events by fUML. With the upcome of PSSM, fUML was extended to support *CallEvents*, too. Therefore, future work will be spent on integrating Operations and CallEvents in addition.

Finally, we do not support the UTP concept *test execution schedule*. A test execution schedule is able to base the execution order of test cases within a test set on certain condition. This might be as simple as a sequence of execution (e.g., test case 1 must be executed before test case 2) or as something complex as conditional execution (e.g., if test case 1 concludes with verdict *pass* execute test case 3, otherwise test case 2). The related *«TestExecutionSchedule»* extends *Behavior* so that test execution schedules could be provided as fUML-compliant Activities as well. This would in fact decrease the effort for supporting them in our approach.

## ACKNOWLEDGMENTS

The work described in this paper was funded by the ITEA 3 TESTOMAT Project (no. 16032).

## REFERENCES

[1] Paul Baker, Zhen Ru Dai, Jens Grabowski, Oystein Haugen, Ina Schiefdecker, and Clay Williams. 2007. *Model-Driven Testing âĂŞ using the UML Testing Profile.* Springer, Heidelberg.

[2] Florin Craciun, Simona Motogna, and Ioan Lazar. 2013. Towards Better Testing of fUML Models.

[3] David Frankel. 2003. *Model-Driven Architecture.* OMG PRESS.

[4] Beatriz Lamancha, Macario Usaola, and Mario Velthius. 2009. Towards an automated testing framework to manage variability using the UML Testing Profile. In *2009 IEEE International Workshop on Automation of Software Test (AST'09) co-related with the International Conference on Software Engineering (ICSE) 2009.* Vancouver, Canada.

[5] Stefan Mijatov, Philip Langer, Tanja Mayerhofer, and Gerti Kappel. 2013. A Framework for Testing UML Activities Based on fUML. In *2013 10th International Workshop on Model Driven Engineering, Verification and Validation (MoDeVVa) co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013).* Miami, USA.

[6] Object Management Group (OMG). 2012. Semantics of a Foundational Subset for Executable UML Models (fUML). *The Object Management Group* October (2012), 441. http://www.omg.org/spec/FUML/

[7] Object Management Group (OMG). 2017. Action Language for Foundational UML (ALF). *The Object Management Group* March (2017). https://www.omg.org/spec/ALF/1.1/

[8] Object Management Group (OMG). 2017. Precise Semantics of UML State Machines Structures (PSSM). *The Object Management Group* March (2017). https://www.omg.org/spec/PSSM/1.0/Beta1

[9] Object Management Group (OMG). 2018. Precise Semantics of UML Composite Structures (PSCS). *The Object Management Group* March (2018). https://www.omg.org/spec/PSCS/1.1/

[10] Object Management Group (OMG). 2018. UML Testing Profile (UTP). *The Object Management Group* March (2018). https://www.omg.org/spec/UTP/2.0/Beta1

[11] Alin Stefanescu, Marc-Florian Wendland, and Sebastian Wieczorek. 2010. Using the UML testing profile for enterprise service choreographies. In *2010 IEEE 36th EUROMICRO Conference.*

[12] Marc-Florian Wendland. 2016. Towards Executable UML Interactions based on fUML. In *Proceedings of the 4th International Conference on Model-Driven Engineering and Software Development.* SCITEPRESS - Science and and Technology Publications, 405–411. https://doi.org/10.5220/0005809804050411

[13] Marc-Florian Wendland, Niels Hoppe, Martin Schneider, and Steven Ulrich. 2018. Extending the UML Testing Profile with a fine-grained test logging model. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST).* Vasteras, Sweden.

[14] Schieferdecker I. Din G. Zander J., Dai Z.R. 2005. From U2TP Models to Executable Tests with TTCN-3 - An Approach to Model Driven Testing. In *Khendek F., Dssouli R. (eds) Testing of Communicating Systems. TestCom 2005. Lecture Notes in Computer Science, vol 3502. Springer, Berlin, Heidelberg.*