Validating UML Simulation Models with Model-Level Unit Tests

Isabel Dietrich, Falko Dressler, Winfried Dulz, Reinhard German

Computer Networks and Communication Systems
Department of Computer Science, University of Erlangen, Germany
{isabel.dietrich,dressler,dulz,german}@informatik.uni-erlangen.de

ABSTRACT

We describe model-level unit tests for model-driven simulation based on UML models. We refer to the well-known unit testing method and apply this concept on a higher abstraction level, that is on UML simulation models. The concept of model-based simulation has become more and more popular throughout the last years. This trend is fostered by the availability of tools that automatically transform UML models into executable simulation code. Typically, both functionality and behavior are modeled in UML, whereas debugging and validation are mainly an issue of investigating the executable code. We contribute to the field of model-driven simulation by defining a novel testing method. Our method allows to use UML to specify *model-level unit tests* in order to validate simulation models defined with UML. In addition, we describe the translation, execution and evaluation of these tests within the framework *Syntony*. In this paper, we show the principles of this method and discuss its use in the scope of our simulation framework *Syntony* as well as its general applicability. Our implementation allows to compile and to execute the test code in combination with the simulation code. In spite of the high abstraction level, the full functionality of unit testing is provided and the modeler can rely on automated test case generation and execution. After execution of the tests, the achieved test coverage is computed as a measure for the test quality.

Categories and Subject Descriptors

I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Performance

Keywords

Unit test, simulation, UML, validation

SIMUTools 2010 March 15–19, Torremolinos, Malaga, Spain.

1. INTRODUCTION

Simulation projects, like most other software projects, face the problem of validating that the implemented simulation model behaves as intended. In the domain of software engineering, many methods have been developed to help developers to deal with this problem. A well-known approach for assuring the quality of a software system at different levels of detail is testing. There are numerous different methods for testing, focusing on different aspects of the software project.

At the lowest level, unit testing focuses on validating that the smallest possible software components, which are called units, are implemented correctly. Unit tests are typically written by the developer of a unit, often in parallel, or even in advance, to the development of the unit itself. In writing a unit test, the developer defines the expected behavior of a unit by specifying a set of input values and the expected outputs. In many cases, input values represent border cases, for example the lower and upper bounds of a parameter's valid range. If a unit test fails, the test infrastructure provides the developer with detailed feedback where, and under which conditions, the test failed. Unit tests that are executed frequently, and in the best case automatically, can therefore provide early information about any problems concerning the tested unit. In addition, unit tests simplify refactoring because they provide a method to validate that a module still works even after refactoring. Unit tests also simplify the integration of multiple modules, because they provide a certain level of confidence that the individual modules perform correctly.

It is possible and sensible to apply unit testing to simulation projects. As most simulation models consist of software modules, it might be possible to apply existing unit testing frameworks without modification.

In model-driven simulation, however, the situation is different. In an UML-based simulation model, for example, the modeler describes functionality and behavior in form of UML diagrams. The models are translated to programming language code by the simulation framework, but the modeler should not need to know details about the generated code or the generated interfaces. Therefore, the modeler is not able to write test cases in the target simulator's language.

The main contribution of our paper is to describe a novel *model-level unit testing* technique. This technique allows to use UML to specify unit tests in order to validate simulation models defined with UML. In addition, we describe the translation, execution and evaluation of these tests within the framework *Syntony*. Our implementation also provides the comforts available in existing unit test frameworks, like

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8629
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8629

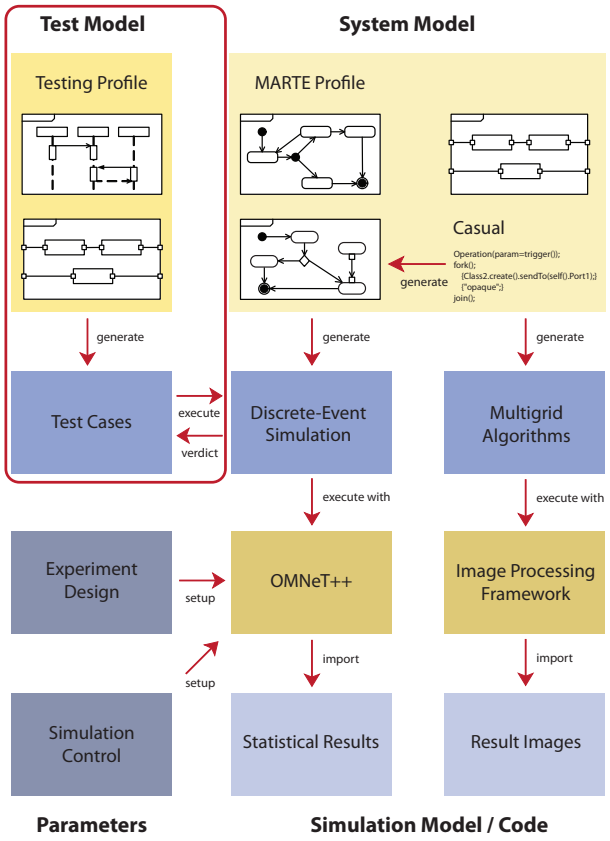


Figure 1: Diagram types and model transformations in *Syntony*

automated test execution and detailed failure reports.

In this paper, we present our methodology and the integration into our simulation framework *Syntony* [2]. Figure 1 shows the capabilities of *Syntony*. On the right-hand side, you can see that *Syntony* currently supports code generation based on standard-compliant UML models for two systems: discrete-event simulation and multigrid algorithms. Input models may consist of state machines, activities and composite structures, following the paradigm of communicating automata. The MARTE profile can be used to specify performance attributes and measures. Syntony fully automatically translates UML models to executable code for the simulation engine OMNeT++, and to code for a numerical image processing framework. Various techniques for simulation control, design of experiments, and result analysis are available from the graphical user interface which is based on Eclipse.

The test-related extensions are shown on the left-hand side (highlighted in the red box). We create test models consisting of sequence diagrams and composite structure diagrams. We use the UML testing profile [6] to annotate which elements perform which function in the test. We present the details in Sections 3.1 to 3.3. In Section 3.4, we present our transformation algorithms that automatically generate test code for the simulation engine OMNeT++. This test code may be compiled and executed in combination with the simulation code that was generated for the system model. After the tests have been executed, the achieved test coverage is computed and presented to the user together with the resulting test verdict. Section 3.5 contains more information about the integration into *Syntony*. With a few modifications, our method is also applicable to other simulation tools. This is described in Section 3.6. We further present a case study in Section 4 and give some concluding remarks in Section 5.

2. RELATED WORK

Our method shares common aspects with mainly the two fields: unit testing and simulation validation.

Unit testing refers to the lowest level of abstraction that is tested – namely the smallest testable units. In object-oriented programming, these units are usually classes. In most cases, unit testing involves writing test code to test the system code. As we build UML system models, we are concerned with creating UML test models. There exists a vast body of literature about unit testing in general, and about the various unit test frameworks in particular. One of the earliest papers was published by Hamlet [4] in 1977. The practical benefits of unit testing are discussed for example by Ellims et al. [3]. An important addition to unit testing are test adequacy criteria, i.e. criteria that allow to estimate the goodness of a set of test cases. An extensive survey about test adequacy criteria has been published by Zhu et al. [12]. The UML testing profile [6] shows how unit tests may be created with UML. In this paper, we move the idea of unit testing to a higher abstraction layer, that is simulation models described in UML.

In the area of simulation validation, testing is done during the implementation verification phase. According to Sargent [9], methods from software engineering should be applied during this phase to ensure the correctness of the implemented simulation. Our method is an attempt to apply a technique from software engineering, unit testing, in an effort to support implementation verification for model-driven simulation. As a side remark, Sargent [8] already pointed out in 1986 that graphical models may be a useful tool in model validation.

We adapt the concept of unit testing to provide a validation mechanism for model-driven simulation. As we specify the test cases in UML, a possible extension of our method is to apply techniques known from the field of model-based testing to derive the test cases. The term model-based testing is commonly used for approaches that derive the test cases from an abstract behavioral model of the system under test. A good survey of model-based testing methods has been presented by Dias Neto et al. [1]. They analyze more then 70 approaches and compare them with regard to testing level, level of automation, type of behavior models used for the system model, available tool support, and complexity of the approach. A good example is the work by Pickin et al. [7] who present an approach for deriving UML test cases based on UML system models. The cited papers in the field of model-based testing focus on transforming abstract UML descriptions to test cases. In contrast, we specify the test cases themselves in UML. Therefore, we transform test cases defined with UML to an executable simulation program. We certainly envision that model-based testing approaches will be used in addition to our method. In that case, the target language for the test case generation from the model-based testing methods would be UML instead of a regular programming or test language. Our method would

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8629
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8629

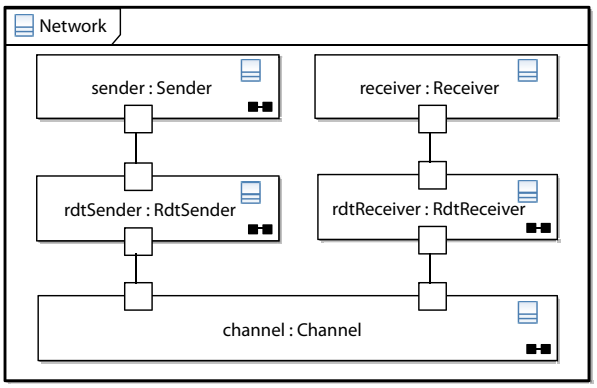


Figure 2: Structure of the stop-and-wait model

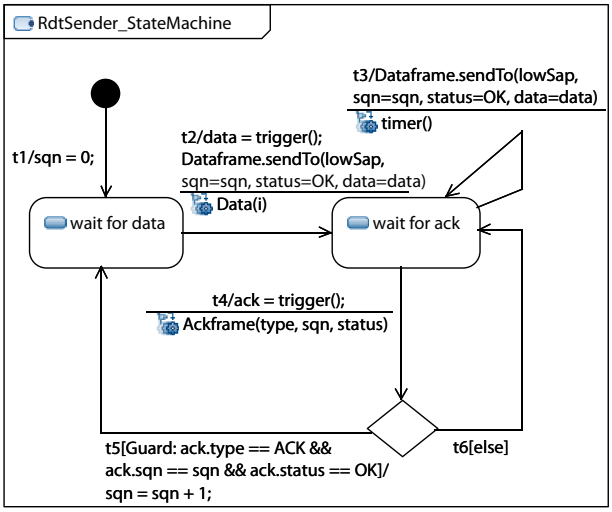


Figure 3: Statemachine for the RdtSender in the stop-and-wait model

then transform and execute the UML test cases.

When compared to industry tools like Rational Rhapsody model-driven testing[1], our approach stands out in that standard tools are not able to generate discrete-event simulations, and therefore the testing approaches cannot be applied directly to model-driven simulation. In addition, our method is fully compliant to both UML and the UML testing profile.

3. MODEL-LEVEL UNIT TESTS

In this section, we describe our method for automated component testing of UML simulation models. We provide basics about unit testing and UML sequence diagrams in Sections 3.1 and 3.2, respectively. We then introduce the general ideas of our methodology in Section 3.3 and the concrete transformation algorithms in Section 3.4. We show how we integrated our method into the *Syntony* framework in Section 3.5. Finally, we explain how our method can be applied to other simulation tools in Section 3.6.

Throughout this section, we use a model of the stop-and-wait protocol [5] as an example to illustrate the various concepts. The name stop-and-wait refers to a family of protocols where the sender stops transmission and waits for an acknowledgment after every transmitted data frame. Typically located on the data link layer, stop-and-wait protocols are responsible for providing reliable data transfer (RDT) by handling error detection, receiver feedback and retransmissions. As shown in the UML composite structure diagram of Figure 2, our model consists of a sender and receiver acting as a simple data source and sink. They are connected via ports to the protocol units RdtSender and RdtReceiver that realize the protocol behavior on the sending and receiving side. The protocol units are connected to a simple channel model that allows to delay, alter or drop frames. The RdtSender's behavior is shown in a UML state machine diagram in Figure 3. Upon reception of data from the sender – indicated by transition *t2* – the RdtSender sends a single data frame and then remains in the state *wait for ack* to wait for an acknowledgment frame (ACK) before sending the next frame. It retransmits the original frame if an ACK does not arrive within a specified timeout (transition *t3*), or if the ACK indicates that the frame was damaged (transition *t6*). Transition *t5* indicates that a correct ACK has arrived, and that transmission of the next data frame may start. The model uses sequence numbers to enable checking for duplicate data or ACK frames. In the following, we will show how test cases modeled in UML can be used to gain some confidence that the system model does indeed behave as it should.

[1] `http://www.ibm.com/software/awdtools/rhapsody/`

3.1 Unit Testing

In the following, we introduce the testing concepts and terminology relevant for our method. As we use the UML testing profile [6] to annotate our models with information for testing, we restrict our introduction to the terminology used there.

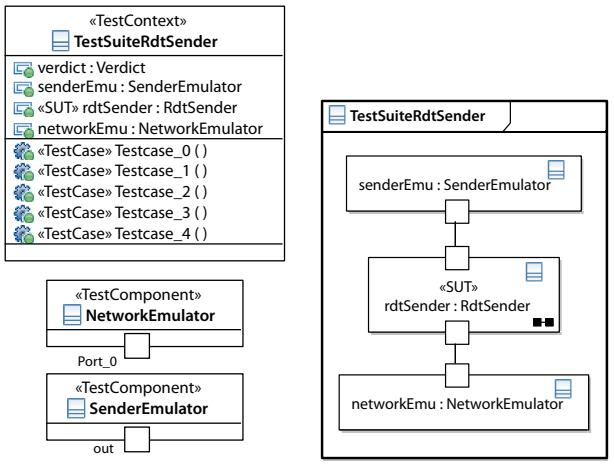
The UML testing profile was created to address black-box conformance testing. This means that the goal is to verify that the system under test conforms to a given specification, while the internals of the system under test are not known during the test. The profile provides mappings to two of the major test infrastructures: TTCN-3 and JUnit.

The *System Under Test (SUT)* is the entity to be tested. It is treated as a black box and can only be exercised via its interface operations and signals. In our example, the SUT is the RdtSender module. Its behavior (shown in Figure 3) is therefore not known to the test infrastructure.

The concrete test objective, i.e. the manner in which the SUT is exercised, is specified by the *test components*. Test components are connected to the SUT's interfaces and thus replace some part of the system model during the test. In this way, the test system gains control in triggering specific inputs to the SUT, and analyzing the output of the SUT. If the output conforms to the expected result, the verdict PASS is given, otherwise the verdict FAIL indicates a failure situation. Figure 4(b) shows how two test components, senderEmu and networkEmu, are connected to the two interfaces of the SUT. The senderEmu is used to emulate the behavior of the sender, and the networkEmu emulates the behavior of the channel and the entire receiving side of the model.

A *test case* is an operation that specifies how a set of test components interact with a SUT. A test case therefore con-

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8629
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8629



(a) Class Diagram (b) Composite Structure Diagram

Figure 4: Test context for the RdtSender module. SenderEmulator and NetworkEmulator are test components interacting with the system under test, i.e. the RdtSender.

tains a specification of the inputs given to the SUT by the test components, and the expected outputs. The specification may include additional constraints, such as timing information. It may also indicate valid behavior alternatives, or point out invalid outputs. Typically, a test case is given as a valid sequence of interactions between the SUT and the test components. A test case results in a *verdict*. PASS and FAIL indicate that a test case passed or failed. INCONCLUSIVE indicates that no conclusion can be drawn about the test outcome, and ERROR indicates that an error occurred that does not lead to test failure, but inhibits further execution (like an unhandled exception or a segmentation fault).

SUT, test cases, and test components are gathered in the *test context*. The test context describes how the test components interface with the SUT, and which test cases have to be executed. Figures 4(a) and 4(b) both contain a representation of the test context in our example. Figure 4(a) shows the SUT and the test components as well as a list of available test cases in a class diagram. Figure 4(b) shows the connections between test components and SUT, i.e. the internal structure of the test context, in a composite structure diagram.

Figure 5 shows how these concepts are represented as stereotypes in the UML testing profile. A test case can be modeled by any UML behavior or operation. Both test components and test contexts are modeled with structured classifiers (a class is one kind of structured classifier). The system under test is represented as a UML property. Finally, the verdict is a simple enumeration of the possible test verdicts.

Unit tests themselves offer no possibility to judge how thoroughly a unit is exercised by the test cases. This does not matter much as long as there are failing test cases because a test failure always reveals new knowledge about the system, i.e. that there is an error in the unit. However, as soon as all test cases pass, it is unclear why: either because the unit does not contain any more errors, or because

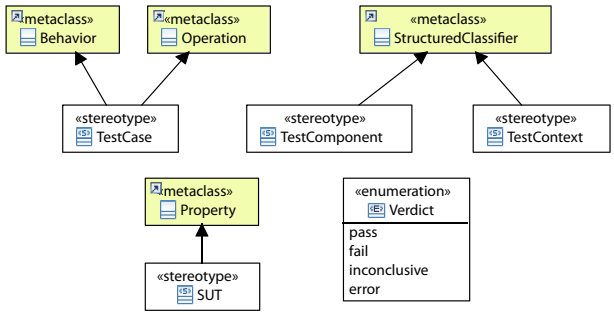


Figure 5: The UML Testing Profile (showing the portion currently supported by our method)

the unit is not tested thoroughly enough. Test adequacy criteria [12] are one method to close this gap. The most commonly known test adequacy criteria are statement coverage, branch coverage and path coverage. The coverage of a test can be computed by looking inside the unit during test execution and recording which parts of the unit are executed. Statement coverage checks if all statements in a unit are executed at least once. Branch coverage checks the same for all execution branches. Path coverage checks if all possible execution paths are executed at least once. Of the three, path coverage is the strongest criterion, followed by branch coverage. Statement coverage is the weakest. In this paper, we use branch coverage based on transitions in state machines.

3.2 UML Sequence Diagrams

A UML interaction describes the interactions occurring between parts in a system, for example message exchanges or operation calls. It does so by specifying one possible sequence of event occurrences. For a typical system, many interactions would be necessary to capture all alternatives of the entire behavior. There are several diagram types that can be used to visualize interactions. The most common type are sequence diagrams.

Figure 6 shows a sample sequence diagram realizing one test case for the RdtSender module. It consists of three lifelines, senderEmu, rdtSender, and networkEmu. Each lifeline represents one participant in the test case. The events occurring in the interaction are ordered from top to bottom on the participating lifelines. In UML terminology, events on a lifeline are called interaction fragments.

Message *1: Data* indicates that the senderEmu starts by sending a Data signal to the rdtSender. This message connects two interaction fragments: one for the sending and one for the reception of the signal. The rdtSender is expected to react to a Data signal by sending a specific Dataframe to the networkEmu (message 2). The parameter values for the signals are not shown in the figure, but are accessible through the properties view in the UML editor. The networkEmu responds (message 3) with an erroneous Ackframe. As indicated by the loop fragment containing both messages 2 and 3, this is repeated for three times. Fragments containing other fragments (like the loop fragment in the figure) are called combined fragments. There are several types of combined fragments, depending on the fragment's operator, for example alternative, parallel, and negative fragments. After execution of the loop fragment, rdtSender sends another

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8629
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8629

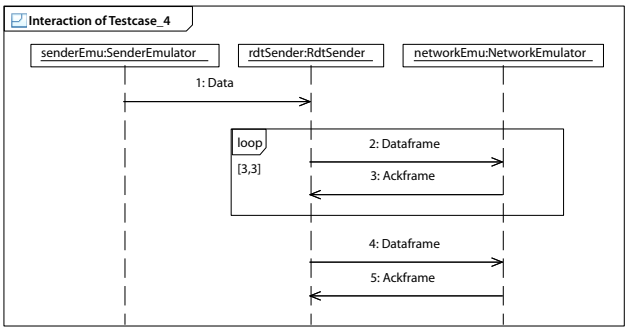


Figure 6: Sample test case for the RdtSender: an erroneous Ackframe is sent back to the RdtSender inside the loop. The reaction expected from the RdtSender is that it retransmits the original Dataframe for an unlimited number of times.

correct Dataframe (message 4). The networkEmu finally responds with a correct Ackframe (message 5). This concludes the test case.

3.3 Methodology

We assume that our system model consists of composite structure, state machine, and activity diagrams. To this system model, we add a test model consisting of UML sequence and composite structure diagrams.

In the next step, both the system model and the test model are transformed to executable code for the target simulation engine. We use OMNeT++ for this, but since the generated code does not rely on specific features of the OMNeT++ simulator, any simulation engine would work. Finally, the test cases are executed in the simulation environment, and the test verdict is recorded and presented to the user.

Along with the test verdict, we record the test coverage. If the system under test contains a state machine, we compute the branch coverage based on the transitions in the state machine. We then compute the test coverage as the number of executed transitions divided by the total number of transitions in the state machine. We compute this number for each test case separately, and also for the entire test context representing the total test coverage for the system under test.

There are several advantages of using the simulation engine as platform for unit testing. First, we do not have to invent a new concept of simulation time for the test environment, because the simulation engine already knows about simulation time. Second, the mechanisms to send and receive messages are already present in the simulation engine; therefore, we do not need to map them to a new mechanism in the test enviroment. Third, we need to transform the system model only to a single target system – the simulation engine – instead of two. Finally, this is what the existing unit testing infrastructures do: they use the system's target platform also as test platform.

3.4 Transformation Algorithms

We now present the transformation algorithms we have developed to transform the test models described above into executable code for the simulation engine OMNeT++ [11], which is based on C++ and was designed to support efficient network simulation. OMNeT++ distinguishes two kinds of classes, compound and simple modules. Classes that are composed of other classes and the connections between them, are called compound modules. They are specified in the network description language (NED). Atomic classes with an associated behavior are called simple modules. They are written in C++, and are accompanied by a short NED description of their configuration parameters and interfaces. We refer to [2] for details on the transformation of the system model. The transformation runs fully automatically, and *Syntony* generates compiled code that can be executed immediately without manual intervention.

3.4.1 Test Control

We implement test control mechanisms on two levels. The first level is placed inside the generated simulation; it governs recording of test verdicts from single test cases. The second level controls which test cases are executed and takes care of presenting the verdicts from all executed test cases to the user. We place the second level in *Syntony*'s graphical user interface and discuss it in Section 3.5.

For the first level, we create a generic test control module in OMNeT++. Each test component registers itself with the test control just before the test case execution is started. As soon as a test component knows its verdict, it records it with the test control. If the verdict is any other than PASS, the test is terminated immediately. Otherwise, the test control terminates execution only if all known test components have submitted their verdicts. Before termination, test control records the test verdict and, if available, detailed information about the verdict's cause, in an OMNeT++ scalar file that is available for analysis later on.

3.4.2 Test Component

A test component's behavior is specified only through the lifelines in which it participates. We do not attempt to unify these behaviors in a single OMNeT++ class; instead we create a separate simple module in OMNeT++ for every lifeline representing a test component. For example, in the test case shown in Figure 6, there are two lifelines corresponding to two test components (senderEmu and networkEmu), and one lifeline corresponding to the system under test (rdtSender). Therefore, we create two simple modules, one for each of the two test component lifelines. If a test component is used in more than one test case, a new simple module is created for each test case a test component is used in.

Basically, the behavior we create walks along its lifeline from top to bottom. The behavior for the networkEmu from Figure 6 therefore starts by waiting for the reception of a Dataframe. It then responds with a specific Ackframe, then waiting again for the next Dataframe, and so on. At any point, the next event on a lifeline is processed immediately if possible. If the next event has to be waited for, like a receive signal event or a timeout, execution is interrupted and resumed only when a new event is received. In these cases, the current position on the lifeline has to be recorded. We generate an enumeration listing all fragments on the lifeline for every lifeline representing a test component. A class attribute stores the current position using this enumeration. When all events on a lifeline have been processed, the behavior records a PASS verdict with the test control module.

A number of different event types, combined fragments, and constraints may occur on a lifeline. In the following,

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8629
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8629

we give a detailed description for send signal events, receive signal events, loop and alternative fragments, and duration constraints. The remaining elements can be handled in a similar way.

Send Signal Event: Signal events do not only specify the type of signal that is to be sent or received, but also the values for the signal's arguments and the connection on which the signal should travel. When a send signal event is next on a lifeline, the behavior simply creates a new signal of the specified type. It also sets the given arguments, and sends the signal to the port indicated by the connection. Then, it updates the position on the lifeline and immediately starts processing the next event.

Receive Signal Event: Processing of a receive signal event is triggered by the reception of a signal. The received signal is subject to several tests. First, the behavior checks the type of the received signal against the type specified in the model. If the check fails, it records a FAIL verdict with the test control. The detailed report for this verdict includes information about the lifeline, the event, and the received and expected type of the signal. If the signal type is correct, the behavior checks each of its argument values against the argument value specified in the model. Again, a failed check results in a FAIL verdict. In addition to the information above, the detailed report also includes the received and expected value of the argument. If the signal's type and all of its argument values are correct, the behavior updates the position on the lifeline and starts processing the next event.

Loop Fragment: The processing of combined fragments depends on the specified operator and the guards on the operands. A loop fragment has a single operand. Its guard can specify both the maximum number of loop iterations and an abortion condition that is checked after a minimum number of iterations has been performed. For example, the loop fragment in Figure 6 specifies a maximum iteration count of three, without an additional abortion condition. The operand in this example consists of four interaction fragments, namely one sending and one receiving event on the rdtSender lifeline, and the corresponding events on the networkEmu lifeline. The generated code keeps track of the iteration count. When the behavior reaches the last fragment in the operand, it checks the iteration count and, if applicable, the abortion condition. If the abortion condition is true, or if the maximum number of iterations has been reached, the behavior sets the lifeline position to the first fragment following the loop fragment. Otherwise, it sets the lifeline position to the first fragment in the loop operand.

Alternative Fragment: Alternative fragments may be used to model test cases where the system under test exhibits random behavior, i.e. where the system under test may react differently depending on a random selection. In this case, each of the fragment's operands describes one valid alternative. The generated code then checks if one of the operands matches the system under test's run-time behavior. It records a FAIL verdict if no operand matches.

Duration Constraint: Any two interaction fragments can be constrained by a duration constraint. In this way, a minimum or maximum duration between the two fragments can be specified. At the starting point of a duration constraint, the behavior records the current simulation time. At the end point of the constraint, the behavior then checks the recorded duration against the specified duration interval. If the check fails, it records a FAIL verdict indicating both the expected and the recorded durations. Figure 8 shows the detailed report for a test failure caused by a violation of a duration constraint.

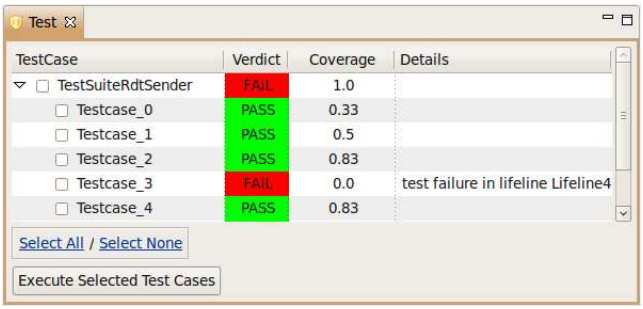


Figure 7: *Syntony* GUI showing the test verdict as well as the coverage for several test cases

3.4.3 Test Case, Test Context

For every test case, we create a separate network definition. This network composes all modules participating in the test case's interaction, typically the SUT (taken from the system model) and the modules created for the test component lifelines. The test context's internal structure specifies how the modules are connected with each other.

3.4.4 Test Coverage

We measure the test coverage of a given set of test cases by recording which transitions were executed. To record execution of a transition, the transformation creates a subclass of the system under test and overwrites all transition execution methods. The new methods record the execution in an OMNeT++ scalar file and then simply call the original method. The resulting scalar file is analyzed automatically after the test execution completes (see Section 3.5.1).

3.5 Integration in Syntony

We integrated the test method presented in the previous sections into our framework *Syntony*. If a model contains test cases, they are translated and compiled together with the system model. Then, the available test cases are presented in a separate view in the Eclipse platform. The view is shown in Figure 7.

3.5.1 Test Control

The test control mechanisms in the graphical user interface handle test case selection, execution, and presentation of test verdicts and coverage. All test cases in a given model are gathered in a list. The user may select all test cases for execution, or choose single test cases. The test control creates a script governing the execution of the selected test cases. For every selected test case, the script contains a call to the generated simulation with the parameters appropriate for the test case. After the test execution has completed, test control parses the OMNeT++ scalar files generated by the run-time test control and extracts the test verdict and detailed messages for every test case (Figure 8). The aggregated verdict for the test context is the worst verdict of the contained test cases. Both verdict and details are presented to the user together with the original test case list.

The coverage achieved by the executed test cases is also computed and displayed. In addition to the coverage per-

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8629
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8629

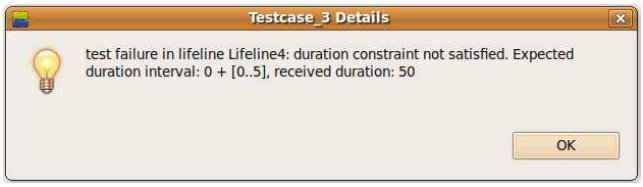


Figure 8: *Syntony* GUI showing the detailed message for a failed test case – in this case, due to a violated duration constraint.

Figure 9: *Syntony* GUI showing the coverage details for test case 5.

centage, the user also gets to know which transitions were not executed by the tests. These transitions are shown with their qualified name which indicates the exact position of a transition in the model. Figure 9 shows how these transitions are presented to the user.

Currently, the user interface does not offer the possibility to start test case execution automatically. However, this is merely a programming exercise. Automatic execution could be triggered, for example, by a timer or by changes to the model. Both variants can be implemented in Eclipse.

3.6 Applicability in other Areas

We implemented our method for UML-based simulation with *Syntony*. *Syntony* offers the possibility to couple UML simulation models with native OMNeT++ models. In this case, it is desirable to use only a single unit testing framework for the coupled model instead of one for the UML model, and one for the existing C++ code. In this section, we explain how our method can be adapted to test native OMNeT++ models with UML test cases. Of course, the adaptation is not strictly limited to OMNeT++. The same ideas can be used to adapt our method to arbitrary simulation tools.

3.6.1 Application to Simulation Tools

Syntony transforms both the system and the test model to simulation code. If our method is used with native system models instead of UML system models, the system model's class structure has to be represented in UML. This is necessary to allow the UML test model to include correct references to all signals, data structures, and classes in the original model. Fortunately, there are UML editors available that allow to create class diagrams automatically from a given body of programming language code, like Java or C++. In some simulation tools, however, the required information is described with a custom language. In this case, the class structure can either be created manually, or, if the custom language's grammar is available, an automatic converter can be created. As OMNeT++ uses the NED language to describe the structure and interfaces of simulation models, we implemented a converter that allows us to create UML classes automatically from OMNeT++ ned files.

For OMNeT++, representing the class structure of the existing simulation models in UML is all that has to be done to apply our method. For other simulation tools, the test code generated by the transformation algorithms would have to be adapted to use the target simulation tool's API.

An important point is that the achieved test coverage can only be computed by our method if a complete system model exists in UML. Otherwise, traditional coverage tools such as gcov[2] can be used.

3.6.2 Application to Multigrid Algorithms

Basically, the same modifications have to be made if our method is applied to *Syntony*'s multigrid transformation. The multigrid transformation works by combining existing hardware-specific code with automatically generated high-level code. Therefore, the class structure for the existing code has to be represented in UML. Then, the code generated by the transformation algorithms has to be adapted to the API of the image processing framework. Like before, the test coverage can only be computed for those modules that are completely modeled in UML.

3.6.3 Application to Test-Driven Development

Our approach can be used without further modification to support test-driven development of UML simulation models. In that case, the user would have to follow three steps. The first step is to create the basic system structure so that the entities used in the test cases reference correct UML elements. In the second step, the user can start to create test cases and execute them (expecting test failures because the system behavior is not yet present). In the third step, the user can add behavior to the system until the test cases pass. In principle, this procedure is not different from test-driven development using, for example, JUnit.

4. CASE STUDY

We demonstrate the application of our method using our model of the stop-and-wait protocol. Although the protocol is quite simple, it has the advantage of being small enough so that it can be shown and explained in its entirety. Also, despite its simplicity and despite the effort we invested in modeling it, the test cases still revealed a few bugs in the model. This indicates that the protocol is complex enough so that useful test cases can be created for it.

4.1 Test Cases

We present a selection of the test cases for the RdtSender module in Figures 6, 10, 11 and 12. Test case 0 (shown in Figure 10) tests the basic functionality of the RdtSender module, i.e. the reception of data from the sender (message *1: Data*), the transmission of this data item in a single data frame (message *2: Dataframe*) and the reception of the corresponding ACK (message *3: Ackframe*).

Test case 3 (Figure 11) is a little more complex. It assumes that the ACK is somehow lost or delayed on the channel, and therefore the RdtSender module has to retransmit the original data frame after a certain timeout. There is a duration

[2]gcov comes with the GCC compiler `http://gcc.gnu.org`

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8629
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8629

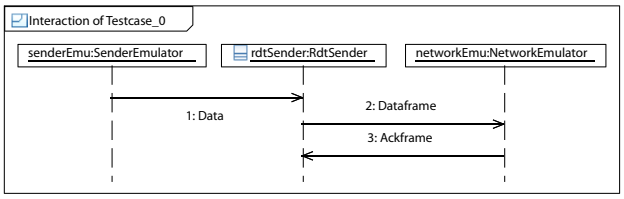


Figure 10: Test case 0 for the RdtSender: test the basic functionality when there are no errors on the channel.

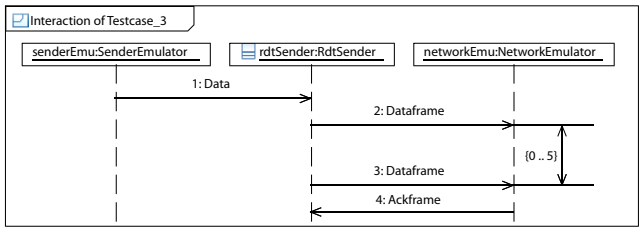


Figure 11: Test case 3 for the RdtSender: test the timeout duration between retransmissions of data frames when no acknowledgement arrives.

constraint between messages 2 and 3 indicating the allowed duration of the timeout.

The test case presented in Figure 12 tests whether the RdtSender module is able to handle transmission of more than one data item correctly. To achieve this, the sender first transmits two data items to the RdtSender (messages *1: Data* and *2: Data*). The RdtSender should then transmit the first data frame (message 3), wait for the corresponding ACK (message 4), and only then send the second data frame (message 5).

Test case 4 has already been shown in Figure 6. It checks whether the RdtSender acts correctly when it receives an erroneous ACK frame. The ACK could be erroneous because it has been modified in the channel, or because it contains a wrong sequence number. The correct action for the RdtSender is to retransmit the original data frame. It should do so for an unlimited number of times as long as it receives false ACK frames. The test case does only check that it does so for three times. This is noted in the condition on the left side of the loop fragment (messages 2 and 3). The test case also checks if the RdtSender stops the retransmissions once it receives a correct ACK frame (message 5).

When all or some test cases have been executed, the view shown in Figure 7 is updated to show the test verdict. In this case, there was a test failure in test case 3. The detailed report for this test failure is shown in Figure 8. It indicates that the duration constraint for the retransmission timeout was violated: a duration between zero and five seconds was expected, while the observed duration was 50 seconds. This failure was due to an error in the system model concerning the calculation of timeout values, and it was one of the bugs we found in our system model.

The third column in Figure 7 shows the coverage achieved by the test cases. As you can see in the first line, the test cases cover 100% of the transitions in the RdtSender module. The test cases we created for the RdtReceiver also achieve full coverage.

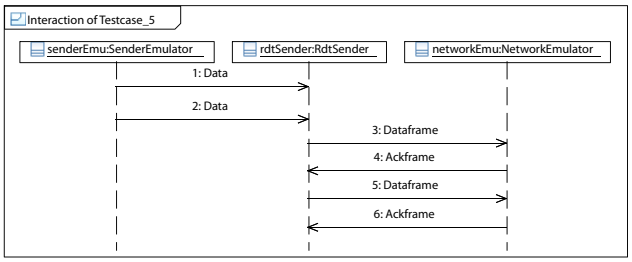


Figure 12: Test case 5 for the RdtSender: test the transmission of more than one data frame.

4.2 Usability

In the following, we discuss three aspects related to the usability of our method: the generation and compilation times for the test model, the run-time behavior, and the modeling process.

Compared to the generation and compilation times for the system model only, the inclusion of a test model naturally does cause some overhead. This overhead grows depending on the number of test cases in the test model. The generation and compilation times also depend strongly on the size of the system model. Therefore, the times given here can only serve as a ballpark figure. For the stop-and-wait model, the generation and compilation process takes about 15 seconds for the system model, and about 21 seconds if the test model is included.

The execution of the test cases runs very quickly, especially when compared with the run-time of the entire simulation. As an example, the execution of all test cases in the stop-and-wait model completes in less than two seconds. This is small enough so that the test cases can be executed frequently and automatically.

As with traditional unit tests, it is possible, or even probable, that creating the test cases for a unit takes longer than creating the unit's behavioral model. For every simulation project, or even for every individual unit, this additional effort has to be weighed against the added confidence in the model's correctness. However, this is not particular to our approach as it applies to all unit test frameworks. What is particular to our approach is that the test cases are created in UML instead of programming language code. Even for an experienced modeler, this probably takes longer than writing a corresponding unit test in code. However, we believe that the clarity added by the graphical representation, and the improved basis for communication about the test cases, make up for this additional effort.

The only real drawback of our method is that the currently available editor support for sequence diagrams leaves room for improvement. For example, neither Papyrus[3] nor the IBM Rational Software Modeler[4] allow to specify or display constraints in the graphical representation of the model. Instead, the modeler has to resort to using a tree-like view of the model which is a lot less comfortable. However, we expect that this will be improved in future versions of these UML editors.

[3] `http://www.papyrusuml.org`
[4] `http://www.ibm.com/software/awdtools/modeler/swmodeler`

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8629
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8629

5. CONCLUSIONS

Unit tests are an established test method for code-based projects. In this paper, we present a method to apply unit testing to simulation projects based on UML models. The method consists of modeling test cases in UML and automatically transforming them to test code for the target platform. In particular, we use sequence diagrams to model individual test cases. The test setup is modeled using class and composite structure diagrams. We then use our tool *Syntony* to transform the test model to C++ code for the simulation engine OMNeT++. The generated code can execute the test cases and record the test verdict for every test case. It also computes the branch coverage achieved by the test cases based on transitions in the system model. We integrated our method into *Syntony*'s graphical user interface to achieve a usability similar to existing unit test infrastructures. We applied the method to several UML simulation models, including a model of the stop-and-wait protocol. In doing so, we uncovered some previously unnoticed bugs in the models. This again demonstrates the practical usability of unit testing in general, and our method in particular.

As a next step, we plan to couple our method with model-based testing approaches. In that way, test cases would not have to be created manually, but could be generated automatically from abstract behavior models [1], or from usage models [10].

6. ACKNOWLEDGMENTS

This research was funded in part by the German Federal Ministry of Education and Research under grant number 01IA08001C.

7. REFERENCES

[1] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos. A survey on model-based testing approaches: a systematic review. In *WEASELTech '07: Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies*, pages 31–36, Atlanta, Georgia, 2007. ACM.

[2] I. Dietrich, V. Schmitt, F. Dressler, and R. German. SYNTONY: Network Protocol Simulation based on Standard-conform UML 2 Models. In *2nd ACM/ICST International Conference on Performance Evaluation Methodologies and Tools (ValueTools 2007): 1st ACM/ICST International Workshop on Network Simulation Tools (NSTools 2007)*, Nantes, France, October 2007. ACM.

[3] M. Ellims, J. Bridges, and D. C. Ince. Unit Testing in Practice. In *15th International Symposium on Software Reliability Engineering*, pages 3–13, Saint-Malo, Bretagne, France, November 2004. IEEE Computer Society.

[4] R. G. Hamlet. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.

[5] J. F. Kurose and K. W. Ross. *Computer Networking: A Top-down Approach*. Addison Wesley, 4th edition, 2008.

[6] Object Management Group (OMG). UML Testing Profile Specification, v1.0. Technical report, OMG, 2005.

[7] S. Pickin, C. Jard, T. Jeron, Y. Le Traon, and J.-M. Jezequel. Test Synthesis from UML Models of Distributed Software. *IEEE Transactions on Software Engineering*, 33(4):252–269, 2007.

[8] R. G. Sargent. The use of graphical models in model validation. In *18th conference on Winter simulation (WSC '86)*, pages 237–241, Washington, D.C., USA, December 1986. ACM.

[9] R. G. Sargent. Verification and validation of simulation models. In *39th Winter Simulation Conference (WSC 2007)*, pages 124–137, Piscataway, NJ, 2007. IEEE.

[10] S. Siegl, W. Dulz, R. German, and G. Kiffe. Model-Driven Testing based on Markov Chain Usage Models in the Automotive Domain. In *12th European Workshop on Dependable Computing (EWDC 2009)*, Toulouse, France, May 2009.

[11] A. Varga and R. Hornig. An overview of the OMNeT++ simulation environment. In *1st ACM/ICST International Conference on Simulation Tools and Techniques for Communications, Networks and Systems (SIMUTools 2008)*, Marseille, France, March 2008. ACM.

[12] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, December 1997.

Digital Object Identifier: 10.4108/ICST.SIMUTOOLS2010.8629
http://dx.doi.org/10.4108/ICST.SIMUTOOLS2010.8629