Dynamic Testing of Executable UML Models with Sequence Diagrams

TAMÁS JÁNOSI, KRISZTIÁN MÓZSI, PÉTER BERECZKY, DÁVID J. NÉMETH and TIBOR GREGORICS, Eötvös Loránd University

Executable UML makes it possible to define high-level models of software systems, which then can be inspected independently of the target platform or translated to efficient platform-specific code. This not only provides early validation, but enables modeling to even be considered an additional layer of abstraction over programs written in a general-purpose language. In both cases, reasoning about the properties of constructed models plays an important role in the development process.

To advance this area, in our paper we propose a framework supporting the definition and evaluation of sequence diagram-based test cases for executable UML models. As part of this, we introduce a textual language capable of describing assertions about the communication in and between, and the state of, system components. We also discuss a test evaluation strategy characterized by the idea of executing sequence diagrams and models alternately, and show how arising challenges like synchronization or the quantification of possible divergence can be tackled. Finally, we illustrate the feasibility of our approach by presenting our prototype implementation, embedded into an open-source xUML modeling toolchain.

1. INTRODUCTION

As software systems gradually grow both in size and complexity, their design, development and maintenance become increasingly difficult tasks. Although modern general-purpose programming languages offer a rather high level of abstraction, the structure and behavior of large-scale architectures can still be hard to manage in conventional source code. One way to address this issue is the use of UML [Object Management Group 2017], a language which enables us to model several aspects of a system in the form of high-level diagrams. With the help of dedicated tooling, appropriately defined, potentially platform-independent models can also be executed directly or translated to runnable platform-specific representations. This is the essence of executable UML (xUML) modeling [Mellor and Balcer 2002] – a method for early functional validation, or even a paradigm for developing software on a higher layer of abstraction.

Whichever aim of xUML we consider, the ability to reason about properties of constructed models is of great importance. Static analysis is an emerging possibility, however, in this paper we focus on monitoring the runtime behavior by dynamic testing. To realize this, one could observe the interpreted model or the generated platform-specific code in a black-box fashion. Yet we argue that formulating test cases as part of, and on the same layer with, defined models has the advantage of keeping the specification and implementation coupled together closely enough for simplified comparison and main-

This research has been supported by the European Union, co-financed by the European Social Fund (EFOP-3.6.3-VEKOP-16-2017-00002, Integrated Program for Training a New Generation of Scientists in the Fields of Computer Science).
Author's address: Eötvös Loránd University, Faculty of Informatics, Pázmány Péter sétány 1/C., Budapest, H-1117, Hungary. Email: {yaki96,mozsik,berpeti,ndj,gt}@inf.elte.hu. ORCID: 0000-0002-0080-8689, 0000-0002-6165-1865, 0000-0003-3183-0712, 0000-0002-1503-812X, 0000-0002-9503-9623.

In: Z. Budimac and B. Koteska (eds.): Proceedings of the SQAMIA 2019: 8th Workshop on Software Quality, Analysis, Monitoring, Improvement, and Applications, Ohrid, North Macedonia, 22–25. September 2019. Also published online by CEUR Workshop Proceedings (http://ceur-ws.org, ISSN 1613-0073)

tenance. Inspecting the feature set of UML, sequence diagrams appear to be a suitable candidate for expressing expected interaction in and between components. Regarding the evaluation of described test cases, we optimize for traceability and interactivity. To achieve these goals, our solution executes sequence diagrams directly and in parallel with models under test. Building upon these ideas, in the following we propose a framework supporting the definition and evaluation of sequence diagram-based test cases for executable UML models.

The main contributions of this paper are the following:

—a textual language for defining sequence diagram-based test cases for executable UML models,

—a methodology for evaluating these tests by executing sequence diagrams and inspected models synchronously,

—a prototype implementation of the proposed framework embedded into an open-source xUML modeling toolchain.

The rest of the paper is structured in the following way. As introduction, Section 2 summarizes related work. In Section 3 we present general decision concerns of the proposed framework, including the main ideas behind the testing language and its intended evaluation strategy. The latter are discussed in detail in Section 4 and Section 5, respectively. Section 6 introduces our prototype implementation, demonstrating the feasibility of our approach. Finally, Section 7 briefly evaluates our results and Section 8 concludes.

2. RELATED WORK

Model-based testing (MBT) [Utting et al. 2012] builds upon the idea that test cases can be generated automatically from abstract, simplified representations of the system under test, which encode its intended behavior. When the system to be tested is a UML model itself, for example interaction diagrams can be seen as the higher-level specification descriptions MBT promotes. This approach helps keep the system and its test cases close, however, the abstractional difference between tested systems and generated test cases makes traceability challenging.

Executable UML models can also be evaluated by manually examining their behavior during direct execution, a method called model debugging. Two xUML modeling tools providing this feature are BridgePoint [One Fact 2012] and txtUML [Dévai et al. 2018], where the latter even offers a visual debugger which highlights transitions in state machines during execution. One great disadvantage of this approach is that instead of automatically evaluating predefined test cases, the model has to be actively monitored while it is running.

In conclusion regarding xUML testing, currently we have to choose between well-defined test cases (MBT) and interactive traceability (model debugging). A new approach would be desirable to combine the advantages of the two discussed methods. In the following we show how this can be achieved with executable sequence diagrams.

3. METHODOLOGY

Our approach is to create executable sequence diagrams to test existing models written with an xUML tool. In this way, users could test their programs by expressing message interactions during a specific execution. This could eliminate the need for writing test programs, because the sequence diagrams themselves would serve as test cases. The level of abstraction of these test cases would be the same as of the models'.

3.1 Abstractions required to construct meaningful test cases

To achieve the abovementioned goal, a textual domain-specific language is needed to provide a way to define and execute sequence diagrams. In this language, there should be a language element which makes it possible to create the lifelines of the given diagram. Each lifeline will represent a model element. As we also want to draw sequence diagrams from the written format, we need to provide a way for users to manipulate the drawn position of the lifelines inside the textual notation.

Before executing the sequence diagram, some initializing steps are needed. Model elements have to be created and users have to define explicitly which model element is represented by each lifeline. This means that each sequence diagram needs an initializing part, which will be performed before we execute the sequence diagram itself.

During the execution, we want to be capable of investigating message exchanges and state assertions. Users should be able to define the order of sent messages during an execution with the sources and the targets of the messages. Combined fragments from the UML standard can be used in the sequence diagrams too, and messages can be defined in these fragments.

State assertions are not part of the UML standard, but as our model elements in an executable UML tool are state machines, we wanted to provide a way to check the state of a given model element at a given point of the execution.

3.2 Advantages of making the test description language textual

Textual descriptions have numerous advantages over graphical approaches [Grönniger et al. 2007]. In a graphical solution, models are usually persisted in a format that is hard or impossible to edit directly. Experience shows that the existing tools still need to evolve a lot. Furthermore, while graphical notations help understand software more than textual representations, editing graphics is usually less productive than editing text. At the same time, many high-quality text editors with editing and search-related features, as well as merge and compare tools exist, which can help the user during the development process. Moreover, graphical diagrams can be generated easily from textual descriptions to visualize the defined diagram.

3.3 Evaluating test cases

To test a model with sequence diagrams, we have to execute both the model and the diagram. To be able to test a model based on message exchanges, the execution strategy of the host xUML tool has to be message-driven. When running a sequence diagram, a given model will be executed. Each time a message exchange happens in the executed model, we have to check whether it is equal to the next exchange in the sequence diagram. Two exchanges are said to be equal if their sources and targets are respectively the same and their included messages are equal as well.

4. THE TEST DESCRIPTION LANGUAGE

As it was mentioned earlier, to create sequence diagrams the language has to provide a way to declare the lifelines of the diagram. This can be achieved with the following syntax:

```
lifeline <identifier> <positive number>
```

The positive number is an optional part of the declaration. It defines the order of the lifelines on the diagram generated from the textual description – therefore positions will be ignored during the execution.

There should be an initialization part too, where model elements can be created and linked with the lifelines. While the syntax of the model element creation is based on the host xUML tool, connecting the elements with the lifelines should look like the following:

```
<lifeline> = createLifeline(<model element instance>)
```

The lifeline will represent the given model element instance during the execution.

After the initialization, the execution itself can be described. Message exchange expectations have two different syntaxes, because we differentiate messages between model elements and between an actor and a model element.

```
assertSend(<lifeline>, <message>, <lifeline>)
fromActor(<message>, <lifeline>)
```

In the first case we have to define the source, the message and the target. In the second case the message comes from an actor, so the source does not have to be specified. If a message exchange happens during the model execution, we try to match the source, the message and the target with the currently expected message in the diagram.

State assertions can also be made with the help of a lifeline and a state as parameters.

```
assertState(<lifeline>, <state>)
```

If the sequence diagram execution starts with state assertions, the parameter model element instances have to be in the given states before the execution. Whereas if a state assertion is placed after an expected message exchange, the model element has to be in the given state after the exchange happens.

Sequence diagrams may also contain combined fragments. Currently, our language supports only the *loop*, *opt*, *alt* and *par* fragments from the UML standard. The *loop*, *opt* and *alt* fragments can be defined with conventional loop and conditional statements, and they can contain further message exchange expectations, state assertions and combined fragments. The *par* combined fragment can be used with the following language element to express parallel execution of child sequence diagrams.

```
par(<sequence diagram>+)
```

By using the language defined above, not all models could be tested. Model element instances should be observable from outside the model, as their references are needed to connect them with lifelines. This is not always feasible, since some references may only be accessible from inside the model. Our idea is to introduce dummy objects, called proxies to substitute such instances. Proxies can be thought of as special lifelines, as they behave quite similar. The language is extended with proxy creation, where the only information needed for this is the type of the object to be substituted.

```
<proxy> = createProxy(<type of model element>)
```

4.1 Example diagram

Let us demonstrate our language with an example model. The model has two elements: the factory and the customer. If a customer gets an `Order` message from the actor, it orders from the factory by sending it a `DoOrder` message. If the factory gets a `DoOrder` message and the number of orders reached a limit, it starts working until it fulfils all orders. It has a counter which indicates the number of the active orders. Every time an order is ready, it sends an `OrderReady` message to the customer who gave the order. This model can be tested for example with the following sequence diagram.

```
seqdiag factorySequenceDiagram {
    lifeline factory 1;
    lifeline customer1 2;
    lifeline customer2 3;
    lifeline customer3 4;

    initialize {
        Factory f = createFactory(3);
        Customer c1 = createCustomer();
        Customer c2 = createCustomer();
        Customer c3 = createCustomer();

        factory = createLifeline(f);
        customer1 = createLifeline(c1);
        customer2 = createLifeline(c2);
        customer3 = createLifeline(c3);
    }

    run {
        assertState(factory, WaitingForOrder);
        fromActor(Order, customer1)
        assertSend(customer1, DoOrder, factory);
        fromActor(Order, customer2)
        assertSend(customer2, DoOrder, factory);
        fromActor(Order, customer3)
        assertSend(customer3, DoOrder, factory);
        assertState(factory, Working);
        assertSend(factory, OrderReady, customer1);
        assertSend(factory, OrderReady, customer2);
        assertSend(factory, OrderReady, customer3);
    }
}
```

Fig. 1. Example textual sequence diagram describing a test case for a producer-consumer model

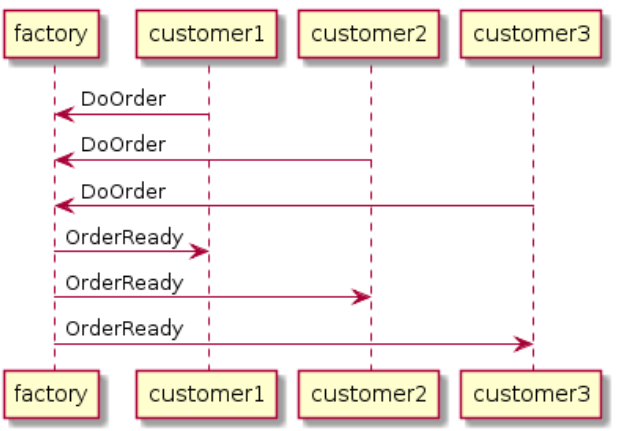

Fig. 2. Graphical sequence diagram corresponding to the textual example (generated with PlantUML [Arnaud Roques 2009])

5. EVALUATION STRATEGY

One of the problems which have to be solved to create executable sequence diagrams is synchronization. When running a sequence diagram, we have to execute the model which is under test, and the sequence diagram which tests the given model. The reason behind this is that we want to be able to check the state of model elements for state assertions and guards of the interaction operators. This can be achieved either by saving the states of the model elements after every message exchange during the model execution; or by stopping the execution of the model at certain points to run a part of the sequence diagram. The latter is much more efficient if we have a complex model and a long execution, so we chose this solution.

5.1 Synchronization

The execution of a sequence diagram must start with initializing the needed elements, as it was described in Section 3.1. After that, we have to execute the main part of the diagram, which describes the expected exchanges, until we arrive at a message exchange. If the exchange is between an actor and a model element, the message has to be sent before we can check if the exchange happened during the model execution. At this point, we have to stop the execution of the sequence diagram and start executing the model. This is because the model has to run to send messages which we can test in our sequence diagram. Here is where the most difficult question regarding synchronization arose. When do we stop the execution of the model and continue executing the sequence diagram?

We came to the conclusion that we have two options. The model execution can be stopped after a message is sent – which we call *sending semantics* – or after a message is processed – which we call *processing semantics*. Both solutions has its advantages and disadvantages. Synchronizing after message sending is probably more intuitive, but this way we can only check if the given message was sent – we do not know whether it was processed or not. This is problematic because if a message is sent during the execution, most of the time we want that message to be processed by the target. Furthermore, with this solution if we want to write a condition in the sequence diagram which checks the number of the orders in our previous example, we have to know whether the counter is increased before or after an `OrderReady` message was sent by the factory. If the order of events changes here, the sequence diagram has to be altered too.

Synchronizing after processing a signal is not so intuitive, and writing conditions can be tricky. The condition has to reflect to the state after the signal was processed. On the other hand, if the behavior of our model does not change, the sequence diagram does not need to be altered either, as opposed to the previously discussed solution. Last but not least, if we synchronize after processing a signal, the message processing can also be checked in addition to the sending, which is a huge advantage.

We concluded that stopping the model execution after processing a message has more significant advantages than synchronizing after message sending. Therefore, this is our chosen solution for the described synchronization problem.

5.2 Error counting

Counting errors during the execution is not a trivial problem either. Diagrams can be executed with lenient and strict execution mode and both of them require a different error counting logic. Lenient execution mode means that a test passes if there is a *subsequence* of the actual exchanges which matches the diagram. Additional messages can be sent, and they are not counted as an error. The test fails only if there is at least one message exchange in the diagram which did not happen during the execution, or there is a state assertion which is not correct. In strict execution mode the test passes only if the diagram describes *exactly* the same exchanges that happened during the model execution.

In lenient mode the error counting problem is simple. We have to execute the model, and when a message exchanges happens, we check the next exchange in our diagram. If the two exchanges are not equal, we keep checking the same exchange from the diagram after every message in the model execution, until we get a match. Then we can continue to execute the rest of the diagram. After the execution terminates, every expected message exchange which did not happen during the model execution is an error.

In strict execution mode counting errors is more complicated. If an actual exchange is not equal to the next exchange in the diagram, should we proceed with the sequence diagram execution – which we call *continuous semantics* –; or keep trying to match the missed message exchange from our diagram until it occurs in the model – which we call *recurrent semantics*.

First consider continuous semantics. If we encounter a mismatching pair of expected and actual message exchanges, but after that all remaining exchanges match, only one error will be shown. This is desirable. However, if we only *miss* (and not *mismatch*) a message exchange from our test case, and we proceed with the execution, every remaining exchange processed during the execution will raise a new error. Here, it would be more intuitive to report only one.

Now consider recurrent semantics. If the only problem is that actual exchanges were left out of the diagram, the number of reported errors will be the same as the number of missed messages. This is an advantage compared to the other possible solution. However, in cases where the diagram contains an exchange which does not occur in the model, we will keep trying to check the same expected exchange until the model execution terminates. With this semantics there is also another open question: should we count errors individually or group them between matching exchanges? In the first case it can happen that we have a diagram with one expected message exchange and we get for example twenty errors after the execution, which may be considered unexpected.

This problem is still under discussion, as the most suitable solution seems to be different for every diagram and model. Currently, we use recurrent semantics and we raise a new error for every mismatch. This may change in the future.

5.3 Proxy objects

Introducing the concept of proxies brings additional questions about the execution. First, in some cases during test execution it is not trivial to determine which proxy substitutes which instance. Another question is, what are the similarities and differences between concrete and proxy objects, as it would be desirable to handle them as similarly as possible. Limitations and effects should be considered as well.

As proxy objects and concrete model objects are considerably similar, let us generalize and call them lifelines. The only difference between them is that in case of model objects, it is known initially which instance is denoted, while with proxies only the type of the object is known. Therefore proxies do not specify a concrete instance when they are created.

To determine the concrete instance denoted by a proxy, the concept of binding should be introduced. Binding can only happen during message comparison, thus it should be defined how to introduce proxies into the existing comparison method of expected and actual communication. Proxies without concrete meaning may be encountered as senders or targets of expected messages. If both sender and target is an unbound proxy, furthermore types of these proxies match with the types of the actual objects, binding happens for both sender and target. If only one proxy is bound and the types are matching, one binding happens. Otherwise expected and actual exchanges are not matching. If both participants are bound proxies, the original matching method is appropriate, as they behave similarly to concrete objects. Note that message sending assertions with unbound proxy participants not only

assert communication, but a binding might also happen in the background. This concept is hidden from the sequence diagram code.

Let us consider corner cases. If a proxy is created to substitute an object that can be referenced from sequence diagram code, the proxy behaves as an alias after it is bound to the object. Two or more distinct proxy objects can be bound to the same concrete object as well. In this case, similarly to the previous one, the concrete object will be accessible via multiple different proxies.

6. PROTOTYPE IMPLEMENTATION

We created an implementation of our language in the txtUML tool. It is an open-source textual, executable and translatable UML tool, implemented in Java. We chose txtUML as the host of our prototype because it is open-source; fulfils the criteria of message driven execution; and its model executor was created in a way that only minor changes are needed to implement the synchronization of two executor threads. The latter makes the technical part of integrating the capability to execute sequence diagrams into the tool relatively easy. Furthermore, unlike in most xUML tools, users can create models in a textual representation instead of a graphical one. This fits into our concept that sequence diagrams should be defined in text because of the advantages mentioned in Section 3.2.

```
public class FactorySequenceDiagram extends SequenceDiagram {
    @Position(1) Lifeline<Factory> factory;
    @Position(2) Lifeline<Customer> customer1;
    // ...

    @Override
    public void initialize() {
        Factory f = Action.create(Factory.class, 3);
        Customer c1 = Action.create(Customer.class);
        // ...

        factory = Sequence.createLifeline(f);
        customer1 = Sequence.createLifeline(c1);
        // ...
    }

    @Override
    @ExecutionMode(ExecMode.STRICT)
    public void run() {
        assertState(factory, Factory.WaitingForOrder.class);
        fromActor(new Order(), customer1)
        assertSend(customer1, new DoOrder(), factory);
        fromActor(new Order(), customer2)
        assertSend(customer2, new DoOrder(), factory);
        fromActor(new Order(), customer3);
        // ...
    }
}
```

Fig. 3. The previous example diagram embedded in Java (excerpt)

6.1 Embedding the language in Java

Our sequence diagram language prototype is capable of executing and testing txtUML models. As both txtUML and its domain-specific modeling language are embedded in Java, we also created the sequence diagram language in this way. Users can define their own sequence diagrams by extending the provided `SequenceDiagram` class. To specify expected message exchanges, state assertions and *par* fragments, they can use the static methods of the Sequence API.

Lifelines can be declared as attributes of the diagram. Their position on the visualized diagram can be specified with the `@Position` annotation. The initialization and the execution part can be described in the `initialize` and the `run` methods of the diagram, respectively. The execution type (strict or lenient) can be defined with the `@ExecutionMode` annotation on the `run` method.

6.2 Synchronized execution

Diagrams can be executed with an executor (`SequenceDiagramExecutor`) which runs the model and the sequence diagram on two different threads in a synchronized way. The synchronization is implemented with a `BlockingQueue` which contains the current message in the sequence diagram. After the model processes a signal, we pop the current message from the queue, and the model execution waits until a message is placed into it. The queue contains a message only if the sequence diagram thread is waiting or terminated, indicating that the model executor thread can continue its execution.

6.3 Error counting

First consider the lenient execution mode. If an expected message in the diagram differs from the actual message processed in the model, we keep trying to match subsequent actual messages with the same expected message until the two are equal or the model execution stops. At the end, missed expectations are stored as errors in a list. If the execution mode is strict, an additional error is recorded each time an expected and an actual message differs. After the execution is finished, the executor provides the list with the errors which were found during testing.

7. EVALUATION

txtUML comes with demo projects. We created tests for these projects with the help of our sequence diagram implementation. One of the demos is a producer-consumer model, which is excellent for demonstrating the usage of the *par* combined fragment. Another demo project implements the model of a train with a gearbox: every time a gear is shifted, the train goes into a different state. Here we were able to write a test case with a lot of useful state assertions.

One of the demo projects contains model instances which cannot be accessed from outside the model. This project provided a good opportunity to demonstrate the usefulness of proxy objects: here we were able to create much more meaningful test cases than it would have been possible without proxies.

8. CONCLUSION

In this paper we proposed a novel approach to test xUML models with sequence diagrams. We aimed for the goal of providing a framework which offers model tests that are convenient to write, run, trace and maintain. We presented our methodology consisting of a textual test description language and an evaluation strategy that executes sequence diagrams and models under test synchronously.

Considering the possible realizations of interaction-based testing, we derived abstractions suitable for constructing meaningful test cases. We mainly built these elements upon the UML specification but we also extended it with state assertions and proxy objects. We argued why it is desirable to design the language as text-based and then intuitively defined its syntax with representative code examples.

As for test evaluation, we elaborated two alternatives depending on whether synchronization happens at sending or at processing signals. We showed that the latter has more advantages, which can be considered the informal semantics of our language. We also discussed possible ways to quantify divergence between actual and expected communication in the form of continuous and recurrent semantics. Regarding assertions concerning objects that are not directly accessible in a test case, we presented a binding strategy to realize the previously introduced concept of proxies.

Finally, we demonstrated the feasibility of our approach by elaborating details from our prototype implementation that is integrated into an xUML modeling tool. We presented how the designed language can be embedded in Java using custom classes and annotations; how alternating execution can be realized with standard synchronization features; and how we implemented the formerly discussed error counting methods.

By publishing our framework and its open-source prototype [ELTE-Soft 2019], we intend to make a useful contribution not only for the modeling community, but also for large-scale software development in general.

REFERENCES

Arnaud Roques. 2009. PlantUML. http://plantuml.com/. (2009).

Gergely Dévai, Tibor Gregorics, Boldizsár Németh, Balázs Gregorics, Dávid János Németh, Gábor Ferenc Kovács, Zoltán Gera, András Dobreff, and Máté Karácsony. 2018. Novel Architecture for Executable UML Tooling. In *Proceedings of the 11th Joint Conference on Mathematics and Computer Science (MaCS), Eger, Hungary, May 20-22, 2016.* CEUR-WS.org, online CEUR-WS.org/Vol-2046/devai-et-al.pdf, January 11, 2018.

ELTE-Soft. 2019. txtUML GitHub Repository. https://github.com/ELTE-Soft/txtUML. (2019).

Hans Grönniger, Holger Krahn, Bernhard Rumpe, Martin Schindler, and Steven Völkel. 2007. Text-based Modeling. In *Proceedings of the 4th International Workshop on Software Language Engineering (ateM 2007), Nashville, TN, USA, October 2007*.

Stephen J. Mellor and Marc Balcer. 2002. *Executable UML: A Foundation for Model-Driven Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Object Management Group. 2017. Unified Modeling Language (UML), standard, version 2.5.1. https://www.omg.org/spec/UML/2.5.1/PDF. (2017).

One Fact. 2012. xtUML – eXecutable Translatable UML with BridgePoint. https://xtuml.org/. (2012).

Mark Utting, Alexander Pretschner, and Bruno Legeard. 2012. A Taxonomy of Model-based Testing Approaches. *Softw. Test. Verif. Reliab.* 22, 5 (Aug. 2012), 297–312. DOI:http://dx.doi.org/10.1002/stvr.456