Test Case Generation by EFSM Extracted from UML Sequence Diagrams

Mauricio Rocha[1,2], Adenilso Simão[1], Thiago Sousa[2], Marcelo Batista[2]
[1]*Instituto de Ciências Matemáticas e de Computação (ICMC), USP, São Carlos, SP, Brazil*
mauriciormrocha@usp.br, adenilso@icmc.usp.br
[2]*Centro de Tecnologia e Urbanismo (CTU), UESPI, Teresina, PI, Brazil*
mauricio@ctu.uespi.br, thiago@ctu.uespi.br, marcelo.araujo@uespi.br

***Abstract*—The effectiveness of Model-Based Testing (MBT) is mainly due to the potential for automation it offers. If the model is formal and machine-readable, test cases can be derived automatically. The Extended Finite State Machine (EFSM) is a formal modeling technique widely used to represent a system. However, EFSM is not a common practice in industry. On the other hand, the Unified Modeling Language (UML) has become the de-facto standard for modeling software, but due to the lack of formal semantics, its diagrams can have ambiguous interpretations and are not suitable for testing automation. In this context, we present a systematic procedure for generating tests from a UML model. More specifically, our approach proposes a mapping from the UML Sequence Diagram into Extended Finite State Machine in order to provide a precise semantics to them and uses the ModelJUnit and JUnit libraries in order to generate test cases automatically.**

***Index Terms*—Model-Based Testing, Model-Driven Engineering, Sequence Diagram, Extended Finite State Machine, ModelJUnit, JUnit**

I. Introduction

A common practice in most software development processes is the use of abstract models to aid in the construction of products. These models represent the essential parts of a system and allow software engineers to take a conceptual view of several different software perspectives. An option for software modeling is the Unified Modeling Language (UML), since it is widely used and, due to its expressiveness, it is possible to model both static and structural aspects as well as dynamic or behavioral [1]. However, due to the lack of formal semantics, the use of UML can lead to some issues, such as inconsistency, transformation problems and different interpretations [2].

An option to minimize these problems is the use of formal models, since they have a precise semantics to accurately represent system behavior. However, what is observed in practice is that formal methods are little used in industry, probably due to the lack of training and familiarity with the mathematical notation by the developers.

In the context of software testing, modeling can increase the productivity of this activity. According to Utting et al. [3], Model-Based Testing (MBT) allows the automatic generation of tests from models and other software artifacts, making it possible to create tests for the software even before coding, thus reducing the cost of development. The central idea of MBT is generating input sequences and their expected outputs from a model or specification. The input sequences are then applied to the System Under Test (SUT) and the software outputs are compared to the outputs of the model. This implies that the model must be valid, i.e. faithfully represent the requirements. Basically, MBT are used for functional black-box testing, where software functionality is examined without any knowledge of the software's internal coding.

In MBT, it is recommended to use formal models, since they can be used as a basis for automating the testing process, making it more efficient and effective [4]. There are several formal modeling techniques based on state transition machines that can be used to specify a test model. Extended Finite State Machine (EFSM) has been widely used in the formal methods community, since they make it possible to represent the flow of control and data of complex systems. Moreover, EFSM can be implemented as a test model using the the ModelJUnit [5] library, which was designed as an extension of JUnit. Therefore, the models are written in Java, a popular programming language.

In this context, we present a systematic procedure for generation of test cases from a UML model. The idea is to use concepts of Model-Driven Engineering (MDE) to transform the UML Sequence Diagrams into EFSM and using the ModelJUnit and JUnit libraries in order to generate test cases automatically. In summary, the main contributions of this paper include:

1) Definition of transformation rules for mapping the elements of the UML Sequence Diagram into the Extended Finite State Machine constructions using Atlas Transformation Language (ATL) [6].
2) Formalization of the UML Sequence Diagram into EFSM which is a semantically accurate model.
3) Automatic source code generation of ModelJUnit and JUnit classes from EFSM using Acceleo [7].
4) Systematic procedure to generate Java tests from UML Sequence Diagram automatically.

II. Background

A. Sequence Diagram

The dynamic behavioral aspect of an object-oriented software is defined through the interaction of objects and the

DOI reference number: 10.18293/SEKE2019-133

exchange of messages among them. The main diagram of the interaction model is the UML Sequence Diagram, which presents the interactions between objects in the temporal order in which they occur.

Lifelines represent participants of the interaction that communicate via messages. These messages may correspond to the operation call, signal sending or a return message. More complex interactions can be created using combined fragment. A combined fragment is used to define control flow in the interaction. It can be composed of one or more operands, zero or more interaction constraints, and an interaction operator. An operand corresponds to a sequence of messages that are executed only under specific circumstances. Interaction constraints are also known as guard conditions and represent a conditional expression.

In this paper, we use three interaction operators that model the main procedural constructs:

- **alt**: construction of the if-then-else type. Only one operand will be executed.
- **opt**: construction of the if-then type. It is very similar to the alt operator, with the difference being that only one operand is defined, which may or may not be executed.
- **loop**: a construct that represents a loop where the single operand is executed zero or more times.

Other interaction operators defined by UML 2, that can be found in OMG (Object Management Group) [1], are not in the scope of this work.

B. Model-Driven Transformation

Model transformation is a key concept within the scope of Model-Driven Engineering (MDE). The MDE aims at supporting the development of complex software that involves different technologies and application domains, focusing on models and model transformation [8].

Similarly to models, metamodels play a key role on the MDE. A metamodel makes statements about what can be expressed in valid models of a given modeling language. Modeling languages need to have formal definitions so that transformation tools can automatically transform the models built into those languages. The OMG has created a special language called Meta Object Facility (MOF) [9], which is the default metalanguage for all modeling languages. Thus, each language is defined by means of a metamodel using the MOF.

Model transformation is the generation of a target model from a source model. This generation process consists of a set of transformation rules that describes how the elements of the source model are mapped into elements of the target model. The transformations can be performed in two ways: Model-To-Model (M2M) mapping or Model-To-Text (M2T) mapping.

C. Extended Finite State Machine

An Extended Finite State Machine (EFSM) consist of states, predicates, and assignments related to variables between transitions, so that it can represent the control and data flow of complex systems.

An EFSM can be formally represented by a 6-tuple (s_0, S, V, I, O, T) [10], where:

- S is a finite set of states with the initial state s_0;
- V is a finite set of context variables;
- I is a set of transitions entries;
- O is a set of transitions outputs;
- T is a finite set of transitions.

Each transition $tx \in T$ can also be represented formally by a tuple $tx = (s_i, s_j, P_{tx}, A_{tx}, i_{tx}, o_{tx})$, where s_i, s_j and $i_{tx} \in I$ represents the input parameters of the beginning of the state transition tx and $o_{tx} \in O$ represents the output parameter at the end of the state transition tx. In addition, P_{tx} represents the predicate conditions (guards) with their respective context variables and A_{tx} the operators (actions) with their respective current variables.

D. Model-Based Testing

The software test aims to perform an implementation of the system under construction with test data and verify that its operating behavior conforms to its specification. This implementation being tested is named the System Under Test (SUT).

In MBT, the use of models is motivated by the observation that, traditionally, the testing process is unstructured, non-reproducible, undocumented and depends on the creativity of software engineers. The idea is that artifacts used in SUT coding can help mitigate these problems [3].

In summary, the MBT covers the processes and techniques for automatic derivation of test cases from abstract software models. To achieve success in this activity, rigor is necessary in this process.

III. Our Approach

In this section we present a systematic process for test case generation by EFSM extracted from UML Sequence Diagrams. The Figure 1 illustrates our approach, which is divided into two main steps as detailed below:

Step 1 - Transformation between models. Scenarios are written in the form of the UML Sequence Diagram. This UML Sequence Diagram is transformed into an EFSM through the mapping between their respective metamodels using Atlas Transformation Language (ATL). The result of this step is a formal software model represented by an EFSM.

Step 2 - Generation of test cases. From a model of the software represented by EFSM, the test cases are generated using EFSM-based test generation methods from ModelJUnit and JUnit libraries. In this step, a Model-To-Text (M2T) transformation is performed using Acceleo, resulting in a set of test cases.

A. Metamodels

We define the UML Sequence Diagram metamodel (source) and Extended Finite State Machine metamodel (target). These metamodels were implemented in Ecore using the Eclipse Modeling Framework (EMF) [11].

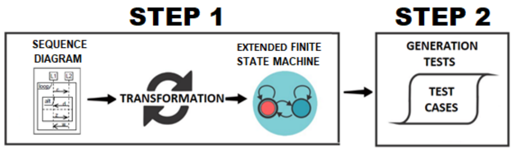


Fig. 1. Our Approach.

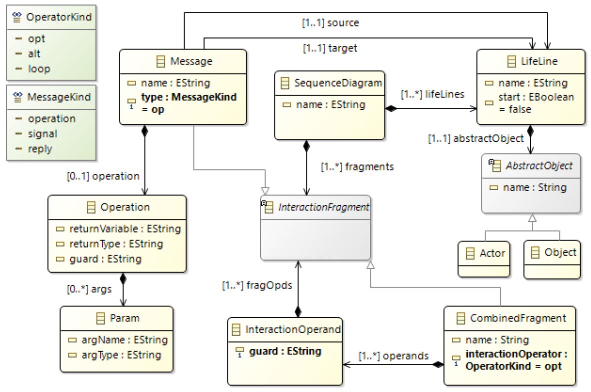


Fig. 2. Sequence Diagram Metamodel.

The complete official UML specification [1] is very complex because abstract syntax is represented in several separate diagrams, which makes it difficult to see all the connections between the important elements. In addition, the specification uses the so-called semantic variation points, meaning part of the semantics is not specified in detail to allow the use of the UML in many domains. Therefore, the official UML metamodel is heavily criticized for having many elements that are seldom used in practice [12], [13]. In this scenario, the metamodel presented in Figure 2 is simpler than the one specified by the OMG for the Sequence Diagram, and does not have constructs that are rarely used in practice. The metamodel proposed contains 13 metaclasses. The use of simplified metamodels occurs in most of the papers published in the literature [14], [15], [16].

The proposed metamodel for EFSM presented in the Figure 3 is based on the formal definition of Yang et al. [10] explained in section II.C. The metamodel is composed of six metaclasses, among which, EFSM represents an Extended Finite State Machine. The EFSM entity is composed of states, transitions and context variable.

B. Transformation Rules

In this section, we present the transformation rules between the Sequence Diagram and the Extended Finite State Machine. The following transformation rules have been defined:

- **InitFsm**: this rule creates an EFSM with the name of the sequence diagram and adds the initial state *S0*. The previous state and the current state are updated with the initial state. This rule can only be applied once.
- **Transition**: for all messages of type signal (*type* = *si*) or operation (*type* = *op*), a state is added (which is now the current state), and a transition that connects the previous state to the current state of the EFSM. The input for this transition will be labeled with the name of the message. If the message operation has a return, the output, guard, and action of this transition are labeled with the return of the operation. In addition, the event is labeled with the name of the operation, its return, and its arguments.
- **ContextVariable**: for all messages of type operation (*type* = *op*) that have a different return of *void*, a context variable is created with the name and return labeled with the name and return of the operation.
- **Alt** and **Opt**: when a fragment combined with the *alt* operator or *opt* operator is found in Sequence Diagram, it is added a new state for each operand and a new transition linking the current state to each of the created states. Every transition will have its input labeled with the message name and its output labeled with the guard of the respective operand.
- **Loop**: when a fragment combined with *loop* interaction operator is found in the Sequence Diagram, a reply message must be defined as the last message of the snippet. As soon as the process finds this message, a new state (which is now the current state) and a transition that connects the previous state to the current state in the EFSM are added. The input of this transition will be labeled with the name of the message and the output with the negation of the operator's guard. Another transition is created by connecting the previous state to the last state created before fragment. The input of this transition will be labeled with the name of the message and the output with the guard of the operator.

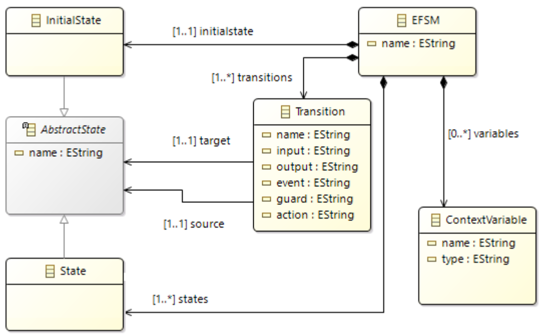


Fig. 3. Extended Finite State Machine Metamodel.

In our approach, these transformation rules were implemented using Atlas Transformation Language (ATL). ATL is one of the packages developed in the AMMA (ATLAS Model Management Architecture) model engineering platform [6]. ATL rules may be specified either in a declarative style (*Matched Rules*) or in an imperative style (*Called Rules*). *Lazy Rules* is kinds of *Matched Rules* are triggered by other rules.

In order to make feasible the transformations described above, the following *lazy rules* were implemented:

- **LrInitialState**: creates the initial state *S0*, increments the order of the states, and changes the previous state and the current state as the initial state created. In addition, the name of the Sequence Diagram being scanned is saved in a variable.

- **LrState**: creates a new state, increments the order of states, the previous state is changed to the current state and the current state changed to the new created state.
- **LrTransition**: creates a transition that connects the previous state to the current state. The transition input and event are labeled with Sequence Diagram message information. The output, guard, and action can be null and depend on the operator and message type of the Sequence Diagram.
- **LrContextVariable**: this rule creates a context variable with the name and type labeled with the return variable of the operation and the type of the operation retract, respectively.

All of these rules implemented in ATL are available in the *Transformation/SD2EFSM/SequenceDiagram2EFSM.atl* file of the approach repository [1].

C. Test Case Generation

For test case generation our approach uses the ModelJUnit and JUnit libraries, since they are open-source and their uses are very simple for Java programmers. In addition, the ModelJUnit library enables the implementation of formal models widely used in MBT, such as EFSM. Other advantages of using ModelJUnit is that it provides a variety of useful test generation algorithms, model visualization features, model coverage statistics, and other features [17].

The process of implementing the MBT environment in the ModelJUnit and Junit libraries consists of four steps:

1) **The Model**: initially, we have implemented the *FsmModel* interface to define our model in ModelJUnit. In this Java Class we define in a enumeration variable (*enum State*) all the possible states of our EFSM and for each context variable we define a variable in the class. For each input in our model, we wrote action methods (*@Action*) to define the transitions that link the states of our model. In addition to these methods, we define in our model the *getState* method that returns the current state and the *reset* method that takes the machine to the initial state.
2) **The Adapter**: in this step we implemented the Adapter class that allows our model to communicate with and take control of our SUT. For each one of the action method define in the model that trigger an event, we added a similarly named method in the adapter class. In our model defined in Step 1, we call the correct adapter method in each action method. In addition, in the Adapter class we need to instantiate an object for each class of the SUT.
3) **Generation Tests**: in this step, we initially need to instantiate the model defined in Step 1. Then, we have to choose the test strategy that will be used. ModelJUnit offers four different strategies: AllRoundTester, GreedyTester, LookaheadTester and RandomTester. In our approach we used the LookaheadTester test strategy, since it is a more sophisticated algorithm and can cover all transitions and states quickly [17]. Finally, we call the *buildGraph* method to build the graph and generate the tests. This graph will also be used to calculate coverage metrics for transitions, states, and action.
4) **Test Concretization**: In this step, the test cases were implemented in Java using the JUnit library.

These four steps described above were automatically generated by Model-To-Text (M2T) transformation using Acceleo. Acceleo is a template-based technology including authoring tools to create custom code generators. It allows you to automatically produce any kind of source code from any data source available in EMF format [7]. We have implemented the *generateClassModel*, *generateClassAdapter*, *generateClassTest* and *generateClassJUnit* generators modules. The input of these modules is the EFSM generated in step 1 of our approach.

These code generators implemented in Acceleo are available in the *Transformation/Efsm2ModelJUnit/src/Common/* folder of the approach repository 1.

IV. Example

In this section, we use an example to illustrate the application of our approach. The UML Sequence Diagram of Figure 4 presents interactions of an ATM (Automatic Teller Machine) for the withdrawal scenario.

Initially, using the Sequence Diagram editor implemented in the EMF, we created the Sequence Diagram model described in Figure 4. Then, using the transformation rules implemented in ATL, the UML Sequence Diagram is converted into an Extended Finite State Machine. At the end of the execution of the transformation rules we will have an EFSM as shown in Figure 5.

In Step 2 of our approach, from the EFSM extracted in Step 1, the test cases are generated. Using the generator modules implemented in Acceleo, the classes (*AtmModel*, *AtmAdapter*, *AtmTest* and *AtmJUnit*) are generated automatically.

The *AtmModel* class is an implementation of the *FsmModel* interface. In this class is defined the variable enumeration *State* that represents all the states (*S0*, *S1*, *S2*, *S3*, *S4*, *S5*, *S6*, *S7*, *S8*, *S9*, *S10*, *S11*, *S12* and *S13*) of our EFSM. The following *@Action* annotated methods have been implemented: *insertCard()*, *validateCard()*, *requestPassword()*, *enterPassword()*, *validatePassword*, *requestValue()*, *enterValue()*, *validateBalance()*, *value()*, *unavailableBalance()*, *exit()* and *cardOut()*. In addition, the *getState* method, the *reset* method and context variables (*cardOk*, *pswOk* and *valueOk*) were defined.

The objects of type *User*, *ATM* and *Bank* that belong to the SUT were instantiated in the *AtmAdapter* class. In this class, a method was created for each event triggered in EFSM transitions. These methods (*insertCard()*, *validateCard()*, *enterPassword()*, *validatePassword()*, *enterValue()* and

[1]https://github.com/TESTSD2EFSM/SEKE2019

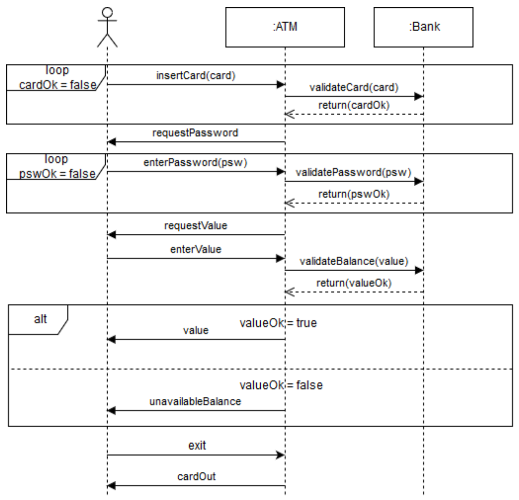


Fig. 4. ATM Sequence Diagram.

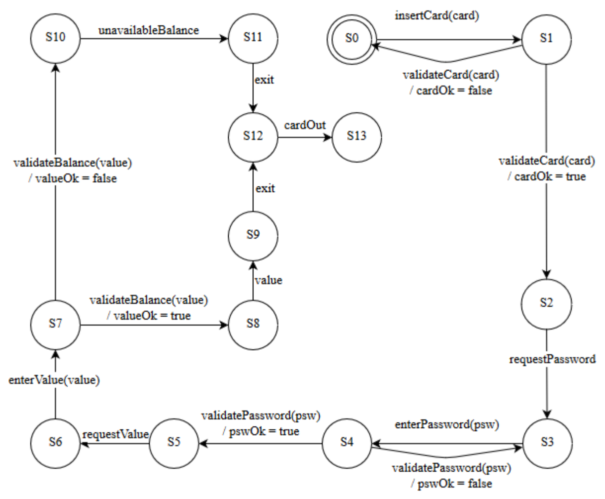


Fig. 5. ATM EFSM Model.

validateBalance()) are what make the communication of the model with the SUT.

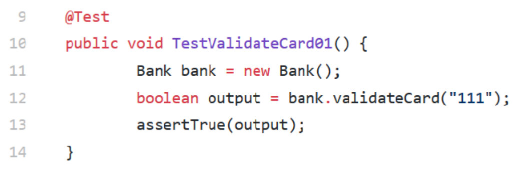
In *ATMTest* class was instantiated the objects of type *ATMModel* and we used the *LookaheadTester* test strategy. To traverse all the transitions we configure the algorithm to generate a sequence of 70 test steps. To perform the tests we set the *card* attribute equal to *111*, the *psw* attribute equal to *123* and the *balance* attribute equal to *100.00*. These attributes belong to the Bank class of the SUT.

To verify the behavior of our approach, we performed the following test cases shown on the Table I. In addition to the test data (*card*, *psw* and *value*), the Table I shows action, state and transition coverages.

In addition to the metrics presented in Table I, the test cases *TestValidateCard01*, *TestValidateCard02*, *TestValidatePassword01*, *TestValidatePassword02*, *TestValidateBalance01* and *TestValidateBalance02* were concretized in Java through the JUnit library. Figure 6 shows an example of the

TABLE I
TEST CASES GENERATED.

id	card	psw	value	Action	State	Trans.
1	222	123	50	12/12	2/2	24/24
2	222	246	200	12/12	2/2	24/24
3	111	246	50	12/12	5/5	60/60
4	111	246	200	12/12	5/5	60/60
5	111	123	50	12/12	12/12	144/144
6	111	123	200	12/12	12/12	144/144



```
@Test
public void TestValidateCard01() {
        Bank bank = new Bank();
        boolean output = bank.validateCard("111");
        assertTrue(output);
}
```

Fig. 6. Example of test case concretized in JUnit.

concrete test case of the *AtmJUnit* class.

These Java classes (*AtmModel*, *AtmAdapter*, *AtmTest* and *AtmJUnit*) are available in the *ModelJUnit/src/test/java* folder of the approach repository repository 1.

V. Related Works

One of the strengths of our approach is the automatic model transformation. As we have developed a tool to support our method, this task can be facilitated by the use of MDE concepts. Another advantage is the formulation of a UML model into formal model, since the UML has semantics problems and the formal models provide a set of techniques based on precise notation that can accurately translate the behavior of a system. In addition, since the main objective of our work is the generation of tests, our approach uses the ModelJUnit library to concretize the test cases in the Java programming language. On the other hand, we identified as a limitation of our work the use of only one UML diagram. Therefore, in this section of related works, we will compare our approach taking into consideration four aspects: used UML diagrams, tool support, use of formal models and concretization of test cases in some programming language.

In [18] is described a systematic test case generation method performed on Model-Based Testing (MBT) approaches by using UML Sequence Diagram. The UML Sequence Diagram is converted into a graph sequence and the graph is traversed to select the predicate functions. These predicates are transformed into Extended Finite State Machine (EFSM). From the EFSM, test cases are generated taking into account state coverage, transition coverage and action coverage. This technique is similar to ours, but EFSM is not automatically generated from the sequence diagram. Moreover, the technique does not use some important constructions of the Sequence Diagrams, such as the combined fragment. In this approach the test cases are concretized in the Java programming language.

In [19] an approach is presented to generate test cases using UML Activity and Sequence Diagrams. The approach consists of transforming the Sequence Diagram into a graph called *Sequence Graph* and transforming the Activity Diagram into the *Activity Graph*. The software graph is formed by integrating the two graphs that are traversed to generate the test suite. The proposal uses UML models for generating tests, but differs from ours since it does not use MDE concepts and does not use formal models for test generation. In addition, the approach does not concretize test cases in some programming language.

In [20] an approach is presented to generate test cases using UML Sequence Diagrams. The approach consists of transforming Sequence Diagram in to Sequence Diagram Graph (SDG) and generate test cases from SDG. The Sequence Diagram is built with Object Constraint Language (OCL) and the SDG defines the activities as nodes and the interactions in the form of paths. The test case is generated by visiting the nodes and edges in the SDG. This proposal uses UML models to generate tests, but differs from ours since it does not use MDE concepts and formal models. In addition, test cases are not implemented in any programming language.

In the work of Seo et al. [16] is presented a method for generating test cases from Sequence Diagrams. This method suggests to generate test cases after conducting an intermediate transformation from a Sequence Diagram to an Activity Diagram. The proposal is similar to ours, since it uses model transformation, but does not use a formal model for generating test cases. Also we can not identify in the work if the transformation of models is carried out using MDE concepts, because it does not describe the manipulated metamodels in the process. In addition, the approach does not concretize test cases in some programming language.

VI. Conclusions and Future Work

In this paper, we present a systematic procedure to generate test cases from UML Sequence Diagrams. Our approach uses concepts of Model-Driven Engineering to formalize UML Sequence Diagrams into EFSM and uses the ModelJUnit and JUnit libraries for automatic generation of test cases.

In Step 1, for the transformation of UML Sequence Diagram to EFSM, we perform the mapping of the elements of the respective metamodels through transformation rules. With this, we can provide a precise semantics to a widely used UML model.

In Step 2 of the approach, the formal model can be used as basis for automating the testing process, making it more efficient and effective. We used the ModelJUnit library to provide an interface to implement a formal test model, an adapter that communicates our model with the SUT and some test strategies already implemented. In addition, the execution of the tests is measured by coverage of state, actions and transitions. We use the JUnit library to perform tests in the Java programming language.

From the example, we can observe the applicability of our proposal, mainly in the generation of functional tests, since the approach starts with UML Sequence Diagrams that are important tools to model software scenarios and we end with test cases materialized in the Java programming language. These tests were performed and metrics were generated allowing to analyze the behavior of the SUT according to the test model created.

As future work, other UML diagrams can be incorporated into the systematic procedure of generating tests and applying it to real examples through case studies or controlled experiments. In addition, the generated EFSM can be used for formal verification, such as checking safety, liveness and fairness properties.

References

[1] O. M. G. OMG. (2015) Unified modeling language 2.5. [Online]. Available: http://www.omg.org/spec/UML/2.5/

[2] M. Petre, "Uml in practice," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 722–731.

[3] M. Utting, A. Pretschner, and B. Legeard, "A taxonomy of model-based testing approaches," *Softw. Test. Verif. Reliab.*, vol. 22, no. 5, pp. 297–312, 2012.

[4] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan, "Using formal specifications to support testing," *ACM Comput. Surv.*, vol. 41, no. 2, pp. 9:1–9:76, Feb. 2009.

[5] ModelJUnit. (2010) The model-based testing tool. [Online]. Available: https://sourceforge.net/projects/modeljunit/

[6] J. Bézivin, F. Jouault, and D. Touzet, "An introduction to the atlas model management architecture," 03 2005.

[7] E. M. Framework. (2018) Acceleo. [Online]. Available: https://www.eclipse.org/acceleo/

[8] S. Kent, "Model driven engineering," in *International Conference on Integrated Formal Methods*. Springer, 2002, pp. 286–298.

[9] O. M. G. OMG. (2016) Mof - meta object facility. [Online]. Available: http://www.omg.org/spec/MOF/

[10] R. Yang, Z. Chen, Z. Zhang, and B. Xu, "Efsm-based test case generation: Sequence, data, and oracle," *International Journal of Software Engineering and Knowledge Engineering*, vol. 25, no. 04, pp. 633–667, 2015.

[11] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *EMF: Eclipse Modeling Framework 2.0*, 2nd ed. Addison-Wesley Professional, 2009.

[12] S. Sen, N. Moha, B. Baudry, and J.-M. Jézéquel, "Meta-model pruning," in *Model Driven Engineering Languages and Systems*, A. Schürr and B. Selic, Eds. Springer Berlin Heidelberg, 2009, pp. 32–46.

[13] F. Fondement, P.-A. Muller, L. Thiry, B. Wittmann, and G. Forestier, "Big metamodels are evil," in *Model-Driven Engineering Languages and Systems*, A. Moreira, B. Schätz, J. Gray, A. Vallecillo, and P. Clarke, Eds. Springer Berlin Heidelberg, 2013, pp. 138–153.

[14] R. Grønmo and B. Møller-Pedersen, "From sequence diagrams to state machines by graph transformation," in *Theory and Practice of Model Transformations*, L. Tratt and M. Gogolla, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 93–107.

[15] Z. Micskei and H. Waeselynck, "The many meanings of uml 2 sequence diagrams: A survey," vol. 10, pp. 489–514, 10 2010.

[16] Y. Seo, E. Y. Cheon, J. A. Kim, and H. S. Kim, "Techniques to generate utp-based test cases from sequence diagrams using m2m (model-to-model) transformation," in *IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*, June 2016, pp. 1–6.

[17] M. Utting, "How to design extended finite state machine test models in java," in *Model-Based Testing for Embedded Systems*, J. Zander, I. Schieferdecker, and P. J. Mosterman, Eds. Boca Raton, FL: CRC Press/Taylor and Francis Group, 2012, pp. 147–170.

[18] V. Panthi and D. P. Mohapatra, "Automatic test case generation using sequence diagram," in *Proceedings of International Conference on Advances in Computing*, A. Kumar M., S. R., and T. V. S. Kumar, Eds. New Delhi: Springer India, 2012, pp. 277–284.

[19] A. Tripathy and A. Mitra, "Test case generation using activity diagram and sequence diagram," in *Proceedings of International Conference on Advances in Computing*, A. Kumar M., S. R., and T. V. S. Kumar, Eds. New Delhi: Springer India, 2013, pp. 121–129.

[20] M. MD* and B. GB, "A new approach to derive test cases from sequence diagram," *Information Technology & Software Engineering*, 2014.