Noname manuscript No.
(will be inserted by the editor)

Model-Based Test Case Generation from UML Sequence Diagrams using Extended Finite State Machines

Mauricio Rocha · Adenilso Simão · Thiago Sousa

Received: date / Accepted: date

Abstract The effectiveness of model-based testing (MBT) is mainly due to its potential for automation. If the model is formal and machine-readable, test cases can be derived automatically. One of the most used formal modeling techniques is the interpretation of a system as an extended finite state machine (EFSM). However, formal models are not a common practice in the industry. The Unified Modeling Language (UML) has become the de-facto standard for software modeling. Nevertheless, due to the lack of formal semantics, its diagrams can be given ambiguous interpretations and are not suitable for testing automation. This article introduces a systematic procedure for the generation of tests from UML models that uses concepts of model-driven engineering (MDE) for formalizing UML sequence diagrams into extended finite state machines and providing a precise semantics for them. It also applies ModelJUnit and JUnit libraries for an automatic generation of test cases. A case study was conducted in a real software towards the evaluation of its applicability.

Keywords Model-based testing · Model-driven engineering · Sequence diagram · Extended finite state machine · ModelJUnit · JUnit

✉ Mauricio Rocha
Instituto de Ciências Matemáticas e de Computação (ICMC), USP, São Carlos, SP, Brazil
E-mail: mauriciormrocha@usp.br

Adenilso Simão
Instituto de Ciências Matemáticas e de Computação (ICMC), USP, São Carlos, SP, Brazil
E-mail: adenilso@icmc.usp.br

Thiago Sousa
Centro de Tecnologia e Urbanismo (CTU), UESPI, Teresina, PI, Brazil
E-mail: thiago@ctu.uespi.br

1 Introduction

Software engineering (SE) aims to discipline software development in order to make it economically viable. According to [18], its main goal is to provide methods, tools, and procedures that enable the management of the software development process and provide a basis for the construction of high-quality software with high productivity.

SE offers several ways of developing quality software, e.g., software testing and formal modeling. The former is considered a critical element of software quality assurance and represents the latest revision of specification, design, and coding [18]. On the other hand, formal modeling is used in the early stages of the development process to avoid ambiguities in specifications and minimizing failures [25]. Both activities have advantages and limitations, and a high degree of complementarity, which makes their combined use improve the quality of the software under construction.

A common practice in most software development processes is the use of abstract models, which represent the essential parts of a system and enable software engineers to take a conceptual view of several different software perspectives. The Unified Modeling Language (UML), a widely used alternative, enables the modeling of both static and structural aspects, as well as dynamic or behavioral ones [14]. According to [17], the main problems of using UML are inconsistencies, transformation problems, and different interpretations, which are caused by the lack of a formal semantics.

Modeling can increase the productivity of software testing. According to [29], model-based testing (MBT) promotes the automatic generation of tests from models and other software artifacts, making it possible to create tests for the software prior to coding, thus reducing the development costs. Its central idea is to generate input sequences and their expected outputs from a model or specification. The input sequences are then applied to the software under testing and the software outputs are compared to the outputs of the model. This implies the model must be valid, i.e., that it accurately represents the requirements.

Using formal models is recommended in MBT, since they can be used as a basis for the automation of the testing process, thus increasing its efficiency and effectiveness [9]. Another advantage is that such models reduce ambiguities generated by natural languages. The various formal modeling techniques based on state transition machines can specify a test model, and differ from each other according to the characteristics of an explicit or implicit representation of certain elements [24].

Extended finite state machines (EFSMs) have been widely used by the formal methods community, since it enables the representation of the flow of control and data of complex systems. Moreover, it can be implemented as a test model by using the ModelJUnit [12] library, designed as an extension of JUnit and written in Java, which is a popular programming language.

Although the UML is the most popular modeling language, its diagrams can generate inconsistencies and different interpretations due to the lack of

formal semantics. Using formal models can minimize such problems, since they have a precise semantics that accurately represents a system's behavior, and the test generation methods from formal models, such as EFSMs, can be taken as a reference, since they are well-established methods. However, in practice, formal models are rarely used in the industry, probably due to the lack of training and familiarity with the mathematical notation by developers.

In this context, we have developed a systematic procedure for the generation of test cases from a UML model [19]. The idea is to use concepts of model-driven engineering (MDE) to transform UML sequence diagrams into EFSMs and automatically generate test cases using the libraries ModelJUnit and JUnit. The main contributions of our previous study include:

1. A definition of transformation rules for the mapping of elements of the UML sequence diagram into extended finite state machine constructions using the Atlas Transformation Language (ATL) [1].
2. The formalization of the UML sequence diagram in terms of EFSMs, which are semantically accurate models.
3. An automatic source code generation of ModelJUnit and JUnit classes from EFSMs using Acceleo [5].
4. A systematic procedure that generates Java tests from UML sequence diagrams automatically.

However, the approach described in [19] imposes the following limitations: (I) nested combined fragments cannot be used in sequence diagrams; (II) the source code that simulates the system under test (SUT), called stubs, is generated manually; and (III) only one example of automatic teller machine (ATM) was considered to illustrate the applicability of the approach.

Towards overcoming the limitations, we have extended the systematic procedure to generate tests from UML sequence diagrams. Therefore, the contributions of this study include:

1. A modification in the transformation rules to allow nested combined fragments with five levels of depth.
2. An automatic source code generation of stub classes from UML sequence diagrams using Acceleo.
3. The application of the test generation procedure in a case study using real software models.

The remainder of this paper is structured as follows: Section 2 addresses preliminary definitions of UML sequence diagrams, model-driven transformation, extended finite state machines, and model-based testing; Section 3 describes the systematic procedure for the test cases generation, the metamodels used, the transformation rules from UML sequence diagrams for EFSMs, stubs generation, and the ModelJUnit and JUnit libraries; Section 4 reports a case study performed on a real-world software system; Section 5 addresses some related work; finally, Section 6 provides the conclusions and suggests some further work.

2 Background

This section introduces the basic concepts discussed, namely, UML sequence diagram, model-driven transformation, extended finite state machine, and model-based testing.

2.1 Sequence Diagram

The dynamic behavioral aspect of an object-oriented software is defined through the interaction of objects and exchange of messages among them. The main diagram of the interaction model is the UML sequence diagram, which shows the interactions between objects in the temporal order of their occurrence.

Basic interaction and combined fragments are the two types of elements that represent the main interactions and their notations in the UML sequence diagram [11].

Fig. 1 shows the elements of a basic interaction. Lifelines represent participants of the interaction that communicate via messages: such messages may correspond to an operation call, signal sending, or a return message. Execution specification is a unit of behavior or action within a lifeline and represents the time at which an object is active, i.e., the time at which it performs some operation. Sending and receiving messages are marked with the specification of a message occurrence.

More complex interactions can be created by combined fragments, which define control flow in the interaction and comprise one or more operands, zero or more interaction constraints, and an interaction operator. An operand cor-

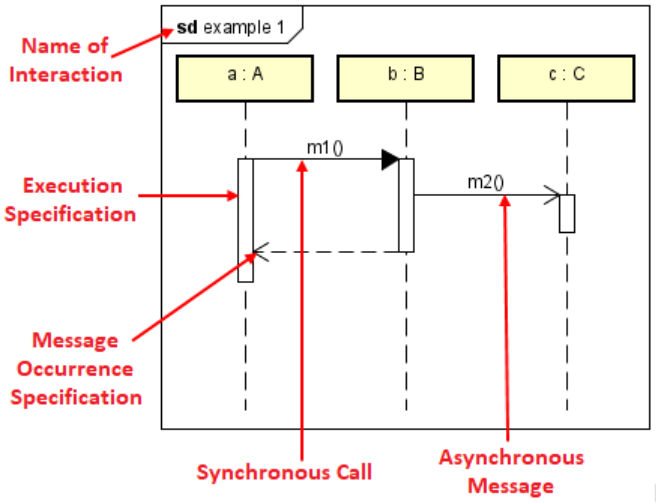


Fig. 1 Elements of a basic interaction. [11]

responds to a sequence of messages executed only under specific circumstances. Interaction constraints are also known as guard conditions and represent a conditional expression. Fig. 2 illustrates the main elements of this construction.

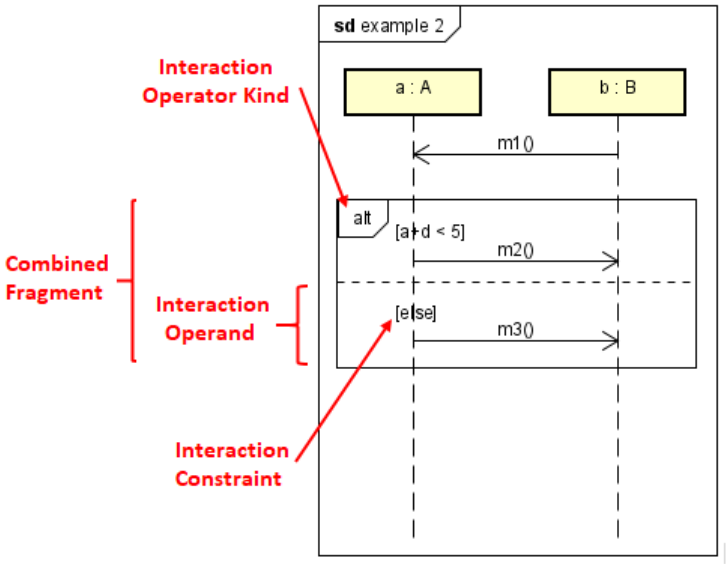


Fig. 2 Interactions with a combined fragment. [11]

We used the following three interaction operators to model the main procedural constructs:

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

Other interaction operators defined by UML 2, such as *seq*, *break*, *par*, *strict*, *critical*, *neg*, *assert*, *ignore* and *consider* and that can be found in OMG [1] (Object Management Group) [14] are not in the scope of this research.

2.2 Model-Driven Transformation

Model transformation is a key concept within the scope of model-driven engineering (MDE). MDE aims at supporting the development of complex software

[1] International consortium of companies that define and ratify standards in the area of object orientation.

that involves different technologies and application domains, focusing on models and model transformation [10,6,21].

Similarly to models, metamodels play a key role in MDE. A metamodel produces statements on what can be expressed in valid models of a given modeling language. Modeling languages must offer formal definitions, so that transformation tools can automatically transform the models built into those languages. OMG has created a special language, called Meta Object Facility (MOF) [15], which is the default metalanguage for all modeling languages. Therefore, each language is defined by a metamodel using the MOF.

Model transformation is the generation of a target model from a source model [20]. The process consists of a set of transformation rules that describes the way the elements of the source model are mapped into elements of the target model. The transformations can occur in two ways, i.e., Model-To-Model (M2M) mapping or Model-To-Text (M2T) mapping.

Fig. 3 shows a simple Model-To-Model transformation scheme. In this case, both models conform to their respective metamodels. The transformation rules are defined from mapping the elements of the source metamodel to the elements of the target metamodel. A transformation is performed in concrete models.

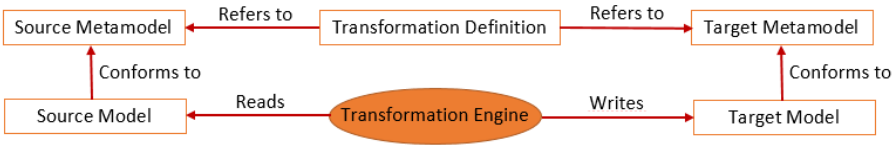


Fig. 3 A simple Model-To-Model transformation scheme. [4]

In the Model-To-Text transformation, a source code is generated from a model. This model transformation can use a template-based technology. In this context, a template consists of the target text containing metacode to access variable information [4]. Fig. 4 shows a simple Model-To-Text transformation scheme.

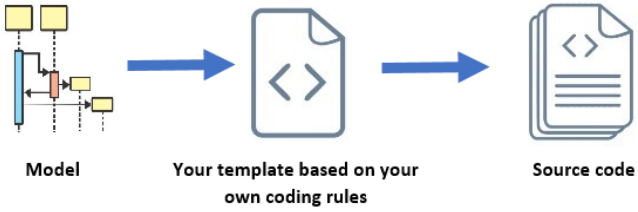


Fig. 4 A simple Model-To-Text transformation scheme. [5]

2.3 Extended Finite State Machine

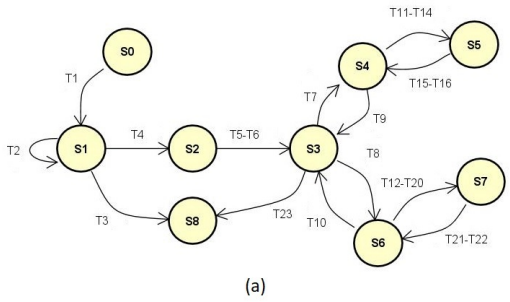
An extended finite state machine (EFSM) consists of states, predicates, and assignments related to variables between transitions, so that it can represent the control and data flow of complex systems.

An EFSM can be formally represented by a 6-tuple (s_0, S, V, I, O, T) [30], where

- S is a finite set of states with initial state s_0;
- V is a finite set of context variables;
- I is a set of transition inputs;
- O is a set of transition outputs; and
- 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 are the origin and target states of transition tx, and $i_{tx} \in I$ represents the input parameters of the beginning of the transition tx, such as events that can be interpreted as special types of input parameters, and $o_{tx} \in O$ denotes the output results at the end of the transition tx. P_{tx} represents the predicate conditions (guards) with their respective context variables and A_{tx} denotes the operations (actions) with their respective current variables.

According to [30], EFSM models can be represented as a directed graph $G(V, E)$. The elements of V represent the states of an EFSM and E denotes its transitions, as shown in Fig. 5. Fig. 5 (a) displays an EFSM model of an



(a)

Transition Name	State Transition	Event	Guard	Action
T1	S0 → S1	card(pin,sb,cb)		attemps=0
T2	S1 → S1	pim(p)	p!=pin && attemps<3	
T3	S1 → S8	pim(p)	p!=pin && attemps==3	
T4	S1 → S2	pim(p)	p==pin	
T5	S2 → S3	english()		l=e
T6	S2 → S3	chinese()		
T7	S3 → S4	current()		
T8	S3 → S6	saving()		
T9	S4 → S3	done()		
T10	S6 → S3	done()		
T11	S4 → S5	balance(l)	l==c	print_c(cb)
T12	S4 → S5	balance(l)	l==c	print_e(cb)
T13	S4 → S5	deposit(d)		cb=cb+d
T14	S4 → S5	withdraw(w)	w>0 && w<cb	cb=cb-w
T15	S5 → S4	display(e)	l==e	receipt(e)
T16	S5 → S4	display(c)	l==c	receipt(c)
T17	S6 → S7	withdraw(w)	w>0 && w<sb	sb=sb-w
T18	S6 → S7	deposit(d)	d>0	sb=sb+d
T19	S6 → S7	balance(l)	l==e	print_e(sb)
T20	S6 → S7	balance(l)	l==c	print_c(sb)
T21	S7 → S6	receipt(l)	l==e	receipt(e)
T22	S7 → S6	receipt(l)	l==c	receipt(c)
T23	S3 → S8	exit()		

(b)

Fig. 5 (a) An EFSM representing an automated teller machine. (b) Detailed information on EFSM transitions. [30]

automated teller machine (ATM) with its states and transitions, and Fig. 5 (b) shows the details of the ATM transitions with their events, guards and actions.

2.4 Model-Based Testing

A software test executes a system under construction with test data and checks if its operating behavior conforms to its specification. This implementation under test is called the System Under Test (SUT).

In MBT, test cases are generated from models, templates, or a combination of models and templates and then executed in the SUT. Using models is motivated by the observation that the testing process is traditionally unstructured, non-reproducible, and undocumented, and depends on the creativity of software engineers. The idea is that artifacts used in the SUT coding can help mitigate such problems [29].

In summary, MBT covers the processes and techniques for the automatic derivation of test cases from abstract software models. Since rigor is required, [29] defined a generic MBT process divided into the following 5 steps (Fig. 6):

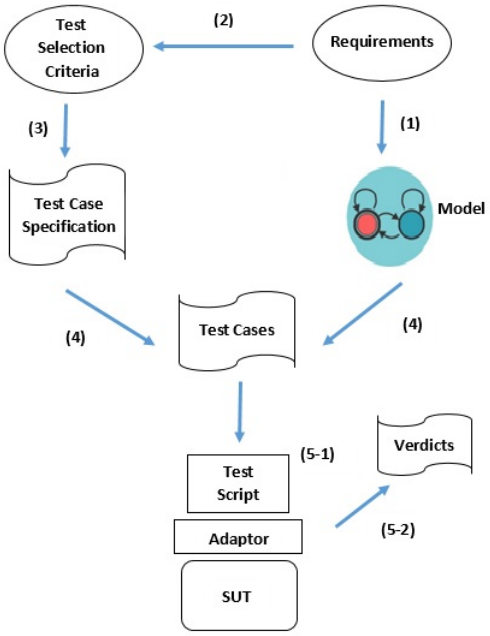


Fig. 6 Generic process for model-based testing. [29]

– **Step 1** - Test model: a model of the SUT is created from the software requirements. It is called a test model, because its elements are in accordance with the objective of the test. The generation of relevant test cases depends directly on the test model generated, therefore the test model must be accurate to represent the system's requirements;

- **Step 2** - Selection criteria: test selection criteria are chosen and established for guiding the automatic generation of test cases from the model created in the previous step;
- **Step 3** - Test case specifications: the test selection criteria are transformed into test case specifications by formalizing the notation of the criteria, making them operational. For example, the state coverage of an EFSM can be transformed into a set of test case specifications;
- **Step 4** - Generation of test cases: after the test model and test case specifications have been defined, a set of test cases is generated to meet the specifications; and
- **Step 5** - Execution of the test cases: the test cases are run, manually or automatically, and yield the result of the execution.

An important point in the MBT process is the choice of the model representation format. A widely used format is EFSMs, as described in Section 2.3.

3 Systematic Procedure

This section describes a systematic procedure for test case generation by EFSMs extracted from UML sequence diagrams. The procedure starts with the construction of a sequence diagram and finishes with the generation of test cases for the scenarios described in the diagram, which are executed in the stubs.

Fig. 7 illustrates the procedure, which is divided into four main steps, as detailed below:

(a) **Transformation between models**. Scenarios are written as a UML sequence diagram, which is transformed into an EFSM through the mapping between their respective metamodels using Atlas Transformation Language (ATL). The result is a formal software model represented by an EFSM.
(b) **Stubs Generation**. The stubs source code is generated from the sequence diagram. Thus, a Model-To-Text (M2T) transformation is performed by Acceleo, resulting in the stub classes.
(c) **Test Case Generation**. Test cases are generated by EFSM-based test generation methods and ModelJUnit / JUnit libraries from a model of the software represented by EFSM. A Model-To-Text (M2T) transformation is performed by Acceleo, thus resulting in a set of test cases.
(d) **Test Case Execution**. The abstract tests generated by ModelJUnit library are executed in the stubs and action, state and transition coverage metrics are automatically generated. After the execution, the concrete tests generated in the JUnit library are executed in the stubs and the verdict is built.

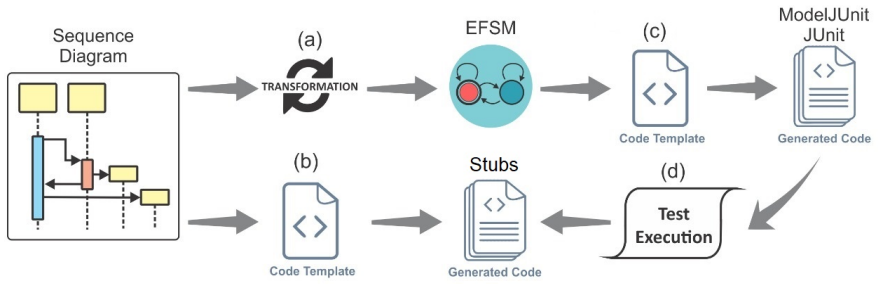


Fig. 7 Systematic procedure for test case generation.

For all steps of the procedure, a project has been created and the source code of the prototype is available at the repository[1]. In the repository, there is a *README* file explaining how to perform each step of the procedure.

3.1 Metamodels

This section focuses on defining the UML sequence diagram metamodel (source) and extended finite state machine metamodel (target), which were implemented in Ecore by the Eclipse Modeling Framework (EMF) [26].

The complete official UML specification [14] is highly complex, since the abstract syntax is represented in several separate diagrams, which hampers the visualization of all connections among the important elements. Moreover, the specification uses the so-called semantic variation points, i.e., part of the semantics is not fully specified for enabling the use of UML in several domains. Therefore, the official UML metamodel has been heavily criticized, since many of its elements are rarely used in practice [22,7]. The metamodel shown in Fig. 8 is simpler than the one specified by OMG for the sequence diagram, and does not have constructs which are rarely used in practice. The application of simplified metamodels has been widely reported in the literature [8,11,23]. The metamodel contains 11 metaclasses, among whose sequence diagram represents a UML sequence diagram model, and provides several life lines (LifeLine) and fragments (InteractionFragment). A lifeline is represented by an abstract object (AbstractObject), which can be an Actor or an Object, and has a *start* attribute, which indicates the lifeline initiates the process. A Message or a Combined Fragment can be represented from the Interaction Fragment abstract metaclass. The Combined Fragment metaclass has an attribute (InteractionOperator) that defines its type (e.g., *alt*, *opt*, and *loop*). A Combined Fragment is composed of one or more operands, represented by the InteractionOperand class. Each operand has a guard condition, and can comprehend fragments represented by messages or nested combined fragments.

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

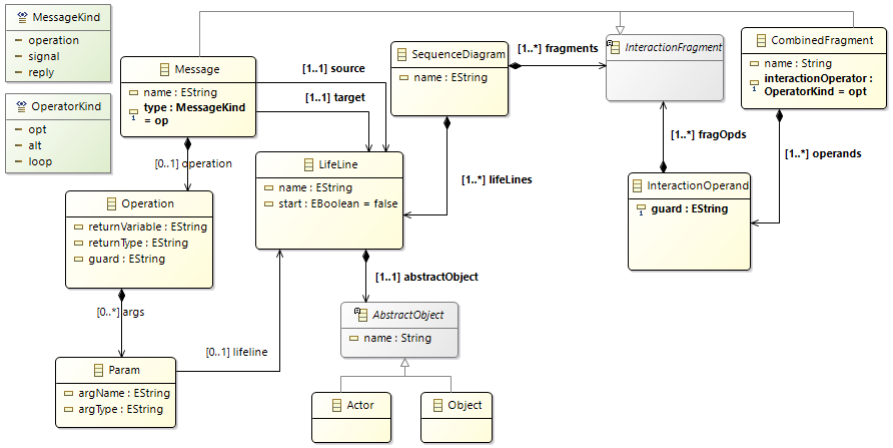


Fig. 8 Sequence diagram metamodel.

A message is triggered between lifelines and can be of three types, namely operation, signal, and return.

The metamodel proposed for EFSMs (Fig. 9) is based on the formal definition of Yang et al. [30], explained in Section 2.3. It consists of ten metaclasses, and the EFSM entity is comprised of states, transitions and context variables. Transitions are composed of inputs, outputs, guards, and actions. Events are a special type of input parameters for transitions, which may or may not be triggered by these transitions.

Such metamodels implemented in Ecore are available in the *SEQUENCE-DIAGRAM/model* and *EFSM/model* folders of the repository1.

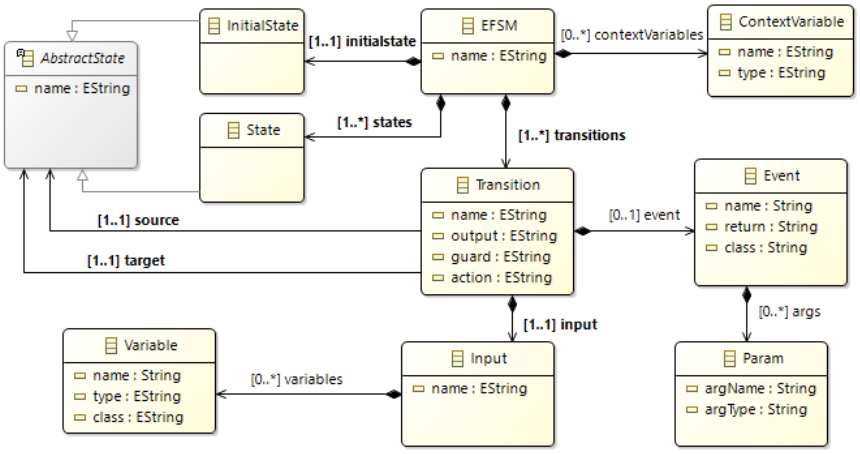


Fig. 9 Extended finite state machine metamodel.

3.2 Transformation Rules

After defining the metamodel that represents the language of the sequence diagram to which formal semantics are assigned, as well as the EFSM metamodel, the next step is to define the transformation rules that map the elements of such metamodels. The rules will have a sequence diagram model as input and an EFSM model as output.

We have implemented these transformation rules in Atlas Transformation Language (ATL), one of the packages developed in the AMMA (ATLAS Model Management Architecture) model engineering platform [1]. ATL rules may be specified in a declarative (*Matched Rules*) or imperative (*Called Rules*) style. *Lazy Rules* are types of *Matched Rules* triggered by other rules.

Before we specifically address the transformation rules, it is important to note that these rules iteratively add new states to the EFSM and connect them to previously added states. For this purpose, the rules use three variables defined in *SequenceDiagram2EFSM* module: the *order* variable representing the *state order*, *preState* describing the *previous state*, and *curState* expressing the *current state*.

The following is a description of the transformation rules defined:

- **InitEfsm**: An instance of a *SequenceDiagram* metaclass is mapped directly to an instance of the *EFSM* metaclass and is given the same name. Moreover, when a lifeline with the *Start* attribute equal to *true* is found, the initial state *S0* of the EFSM is created. This rule can be applied only once. Fig. 10 shows an example of the transformation rule.
- **Transition**: for all instances of *Message* of *signal* (*type* = *si*) or *operation* (*type* = *op*) type, a state is created and added to the EFSM, a transition that connects the previous state to the state added in EFSM is created, and an *Input* instance is created at the EFSM transition, and labeled with the name of the message. If the message type is *operation*, instances of input variables (*Variable*) are created with the name, type of the operation argument and class equal to the lifeline class that sent the message. If there is a return in an operation, the output, guard, and action of this transition are labeled with the return of the operation. The event is labeled with the name of the operation, its return, arguments and class equal to the

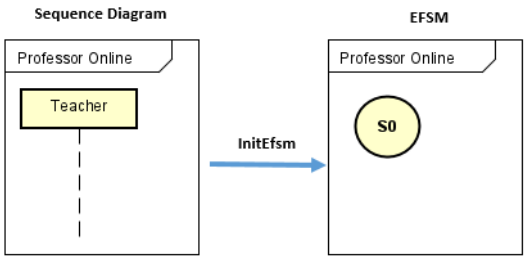


Fig. 10 InitEfsm rule.

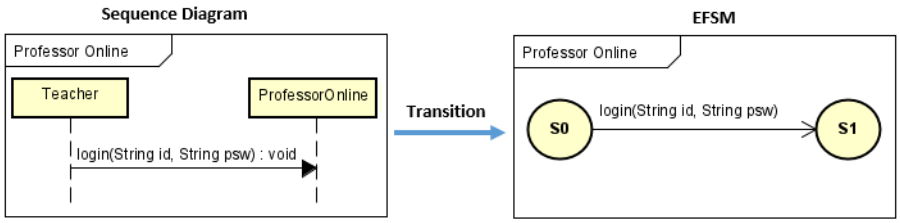


Fig. 11 Transition rule with operation without return.

lifeline that receives the message. Fig. 11 and Fig. 12 show examples of this transformation rule with an operation without a return and an operation with a return, respectively.

– **ContextVariable**: for all instances of *Message* of *operation* (*type* = *op*) type with a return other than *void*, a context variable is created in the EFSM with the same name and return of the message operation. In Fig. 12, the *userOk* context variable with *boolean* type was created in the EFSM.

– **Alt** and **Opt**: for all instances of *CombinedFragment* with *alt* operator or *opt* operator, a new state and a new transition for each operand are created in the EFSM. The new transitions link the current state with the new states. The input of the transitions will be labeled with the message name,

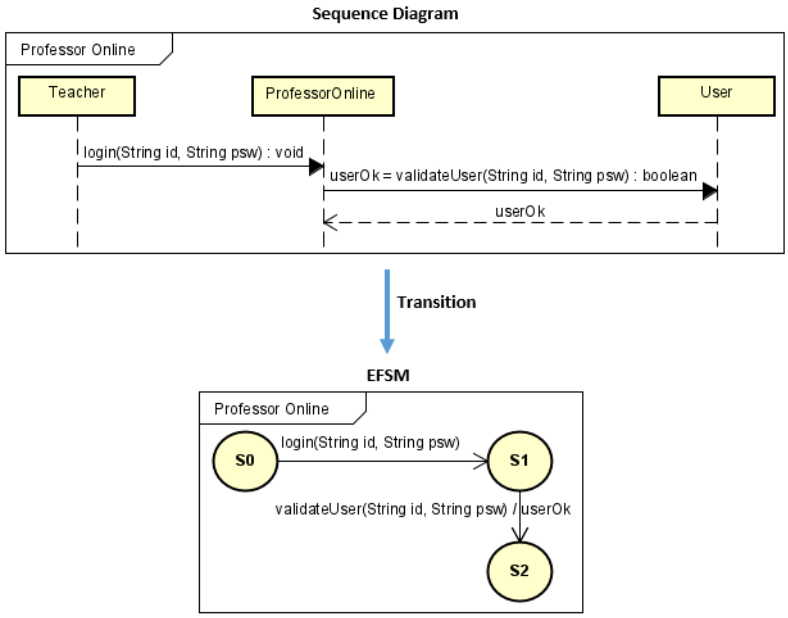


Fig. 12 Transition rule with return operation.

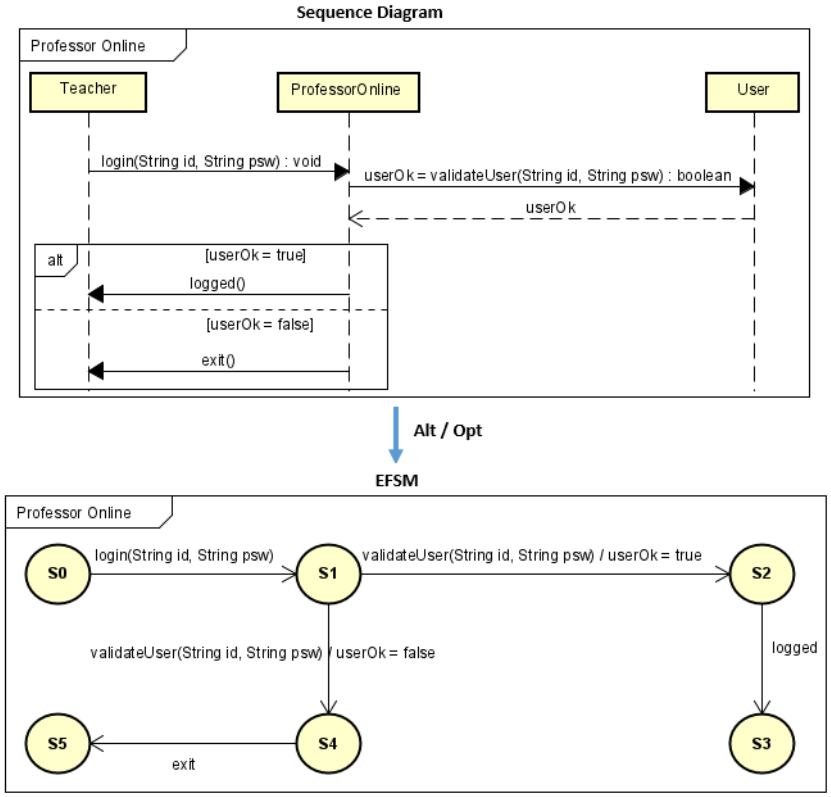


Fig. 13 Alt/Opt rule.

and input variables are created with the name and type of the operation argument and class equal to the lifeline class that sent the message. The outputs, guard and action of the transitions are labeled with the guard of the respective operand. The event is labeled with the name of the operation, its return, arguments and class equal to the lifeline that receives the message. Fig. 13 shows an example of this transformation rule.

– **Loop**: for all instances of *CombinedFragment* with *loop* interaction operator, a reply message must be defined as the last message of the operand. When the process finds an instance of this reply message, a new state and a new transition that connect the current state to the added state are created in the EFSM. The input of the 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 prior to the the 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. 14 shows an example of this transformation rule.

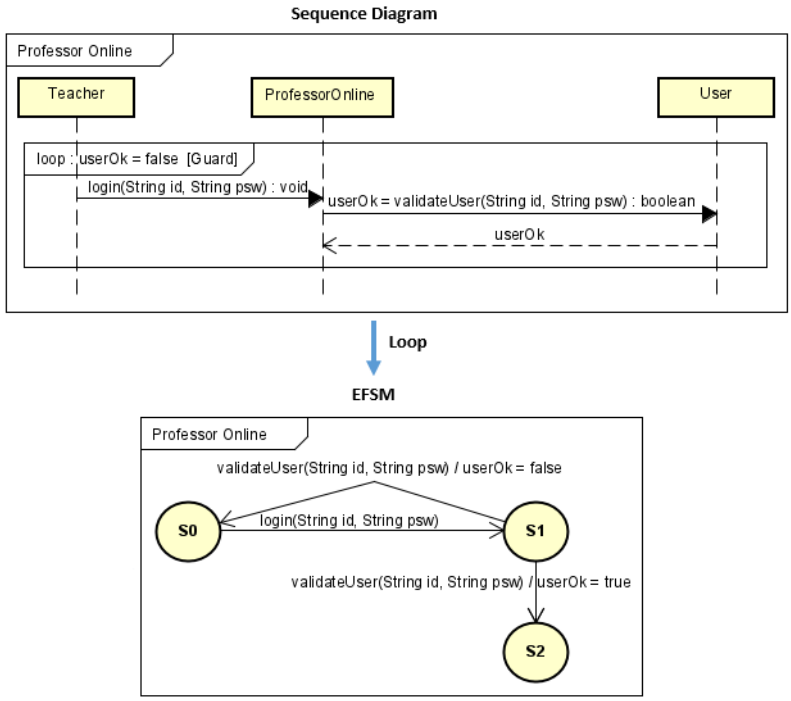


Fig. 14 Loop rule.

The following *Lazy Rules* were implemented for the feasibility of the transformations:

– **LrInitialState**: it creates initial state *S0*, increments the order of the states (*order* variable), and changes the previous (*preState* variable) and current (*curState* variable) states as the initial state created. The *order*, *preState*, and *curState* variables are defined in the *SequenceDiagram2EFSM* module. Below is the ATL code for this *Lazy Rule*.

```
lazy rule LrInitialState {
  from
    l : SequenceDiagram!LifeLine
  to
    i : EFSM!InitialState(name <- 'S0')
  do {
    thisModule.order <- thisModule.order + 1;
    thisModule.preState <- i;
    thisModule.curState <- i;
  }
}
```

– **LrState**: it creates a new state, increments the order of states (*order* variable), the previous state (*preState* variable) is changed to the current state (*curState* variable) and the current state is changed to the newly created

state. The *order*, *preState*, and *curState* variables are defined in the *SequenceDiagram2EFSM* module. Below is the ATL code for this *Lazy Rule*.

```
lazy rule LrState {
  from
    m : SequenceDiagram!Message
  to
    i : EFSM!State(
        name <- 'S'+thisModule.order.toString()
    )
  do {
    thisModule.order <- thisModule.order + 1;
    thisModule.preState <- thisModule.curState;
    thisModule.curState <- i;
  }
}
```

– **LrTransition**: it creates a transition that connects the source state (*source* variable) to the target state (*target* variable). The transition output, transition guard, and transition action can be null and depend on the operator and message type of the sequence diagram. The transition is labeled with a concatenation of source state, symbol "→" and target state. The *source*, *target*, *output*, *guard*, and *action* variables are defined in the *SequenceDiagram2EFSM* module. Below is the ATL code for this *Lazy Rule*.

```
lazy rule LrTransition {
  from
    m : SequenceDiagram!Message
  to
    t : EFSM!Transition(
        output <- thisModule.output,
        source <- thisModule.source,
        target <- thisModule.target,
        name <- thisModule.source.name+'->'
        +thisModule.target.name,
        guard <- thisModule.guard,
        action <- thisModule.action
    )
}
```

– **LrTransitionInput**: it creates transition inputs labeled with the name of the messages. The *inputName* variable is defined in the *SequenceDiagram2EFSM* module. Below is the ATL code for this *Lazy Rule*.

```
lazy rule LrTransitionInput {
  from
    t : EFSM!Transition
  to
    i : EFSM!Input(
        name <- thisModule.inputName
    )
}
```

– **LrTransitionInputVar**: it creates input variables with the name, type, and class of message operation arguments. The *inputVarName*, *inputVarType*, and *inputVarClass* variables are defined in the *SequenceDiagram2EFSM* module. Below is the ATL code for this *Lazy Rule*.

```
lazy rule LrTransitionInputVar {
  from
    t : EFSM!Input
  to
    i : EFSM!Variable(
        name <- thisModule.inputVarName,
        type <- thisModule.inputVarType,
        class <- thisModule.inputVarClass
    )
}
```

– **LrTransitionEvent**: it creates an event with the name and return labeled with message data and class equal to the lifeline class that receives the message. The *eventName*, *eventReturn*, and *eventClass* variables are defined in the *SequenceDiagram2EFSM* module. Below is the ATL code for this *Lazy Rule*.

```
lazy rule LrTransitionEvent {
  from
    t : EFSM!Transition
  to
    i : EFSM!Event(
        name <- thisModule.eventName,
        return <- thisModule.eventReturn,
        class <- thisModule.eventClass
    )
}
```

– **LrTransitionEventArg**: it creates event parameters with name and type of the message operations arguments. The *argName*, and *argType* variables are defined in the *SequenceDiagram2EFSM* module. Below is the ATL code for this *Lazy Rule*.

```
lazy rule LrTransitionEventArg {
  from
    t : EFSM!Event
  to
    i : EFSM!Param(
        argName <- thisModule.argName,
        argType <- thisModule.argType
    )
}
```

– **LrContextVariable**: it creates a context variable with the name and type labeled with the return variable of the operation and type of the operation, respectively. The *returnVariable*, and *returnType* variables are defined in the *SequenceDiagram2EFSM* module. Below is the ATL code for this *Lazy Rule*.

```
lazy rule LrContextVariable {
  from
    o : SequenceDiagram!Operation
  to
    v : EFSM!ContextVariable(
        name <- o.returnVariable,
        type <- o.returnType
    )
```

```
}
```

As previously mentioned, one of the improvements of this study compared to the approach described in [19] is the possibility of using combined fragments nested in up to five levels of depth. This was achieved by changing the main transformation rule called *SequenceDiagram2EFSM*. In this rule, when an instance of *InteractionFragment* is found and this fragment is a combined fragment, we again check if the next fragment is also a combined fragment. If this occurs, all transformation rules are considered. This check was implemented at five levels of depth.

All such rules implemented in ATL are available in the *SD2EFSM/ SequenceDiagram2EFSM.atl* file of the repository1.

3.2.1 Complexity of the Transformation

In this section, we discuss how the size of the sequence diagrams influences the size of the generated EFSMs. The size of the EFSMs is measured in terms of the number of states and transitions. This counting will be performed by comparing the elements of the sequence diagrams in regards to the creation of EFSMs states and transitions.

Table 1 shows the relationship between the elements of the sequence diagrams and the number of states and transitions that are generated in the EFSMs.

Table 1 Impact of the sequence diagrams elements on ESFMs size.

Sequence Diagram Element	Number of States	Number of Transitions
LifeLine start = true	1	0
Message type = op or type = si	1	1
CombinedFragment InteractionOperator = opt	0	InteractionOperand quantity at the last level + 1
CombinedFragment InteractionOperator = alt	InteractionOperand quantity - 1	InteractionOperand quantity - 1 + InteractionOperand quantity at the last level + 1
CombinedFragment InteractionOperator = loop	0	1

If the sequence diagram element is a *CombinedFragment* with an *Opt InteractionOperator* and there is a *Message* after the *CombinedFragment*, the number of transitions is equal to the number of *InteractionOperand* at the last level plus 1.

If the sequence diagram element is a *CombinedFragment* with an *Alt InteractionOperator*, the number of transitions is equal to the number of operands minus 1. If there is a *Message* after the *CombinedFragment*, the number of *InteractionOperand* in the last level plus 1 is added. The number of states is equal to the number of *InteractionOperand* minus 1.

3.3 Stubs Generation

Our procedure automatically generates stubs towards a full automation of the testing process. In this step, all stub classes with their respective attributes and operations are generated from the sequence diagram.

This transformation was automatically generated by the Model-To-Text (M2T) transformation implemented in Acceleo, a template-based technology that includes authoring tools for the creation of custom code generators. It enables the automatic production of any type of source code from any data source available in EMF format [5]. We have implemented the *generateStubs* generator module, whose input is the sequence diagram model created by the sequence diagram editor implemented in the EMF. The creation of each element of the stubs is explained as follows:

- **Classes**: a stub class is created for each *LifeLine* instance of the sequence diagram. The class name is the name of the lifeline.
- **Attributes**: the attributes of the classes are obtained from the parameters of the message operations with returns other than *void*. The parameters are selected in two situations: (1) parameter lifeline name (which indicates which lifeline inserts parameter argument values - baton pass) equal to the created class name, or (2) name of the target lifeline of the message equal to the name of the created class. In these two cases, the attribute name and its type are obtained, respectively, from the types and argument names of the message operation of the sequence diagram. Other attributes are obtained from messages with returns other than *void*, which are of the *GET* type (name beginning with "get") and whose target lifeline name is the same as that of the created class. In this scenario, the name of the attribute and its type are obtained, respectively, from the return variable and the return type of the message operation of the sequence diagram.
- **Operations**: for each attribute created in the class, a GET operation is created under the same conditions of the creation of the attributes of the class. The body of the operation will be the return of the attribute defined in the created class. Other operations are also obtained from messages with returns other than *void*, which are of the *GET* type (name beginning with "get"), and the destination lifeline name is the same as the created class. In this scenario, the name of this operation is the same as the message, the return of the operation is the same type as the return of the message operation, and the parameters of the operation are the same as those of the arguments of the message operation. The operation body will be the return of the message operation return variable. For all messages whose

target lifeline name is the same as that of the class name, a new operation is created. The name of this operation is the same as that of the message, the return of the operation is of the same type as the message operation and the parameters of the operation are the same as the arguments of the message operation. If the message operation guard is filled, the operation body is created with the conditional test of the message operation guard. If the return of the message operation is a boolean, the conditional test returns *true* or *false*. Otherwise, the return variable will return.

The generator code implemented in Acceleo is available in the *Sd2Stubs/src /Sd2Stubs/main/* folder of the repository1.

3.4 Generation of Test Cases

Our procedure uses the ModelJUnit and JUnit libraries for the generation of test cases, since they are open-source and their uses are simple for Java programmers. Moreover, ModelJUnit enables the implementation of widely used formal models (e.g., EFSM) in MBT, and provides a variety of useful test generation algorithms, model visualization features, model coverage statistics, and other features [28].

The implementation of an MBT environment in ModelJUnit and Junit consists of four steps:

1. **The Model**: initially, we implemented the *FsmModel* interface to define our model in ModelJUnit and defined all possible states of our EFSM in an enumeration variable (*enum State*). For each context variable, we defined a variable in the class, and coded action methods (*@Action*) for each input in our model to define the transitions that link the states. We also defined the *getState* method in our model that returns the current state and the *reset* method that takes the machine to the initial state.
2. **The Adapter**: we implemented the *Adapter* class, which enables our model to communicate with and control of our stubs. We added a similarly named method in the *Adapter* class for each action method defined in the model that triggers an event. In our model defined in *The Model*, we add the correct adapter method call to each action method. In addition, in the *Adapter* class we need to instantiate an object for each class of the stubs.
3. **Generation of Tests**: initially, we must instantiate the model defined in the first step, and then, we choose the test strategy to be used. ModelJUnit offers different strategies, namely *GreedyTester*, *LookaheadTester*, and *RandomTester*. We used *LookaheadTester*, since it is a more sophisticated algorithm and can cover all transitions and states quickly [28]. *LookaheadTester* is more efficient than the other two strategies mentioned, since it does not perform random steps like *RandomTester* and, although it is similar to *Greedy Tester*, it provides more refined options, such as look ahead depth and several other parameters. Finally, we applied the *buildGraph*

method to build the graph and generate the tests. This graph is also used to calculate coverage metrics for transitions, states, and action.
4. **Test Concretization**: test cases were implemented in Java, in JUnit library. Two test cases are generated for all transitions that trigger an event and actions are performed. One test case for valid values for guard condition and another for invalid values. The goal is to generate concrete test cases for all possible paths.

These four steps were automatically generated by Model-To-Text (M2T) transformation in Acceleo. We implemented the *generateClassModel*, *generateClassAdapter*, *generateClassTest* and *generateClassJUnit* generators modules, and their input is the EFSM generated in Step (a) of our procedure.

The code generators implemented in Acceleo are available in the *Efsm2ModelJUnit/src/Common/* folder of the repository1.

3.5 Execution of Test Cases

In this step, using the Eclipse Modeling Framework (EMF), the test cases are performed from the *ClassTest* and *ClassJUnit* classes described in the steps 3 (Generation tests) and 4 (Test Concretization) of the previous section.

In the execution of the abstract test cases, in addition to the test paths, the ModelJUnit library also generates action, state, and transition coverage metrics. The following is a brief explanation of these coverage metrics:

- **State Coverage**: shows the comparison between the number of states covered by the number of states defined in the model.
- **Transition Coverage**: shows the comparison between the number of executed transitions by total numbers of transitions defined in the model.
- **Action Coverage**: shows the comparison between the number of executed actions by total numbers of actions defined in the model.

In the execution of concrete test cases, the JUnit library generates the verdicts of the execution of the tests.

4 Case Study

This section describes a case study conducted on real software Teacher Record Book of the State University of Piaui called Professor Online (sistemas4.uespi.br /ProfOnline) to evaluate the applicability of our procedure. This system has the following features:

- **User Validation**: when the teacher accesses the system, the teacher is asked for their credentials (*id, psw*). After validating the credentials, the teacher can access the other features of the system.
- **My Classes**: it enables the teacher to choose the class to be updated. After the choice, all other features are related to this class.

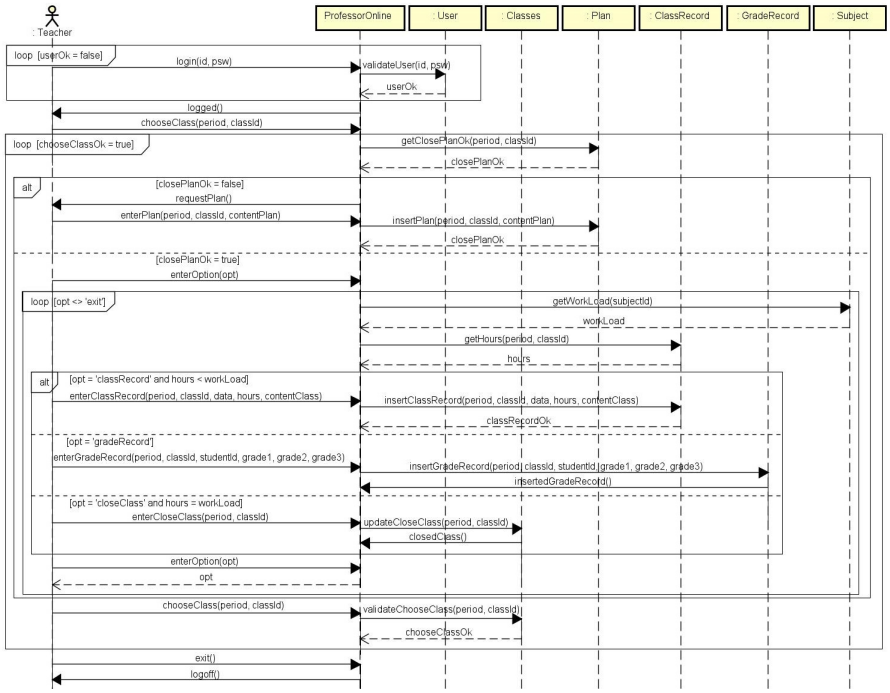


Fig. 15 ProfessorOnline sequence diagram.

- **Program Course**: it enables the teacher to complete the Program Course. All other features are available to the teacher only after the course program has been closed.
- **Lesson Record**: it enables the teacher to complete a lesson until the subject workload is completed.
- **Grade Record**: it enables the teacher to fill in student grades.
- **Class Close**: the teacher can close the class only if all information has been posted. If the class is closed, all other features are unavailable.

The UML sequence diagram in Fig. 15 shows interactions on the Teacher Record Book system.

4.1 Transformation between Models

Initially, we created the sequence diagram model described in Fig. 15 using the sequence diagram editor implemented in the EMF. Then, using the transformation rules implemented in ATL described in Section 3.2, 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 Fig. 16. In order to facilitate the understanding of the transitions, we present in

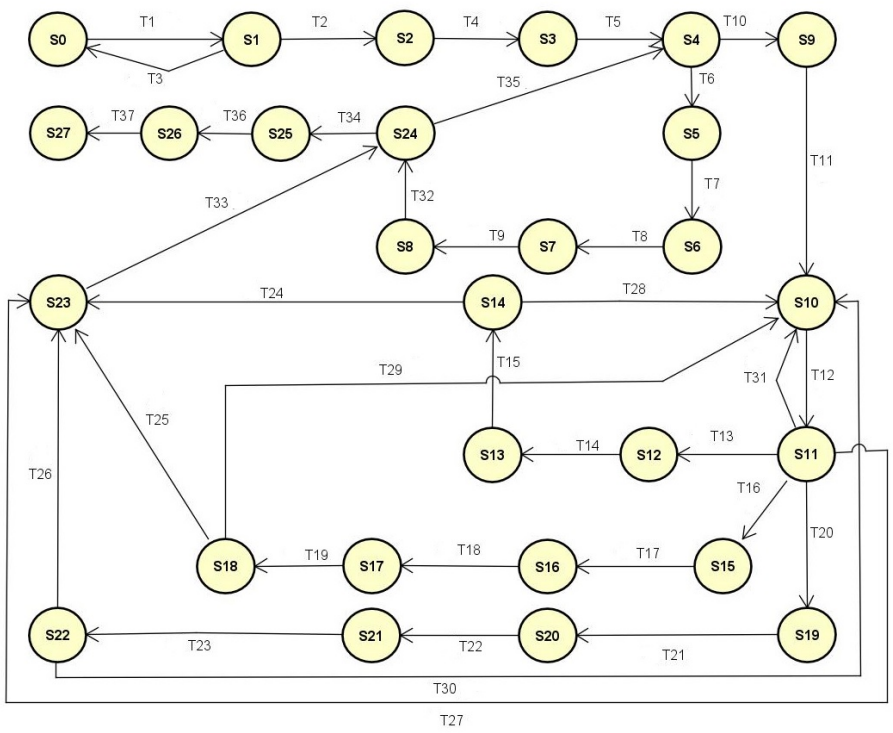


Fig. 16 ProfessorOnline EFSM.

detailing information of the EFSM transitions in Table 2. This model transformation corresponds to Step *(a)* of our procedure.

Table3 shows the relationship between the elements of the sequence diagram and the number of states and transitions that are generated in the EFSM in the case study.

The EFSM model (ProfessorOnline.efsm file) generated is available in the *SD2EFSM/* folder of the repository1.

4.2 Stubs Generation

In Step *(b)*, classes (*Teacher*, *ProfessorOnline*, *User*, *Class*, *Plan*, *ClassRecord*, *GradeRecord* and *Subject*) with their respective attributes and operations are generated automatically from the sequence diagram model described in Fig. 15 by the generator module implemented in Acceleo. Fig. 17 shows an example of a stub class.

Such source codes generated in Java are available in the *Sd2Stubs/Files/* folder of the repository1.

Table 2 Details of EFSM transitions.

Id	Name	Input	Output
T1	S0→S1	login(id, psw)	
T2	S1→S2	validateUser(id, psw)	not (userOk = false)
T3	S1→S0	validateUser(id, psw)	userOk = false
T4	S2→S3	logged	
T5	S3→S4	chooseClass(period, classId)	
T6	S4→S5	getClosePlanOk(period, classId)	closePlanOk = false
T7	S5→S6	requestPlan	
T8	S6→S7	enterPlan(period, classId, contentPlan)	
T9	S7→S8	insertPlan(period, classId, contentPlan)	closePlanOk
T10	S4→S9	getClosePlanOk(period, classId)	closePlanOk = true
T11	S9→S10	enterOption(opt)	
T12	S10→S11	getWorkLoad(subjectId)	workLoad
T13	S11→S12	getHours(period, classId)	opt = “classRecord” and hours < workLoad
T14	S12→S13	enterClassRecord(period, classId, date, hours, contentClass)	
T15	S13→S14	insertClassRecord(period, classId, date, hours, contentClass)	classRecordOk
T16	S11→S15	getHours(period, classId)	opt = ‘gradeRecord’
T17	S15→S16	enterGradeRecord(period, classId, studentId, grade1, grade2,grade3)	
T18	S16→S17	insertGradeRecord(period, classId, studentId, grade1, grade2,grade3)	
T19	S17→S18	insertedGradeRecord	
T20	S11→S19	getHours(period, classId)	opt = “closeClass” and hours = workLoad
T21	S19→S20	insertCloseClass(period, classId)	
T22	S20→S21	updateCloseClass(period, classId)	
T23	S21→S22	closedClass	
T24	S14→S23	enterOption(opt)	not (opt <> “exit”)
T25	S18→S23	enterOption(opt)	not (opt <> “exit”)
T26	S22→S23	enterOption(opt)	not (opt <> “exit”)
T27	S11→S23	enterOption(opt)	not (opt <> “exit”)
T28	S14→S10	enterOption(opt)	opt <> “exit”
T29	S18→S10	enterOption(opt)	opt <> “exit”
T30	S22→S10	enterOption(opt)	opt <> “exit”
T31	S11→S10	enterOption(opt)	opt <> “exit”
T32	S8→S24	chooseClass(period, classId)	
T33	23→S24	chooseClass(period, classId)	
T34	S24→S25	validateChooseClass(period, classId)	not (chooseClassOk = true)
T35	S24→S4	validateChooseClass(period, classId)	chooseClassOk = true
T36	25→S26	exit	
T37	26→S27	loggof	

4.3 Test Case Generation

In Step *(c)*, test cases are generated from the EFSM extracted in Step *(a)*. Classes (*ProfessorOnlineModel*, *ProfessorOnlineAdapter*, *ProfessorOnlineTest* and *ProfessorOnlineJUnit*) are generated automatically in the generator modules implemented in Acceleo.

Table 3 EFSM size of the Case Study.

Sequence Diagram Element	Number of States	Number of Transitions
LifeLine start = true	1	0
Message type = op or type = si	24	24
CombinedFragment InteractionOperator = alt	3	10
CombinedFragment InteractionOperator = loop	0	3
Total	28	37

ProfessorOnlineModel class is an implementation of the *FsmModel* interface. An object called *adapter* instantiated from the *ProfessorOnlineAdapter* class, variable enumeration *State* which represents all states (*S0,...,S27*) of our EFSM, all EFSM context variables, *getState* method, *reset* method, and *@Action* annotated methods are defined in this class.

The objects of *Teacher*, *ProfessorOnline*, *User*, *Plan*, *Subject*, *ClassRecord*, *GradeRecord*, and *Classes* type that belong to stubs were instantiated in the *ProfessorOnlineAdapter* class, where a method was created for each event triggered in EFSM transitions. These methods promote the communication of the model with stubs.

Fig. 17 An example of stub class.

```
public class User {
  private String id = "111";
  private String psw = "123";

  public String getId() {
    return id;
  }

  public String getPsw() {
    return psw;
  }

  public boolean validateUser(String id, String psw) {
    if (this.id.equals(id) && this.psw.equals(psw)){
      return true;
    }
    else{
      return false;
    }
  }
}

```

Objects of *ProfessorOnlineModel* and *LookaheadTester* type were instantiated in *ProfessorOnlineTest*. The algorithm was configured for traversing all transitions and generating a sequence of 70 test steps.

To run the tests we set the attributes of the stub classes to the values below:

- **User**: *id* = "111" and *psw* = "123".
- **Classes**: *period* = "20192" and *classId* = "1".
- **Plan**: *period* = "20192", *classId* = "1", and *closePlanOk* = false or *closePlanOk* = true.
- **Subject**: *subjectId* = "10" and *workLoad* = 60.
- **ClassRecord**: *period* = "20192", *classId* = "1", and *hours* = 30 or *hours* = 60.

Our procedure generated test cases to exercise all machine paths, as shown in Table 4. A test case was generated for each domain. For example, the *id* and *psw* attributes were set to values *111* and *123*, respectively. Then, the procedure generated a test case with *id* = *111* and *psw* = *123* values and another test case with values *id* = *222* and *psw* = *246*, and both scenarios were tested. Fig. 18 shows examples of concrete test cases of *ProfessorOnlineJUnit* class.

Such Java classes (*ProfessorOnlineModel*, *ProfessorOnlineAdapter*, *ProfessorOnlineTest* and *ProfessorOnlineJUnit*) are available in the *Efsm2Model-JUnit/Files* folder of the repository1.

Table 4 Test cases generated.

T	id	psw	period	class Id	close PlanOk	cont. Plan	subj. Id	opt	hours	cont. Class
1	111	123								
2	222	246								
3	111	123	20192	1	true					
4	111	123	20192	2	false					
5	111	123	20192	1	false	aaa				
6	111	123	20192	1	false	null				
7	111	123	20192	1	true		10		60	
8	111	123	20192	1	true		10		30	
9	111	123	20192	1	true		10	class Record	60	
10	111	123	20192	1	true		10	grade Record	60	
11	111	123	20192	1	true		10	close Class	60	
12	111	123	20192	1	true		10	exit	60	
13	111	123	20192	1	true		10	class Record	30	aaa
14	111	123	20192	2	true		10	class Record	30	null
15	111	123	20192	1				exit		
16	111	123	20192	2				exit		

Fig. 18 Examples of test cases concretized in JUnit.

```
import static org.junit.Assert.*;

import org.junit.Test;

public class ProfessorOnlineJUnit {
  @Test
  public void testValidateUser01() {
    User user = new User();
    boolean output = user.validateUser("111","123");
    assertTrue(output);
  }
  @Test
  public void testValidateUser02() {
    User user = new User();
    boolean output = user.validateUser("222","246");
    assertFalse(output);
  }
}
```

4.4 Execution of Test Cases

In this step, using the Eclipse Modeling Framework (EMF), test cases are performed using the *ProfessorOnlineTest* class and *ProfessorOnlineJUnit* class.

The algorithm tests all possible actions by running the *ProfessorOnlineTest* class. For each test case, action, state, and transition coverage metrics are generated (see Table 5). The action coverage metric corresponds to the number of actions that were performed. Since the algorithm used tests all possibilities,

Table 5 Generated coverage metrics.

T	Action Coverage	State Coverage	Transition Coverage
1	22/22	4/28	3/37
2	22/22	2/28	2/37
3	22/22	6/28	5/37
4	22/22	7/28	6/37
5	22/22	9/28	8/37
6	22/22	9/28	8/37
7	22/22	8/28	7/37
8	22/22	8/28	7/37
9	22/22	9/28	8/37
10	22/22	9/28	8/37
11	22/22	9/28	8/37
12	22/22	14/28	13/37
13	22/22	11/28	10/37
14	22/22	11/28	10/37
15	22/22	11/28	10/37
16	22/22	13/28	12/37
Total	22/22	28/28	37/37

the action coverage is equal to the number of *@Action* annotated methods. The state coverage metric corresponds to the number of states visited, and the transition coverage metric indicates the number of triggered transitions. In addition, it is important to note that the execution of all generated test cases in Table 5 provides the combined coverage of all EFSM actions, states, and transitions.

The execution of *ProfessorOnlineJUnit* yielded the expected results for each subset of the input domain and no fault was found. In order to make this execution feasible, we implemented a web server with the system's features, adapted the stubs to connect to the server and run the tests on this implementation.

To execute the test cases, we created a project in the *MODELJUNIT* folder of the repository1.

5 Related Works

In this section, we will compare our approach taking into consideration four aspects: used UML diagrams, MDE concepts, use of formal models and concretization of test cases in some programming language.

Applying a model-based testing (MBT) approach, Cartaxo et al. [3] generated test cases from Labeled Transition Systems(LTS) models translated from UML sequence diagrams. Although it is similar to our proposal, i.e., it employs a formal model extracted from the sequence diagram to generate test cases, it neither uses the resources of the MDE for the transformation between the models, nor has tool support; and also does not materialize the test cases in a programming language.

The authors in [2] proposed using overlapping information inherent to multiple views of UML models for automatic testing. The proposal considers a subset of the UML State Machine and sequence diagrams modeling only forbidden scenarios using only *neg* fragment in the sequence diagram. The SPIN [2] model checker checked whether a set of state machines fulfilled a safety property described as neg fragment of a sequence diagram. The proposal uses UML models to detect errors, however it differs from our method since it does not use model-driven transformation, uses only the *neg* fragment in the sequence diagram, and does not concretize test cases in a programming language.

The authors in [16] described a systematic test case generation method performed on model-based testing (MBT) approaches using UML sequence diagram. The UML sequence diagram is converted into a graph sequence and the graph is traversed for selecting predicate functions. Then, the predicates are transformed into an extended finite state machine (EFSM), from which test cases are generated according to state coverage, transition coverage and action coverage. The technique is similar to ours, however it is not automatically generated from the sequence diagram, and does not use some of its important

[2] SPIN as a general tool for verifying the correctness of distributed software models in a rigorous and mostly automated fashion.

constructions (e.g., combined fragment). Test cases are concretized in the Java programming language.

The approach introduced in [27] generates test cases using UML activity and sequence diagrams. It consists of transforming the sequence diagram into a graph called *Sequence Graph*, and transforming the activity diagram into an *Activity Graph*. The software graph is formed integrating the two graphs traversed to generate the test suite. The proposal uses UML models for generating tests, but differs from ours, since it uses neither MDE concepts, nor formal models for test generation, and does not concretize test cases in a programming language.

Muthusamy et al. [13] designed an approach that generates test cases using UML sequence diagrams. It transforms sequence diagram into a sequence diagram graph (SDG) and generates test cases from this SDG. The sequence diagram is built in Object Constraint Language (OCL) and SDG defines activities as nodes and interactions as paths. The test cases are 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 or formal models, and test cases are not implemented in any programming language.

Seo et al. [23] developed a method that generates test cases from sequence diagrams. This method suggests generating 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, however it does not use a formal model for generating test cases. Moreover, 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. The approach does not concretize test cases in a programming language.

Table 6 shows how our approach and related works are related to the four aspects considered in the comparison, i.e., use of other UML diagrams, tool support, use of formal models, and concretization of test cases in a programming language.

Table 6 Comparison with related works.

Article	Other Diagrams	MDE Concepts	Formal Model	Concretization Test Case
[3]	No	No	Yes	No
[2]	Yes	No	No	No
[16]	No	No	Yes	Yes
[27]	Yes	No	No	No
[13]	Yes	No	No	No
[23]	Yes	Yes	No	No
Our	No	Yes	Yes	Yes

6 Conclusions and Future Work

This paper has presented a systematic procedure for the generation of test cases from UML sequence diagrams, which uses concepts of model-driven engineering to formalize UML sequence diagrams into EFSMs, and ModelJUnit and JUnit libraries for the automatic generation of test cases.

One of the strengths of our approach is the automatic model transformation. As we have developed a prototype 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 UML has semantic problems and formal models provide a set of techniques based on precise notation that can accurately translate the behavior of a system. Since our main objective is to generate tests, our approach uses JUnit library to concretize the test cases in Java programming language. On the other hand, a limitation identified is the use of only one UML diagram.

In Step (a), for the transformation of UML sequence diagram to EFSM, we perform the mapping of the elements of the respective metamodels through transformation rules. By doing this, we can provide a precise semantics to a widely used UML model.

In Step (b), the source code of the stubs is generated from the sequence diagram. This makes it easier for the tester to work since no line of code is required.

In Step (c) of the procedure, the formal model can be used as a 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 stubs and some test strategies already implemented. In addition, at Step (d) the execution of the tests is measured by coverage of states, actions, and transitions. We use the JUnit library to perform tests in the Java programming language.

From the case study, we can observe the applicability of our procedure, 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. Tests were performed and metrics were generated to analyze the behavior of stubs according to the test model created. Importantly, the case study was performed in real software. The sequence diagram used has complex constructions with combined fragments nested at various depths.

As future work, new transformation rules that involve other interaction operators of the sequence diagram defined by OMG and other UML diagrams can be incorporated into the systematic procedure of generating tests and applying to controlled experiments. Moreover, the generated EFSM can be used for formal verifications, such as checking safety, liveness, and fairness properties.

References

1. Bézivin, J., Jouault, F., Touzet, D.: An introduction to the atlas model management architecture (2005)
2. Brosch, P., Egly, U., Gabmeyer, S., Kappel, G., Seidl, M., Tompits, H., Widl, M., Wimmer, M.: Towards scenario-based testing of UML diagrams. In: A.D. Brucker, J. Julliand (eds.) Tests and Proofs, pp. 149–155. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
3. Cartaxo, E.G., Neto, F.G.O., Machado, P.D.L.: Test case generation by means of uml sequence diagrams and labeled transition systems. In: 2007 IEEE International Conference on Systems, Man and Cybernetics, pp. 1292–1297 (2007). DOI 10.1109/ICSMC.2007.4414060
4. Czarnecki, K., Helsen, S.: Feature-based survey of model transformation approaches. IBM Systems Journal **45**(3), 621–645 (2006)
5. EMF, E.M.F.: Acceleo (2018). URL `https://www.eclipse.org/acceleo/`
6. Favre, J.M.: Towards a basic theory to model model driven engineering. In: 3rd Workshop in Software Model Engineering, WiSME, pp. 262–271. Citeseer (2004)
7. Fondement, F., Muller, P.A., Thiry, L., Wittmann, B., Forestier, G.: Big metamodels are evil. In: A. Moreira, B. Schätz, J. Gray, A. Vallecillo, P. Clarke (eds.) Model-Driven Engineering Languages and Systems, pp. 138–153. Springer Berlin Heidelberg, Berlin, Heidelberg (2013)
8. Grønmo, R., Møller-Pedersen, B.: From sequence diagrams to state machines by graph transformation. In: L. Tratt, M. Gogolla (eds.) Theory and Practice of Model Transformations, pp. 93–107. Springer Berlin Heidelberg, Berlin, Heidelberg (2010)
9. Hierons, R.M., Bogdanov, K., Bowen, J.P., Cleaveland, R., Derrick, J., Dick, J., Gheorghe, M., Harman, M., Kapoor, K., Krause, P., Lüttgen, G., Simons, A.J.H., Vilkomir, S., Woodward, M.R., Zedan, H.: Using formal specifications to support testing. ACM Comput. Surv. **41**(2), 9:1–9:76 (2009). DOI 10.1145/1459352.1459354. URL `http://doi.acm.org/10.1145/1459352.1459354`
10. Kent, S.: Model driven engineering. In: International Conference on Integrated Formal Methods, pp. 286–298. Springer (2002)
11. Micskei, Z., Waeselynck, H.: The many meanings of UML 2 sequence diagrams: A survey. Software & Systems Modeling **10**(4), 489–514 (2010). DOI 10.1007/s10270-010-0157-9. URL `https://doi.org/10.1007/s10270-010-0157-9`
12. ModelJUnit: The model-based testing tool. (2010). URL `https://sourceforge.net/projects/modeljunit/`
13. Muthusamy, M., Badurudeen, G.: A new approach to derive test cases from sequence diagram. Journal of Information Technology & Software Engineering **04** (2014). DOI 10.4172/2165-7866.1000128
14. OMG, O.M.G.: Unified modeling language 2.5 (2015). URL `http://www.omg.org/spec/UML/2.5/`
15. OMG, O.M.G.: MOF - meta object facility (2016). URL `http://www.omg.org/spec/MOF/`
16. Panthi, V., Mohapatra, D.P.: Automatic test case generation using sequence diagram. In: A. Kumar M., S. R., T.V.S. Kumar (eds.) Proceedings of International Conference on Advances in Computing, pp. 277–284. Springer India, New Delhi (2012)
17. Petre, M.: UML in practice. In: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, pp. 722–731. IEEE Press, Piscataway, NJ, USA (2013). URL `http://dl.acm.org/citation.cfm?id=2486788.2486883`
18. Pressman, R.S.: Engenharia de Software, 6 edition edn. Mcgraw-Hill Interamericana, Rio de Janeiro (2006)
19. Rocha, M., Simão, A., Sousa, T., Batista, M.: Test case generation by EFSM extracted from UML sequence diagrams. In: The 31 International Conference on Software Engineering & Knowledge Engineering, pp. 135–140 (2019). DOI 10.18293/SEKE2019-133
20. Rutle, A., Rossini, A., Lamo, Y., Wolter, U.: Automatic definition of model transformations at the instance level. pp. 80–81 (2008)
21. Schmidt, D.C.: Guest editor's introduction: Model-driven engineering. Computer **39**(2), 25–31 (2006). DOI 10.1109/MC.2006.58. URL `http://dx.doi.org/10.1109/MC.2006.58`

22. Sen, S., Moha, N., Baudry, B., Jézéquel, J.M.: Meta-model pruning. In: A. Schürr, B. Selic (eds.) Model Driven Engineering Languages and Systems, pp. 32–46. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
23. Seo, Y., Cheon, E.Y., Kim, J.A., Kim, H.S.: Techniques to generate utp-based test cases from sequence diagrams using m2m (model-to-model) transformation. In: 2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS), pp. 1–6 (2016). DOI 10.1109/ICIS.2016.7550832
24. Simão, A.S.: Teste baseados em modelos. In: M.E. Delamaro, J.C. Maldonado, M. Jino (eds.) Introducao ao Teste de Software, chap. 3, pp. 39–57. Elsevier Editora Ltd (2016)
25. Sommerville, I.: Engenharia de Software. Pearson Brasil (2007)
26. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0, 2nd edn. Addison-Wesley Professional (2009)
27. Tripathy, A., Mitra, A.: Test case generation using activity diagram and sequence diagram. In: A. Kumar M., S. R., T.V.S. Kumar (eds.) Proceedings of International Conference on Advances in Computing, pp. 121–129. Springer India, New Delhi (2013)
28. Utting, M.: How to design extended finite state machine test models in java. In: J. Zander, I. Schieferdecker, P.J. Mosterman (eds.) Model-Based Testing for Embedded Systems, pp. 147–170. CRC Press/Taylor and Francis Group, Boca Raton, FL (2012). URL `https://eprints.qut.edu.au/56821/`
29. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing approaches. Softw. Test. Verif. Reliab. **22**(5), 297–312 (2012). DOI 10.1002/stvr.456. URL `http://dx.doi.org/10.1002/stvr.456`
30. Yang, R., Chen, Z., Zhang, Z., Xu, B.: Efsm-based test case generation: Sequence, data, and oracle. International Journal of Software Engineering and Knowledge Engineering **25**(04), 633–667 (2015). DOI 10.1142/S0218194015300018

Mauricio Rocha is Assistant Professor at the Technology and Urbanim Center of the State University of Piauí (UESPI), Teresina, Brazil. He received the B.Sc. degree in Computer Science from the State University of Piauí (UESPI), Teresina, Brazil, in 2002, and the M.Sc. degree in Electrical Engineering from the Mackenzie Presbiterian University, São Paulo, Brazil, in 2008. He is Ph.D. student in computer science from the University of São Paulo (USP), São Carlos, Brazil. His research interests include model-driven engineering (MDE), software testing and formal models.

Adenilso Simão received the B.Sc. degree in computer science from the State University of Maringa (UEM), Brazil, in 1998, and the M.Sc. and Ph.D. degrees in computer science from the University of São Paulo (USP), São Carlos, Brazil, in 2000 and 2004, respectively. Since 2004, he has been a professor of computer science at the Computer System Department of USP. From August 2008 to July 2010, he has been on a sabbatical leave at Centre de Recherche Informatique de Montreal (CRIM), Canada. He has received best paper awards in several important conferences. He has also received distinguishing teacher awards in many occasions. His research interests include software testing and formal methods.

Thiago Sousa is Assistant Professor at the Technology and Urbanism Center of the State University of Piauí (UESPI), Teresina, Brazil. He received the B.Sc. and the M.Sc. degrees in Computer Science from the University of São Paulo, São Paulo, Brazil, in 2002 and 2007, respectively, and the Ph.D. degree in Electrical Engineering from the University of São Paulo, São Paulo, Brazil, in 2013, with a doctoral training at the University of Southampton, England, in 2011. He has experience in formal methods and model checking.

Author Photographs

Mauricio Rocha

Adenilso Simão

Thiago Sousa