Model-Driven Software Development Combined with Semantic Mutation of UML State Machines

Anna Derezinska[0000-0001-8792-203X] and Łukasz Zaremba

Institute of Computer Science, Warsaw University of Technology, Nowowiejska 15/19, 00-665 Warsaw, Poland
A.Derezinska@ii.pw.edu.pl

Abstract. The paper presents an approach to semantic mutation of state machines that specify class behavior in Model-Driven Software Development. The mutations are aimed at different variants of UML state machine behavior. Mutation testing of a target application allows to compare different semantic interpretations and verify a set of test cases. We present a notation of a process combining model-driven development with semantic mutation and semantic consequence-oriented mutations. Origin and details of the proposed mutation operators are discussed. The approach has been supported by the Framework for eXecutable UML (FXU) that creates a C# application from UML classes and state machines. The tool architecture has been reengineered in order to apply semantic mutation operators into the model-driven development process and realize testing on a set of semantic mutants. The tool and the implemented mutation operators have been verified in a case study on a status service for a social network.

Keywords: Model-Driven Software Development, State machine, Code generation, Mutation testing, Framework for eXecutable UML (FXU), C#.

1 Introduction

Improvement of test sets is a challenging task which can be assisted by mutation testing [1,2]. Software artefacts could be modified in different approaches to mutation testing. Testing of resulting applications can be further verified against test sets in order to evaluate their quality and enhance with new test cases.

Mutation testing could be applied to any kind of software applications, in particular, those build in Model-Driven Software Development (MDSD) [3]. In this case, not only code-based , but also model-based mutations could be considered. Moreover, potential source models for MDSD could be, apart from structural models, like UML classes, also behavioral models, as state machines [4]. A proper behavior interpretation is in such case of the high importance, because behavior of these models could be directly and automatically reflected in operation of the target application.

This is a pre-print of contribution published as:
Derezinska A., Zaremba Ł.: Model-Driven Software Development Combined with Semantic Mutation of UML State Machines. In: Damiani E., Spanoudakis G., Maciaszek L. (eds) Evaluation of Novel Approaches to Software Engineering. ENASE 2019. Communications in Computer and Information Science, vol 1172. pp. 204-226, Springer, Cham (2020).
The final authenticated version is available online at https://doi.org/10.1007/978-3-030-40223-5_10

In this paper, we address a problem of a process that combines MDSD with semantic mutation of state machine behavior. An important issue is to automate these activities and integrate them in a user friendly manner. One of challenges is automating of test execution for different semantic variants, preserving consistency of generated and supplemented code, as well as code of test cases.

This paper extends the work presented in the ENASE'2019 conference [5]. It discusses steps of the processes of concern and some formal issues. The proposed mutation operators are explained in more detail and supplemented with examples of the operator origin. We have also added more information about tool support, its architecture, notation of semantic specification, and realized process. Finally, we present a case study on a social service that has been used in experimental evaluation of the approach.

Within this paper, we focus on all state machine concepts approved in the UML specification [6]. A target application is built in an object-oriented programming language, especially in C#.

The rest of the paper is organized as follows: fundamentals and basic notions of the combined process are presented in Section 2. In Section 3, related work is discussed. The semantic mutation operators, their origin and meaning, are presented in Section 4. Information of the automatic support for the process are described in Section 5. In Section 6, a case study and the experimental results are discussed. Finally, Section 7 concludes the paper.

2 Process Fundamentals

The basic notions of mutation testing and processes of concern will be introduced.

2.1 Mutation Testing

A main goal of mutation testing is evaluation of quality of a test set and support for development of additional tests if necessary. A "standard" mutation testing approach refers to a program and its set of tests. Based on a given program, a set of its mutants is created. A *mutant* is a variant of the original program into which a simple change, so-called *mutation*, was introduced. A mutation is typically a simple syntactic change of the program. A mutated program should be syntactically correct, but its functionality can differ from the original program. This difference could be detected by appropriate test cases. In the mostly used first-order mutation, a change is injected into one program location per one mutant. A kind of a change, in fact a kind of a program transformation, is specified by a *mutation operator*. Different sets of mutation operators are specified for different programming languages [1].

The basic process of mutation testing can consist of the following steps:

1. Preparing an original program
2. Generation of a set of mutants for the program.
3. Running all program mutants against a set of test cases.

4. Evaluation of results, creation additional test cases if necessary (and repeat from step 3).
5. Optionally correcting faults in the original program (and repeat from step 1).

Apart from a typical code mutation, different source notations can be mutated, e.g., domain libraries, selected features of a paradigm – like concurrent mechanism, logical constraints, component contracts, UML models, and other specifications. According to a selected notation, corresponding mutation operators often imitate possible faults in the field of concern.

A special kind of mutation, discussed in this paper, is semantic mutation [7]. In this case a source notation can remain unchanged, but different mutants refer to different variants of semantic interpretations.

2.2 Model to Code Transformation

Model to code transformation is a core of Model-Driven Software Development. As a source of a transformation, structural models, e.g. class diagrams, as well as behavioral models, e.g. state machines, can be used. When a kind of complete behavioral specification is submitted, e.g. a complete state machine for each class, the target application could automatically operate as given in the input specification.

The basic MDSD process consists of the following steps:

1. Preparing of source models
2. Verification of models
3. Generation of code from models
4. Supplementing program with additional code, and building an executable project using additional standard and specialized libraries [result: code project].
5. Program execution and testing

In general, an MDSD process could be more complicated and cover many feedback loops. After model verifying or code testing the models could be improved, handwritten code can be reverse-engineered to models, modified models have to be transformed, new code can be supplemented, etc. Moreover, at different process stages some tests can be written or generated, e.g. prepared as models or code before source model development, after verification of models, based on the generated code, etc.

2.3 MSDS Process Combined with Mutation Testing

In general, mutation testing can be combined within an MDSD process in different ways. We can take into account various selection of:

- process stages at which mutations are applied,
- source artefacts into which mutations are introduced,
- types of software features that are mutated,
- process stages at which software behavior is evaluated,
- kinds of verification applied to the software.

When models and code are mutated, we can distinguish the following steps in a combined generic process. The main created artefacts are given in brackets.

1. Preparing of source models [result: a base model with its semantics].
2. Mutation of source models with structural mutation operators of models [result: mutated models, unchanged semantics].
3. Verification of mutated models against model constraints
4. Generation of code from models, and creation of mutants
5. Supplementing program with additional code, and building executable projects using additional standard and specialized libraries [result: code projects].
6. Creating of sets of mutants at a code level [result: code projects].
7. Running program and its mutants against a set of test cases.
8. Evaluation of results, creation additional test cases if necessary (and repeat from step 7).
9. Optionally correcting faults in the original model (and repeat from step 1).

In the above process, semantics does not change, therefore, it has not to be specially considered. Information about a mutant constitutes a set of models or a program code, in dependence of the process stage.

Different situation is in the case of semantic mutation. Below, we show the basic steps in a combined generic process in which models are not structurally mutated, but a model semantics is mutated.

1. Preparing of source models [result: a model with its base semantics].
2. Mutation of source models with semantic mutation operators of models [result: unchanged model, mutated semantics].
3. Verification of mutated models against model constraints
4. Generation of code from models, and creation of mutants
5. Supplementing program with additional code, and building executable projects using additional standard and specialized libraries [result: code projects].
6. Creating of sets of mutants at a code level [result: code projects].
7. Running program and its mutants against a set of test cases.
8. Evaluation of results, creation additional test cases if necessary (and repeat from step 7).
9. Optionally correcting faults in the original model (and repeat from step 1).

An important difference should be noticed in the considered artefacts. In the second variant of the combined generic process, the models are not modified, but variants of the base semantics are created.

In this paper, we will discuss realization of a subset of the latter proposed process. We focus on semantic mutation, therefore, the final application will be not mutated at code-level, i.e. step 6 will be omitted.

2.4 Basic Definitions

Semantic of state machines described in [6] builds on a state machine elements interpreted in a given manner. Determination of a practical semantics to be used in a MDSD process is, therefore, a selection of one of interpretations to each notion, taking into account possible combination of interpretations.

Let us denote by F – a set of all concepts of state machines. A concept or a set of concepts can be associated with a unique interpretation or with a set of possible semantic interpretations. We can assume that a set of interpretations applicable for a given concept or a group of concepts is countable and finite. Each element of a powerset of F, $\Pi(F)$ can be associated with a set of applicable interpretations $I(\Pi(F))$.

Therefore, in a context of a state machine, we can specify a semantics S, as a set of functions which map an element of $\Pi(F)$ into a selected interpretation from a set $I(\Pi(F))$.

In a code-based, or other structural mutation, a mutant is a single artefact, for example a modified program or model. In semantic mutation, a mutant that would be tested is specified by a tuple <P, S>, where P is a project code and S one of semantics.

3 Related Work

We discuss here work related to interpretation of state machine behavior, transformation of UML state machines and mutation testing.

3.1 Behavioral State Machines

UML has incorporated some existing modeling notions, as the well-known concepts of state machines. State machines are a kind of hierarchical, event-driven automata proposed by Harel [8]. They are a powerful modelling notion to describe behavior of classes or subsystems. State machines are widely used models in embedded system domain, and other application areas [9].

While interpreting behavior specified by a state machine, we can face many possible variants. The UML specification [6] provides the general boundaries of state machines accepted in UML models. However, it leaves open many unspecified issues and semantic variation points. Moreover, a precise semantics is not a part of the official specification. Therefore, different variants of state machine behavior within UML and apart can be met [10].

This situation might be acceptable during a model development, assuming that a model should cover various approaches. Though, in some cases this could lead to ambiguous interpretations. Higher precision is especially important when a model has to be interpreted or transformed to an executable application. Moreover, in most of implemented solutions, there are different interpretations of behavioral variants, but often without direct declarations about their semantics.

One of possibilities is leave to a user decision about behavioral variants. For example, event handling and queuing polices can be decided by a user of the Umple tool [11]. A similar approach, in which different variants are selected by a user were proposed by Chauvel and Jézéquel [12]. Prout presented another generic approach to creation of a code generator parametrized with semantic variants [13].

An MDSD tool can be associated with a primary selection of possible semantics. Therefore, during development of initial versions of the FXU tool, different problems of state machine interpretation have been resolved [14,15]. In the case discussed in this paper, incorporation of different variants of state machine behavior into solutions offered to a user were considered as a mutation in mutation testing.

3.2 Transformation of UML State Machines

While considering UML models, class models are the main sources of transformations [16]. These structural models contain many notions which have direct mapping to basic structures used in object-oriented programming languages. Apart from structural models, behavioral models, especially state machines, are also common sources in model to code transformations [4]. Different methods are used to transform concepts of state machines into code, e.g., replication of states by attributes, applying state design patterns, and others, [11,17-21].

An alternative to the code generation is an interpretation technique. It could be performed by direct execution of code [22]. Some case studies reported in [23] showed that interpreting UML state machine, although much slower, can give acceptable results in the context of network and system management.

It should be noted that the most of solutions dealing with state machines take into account only a restricted subset of UML notions. More advanced features, such as composite states, in particular with orthogonal regions, different pseudostates, including deep and shallow history, deferred events, entry/do/exit actions or internal transitions have often been omitted. This is, for example, in case of code generation supported by some commercial tools – as IBM Rational Software Architect Developer [24].

State machines taken into account in fUML (Foundation Subset for Executable UML) are also limited [25]. The seminal description of state machine coding in C++ also omits concurrency issues [18].

There are some approaches that try to cover state machine models in more comprehensive way. They considered composite states [11,17], state machines labelled with constrains in the OCL language [21], or composite states with history [14].

Some solutions apply more complete set of state machine concepts, as IBM Raphsody [26], Umple [11], although most of them do not support the C# language. A distinguishing feature of a Framework for eXecutable UML (FXU) [27] is covering of full UML state machines with a target to the C# language.

3.3 Mutation Testing for Programs and Models

Mutation testing has been primarily applied as fault injection method for programs written in different programming languages, e.g. C, Fortran, Java, C++ [1,2]. Mutation operators and tools have also been developed for programs in C# [28,29].

Mutation testing approach has also been used to mutate specifications, constraints, and UML models [30], e.g. class models [31], [32].

Automata-based models were also considered as a mutation source in different approaches. Fabbri at al. focused on mutations in finite state machines [33] and hierarchical state charts [34]. Much research on state machines were dealing with syntactical changes of diagrams [35]. Some others related to syntactic changes of specification expressions labelling transitions in state machines. Those expressions were mutated in a similar way as any other programming code.

Another direction of mutation testing is semantic mutation. Semantic mutation has been specified for behavioral models, mainly state machines [7], in some variants called an implementation mutation [35,36]. In semantic mutation, graph structure of a model is not changed, as in a "standard" mutation testing. In this approach, different semantic interpretations of a model are analyzed [37,38].

4 Mutation Operators

Semantic mutation operators can be defined for different aspects of a state machine behavior.

4.1 Origin of Semantic Mutation Operators

In the code-based mutation testing [1,2] many mutation operators were defined based on common mistakes performed by developers. Expert analysis of programming paradigm, e.g. object-oriented constructions, as well as specific features of different programming languages contributed also to specification of various mutation testing operators.

In case of semantic mutation of UML state machines, the primary source of mutation operators is the UML specification [6]. Operators can be driven from variants included in the specification and some ambiguities or issues left undefined.

Other variants of state machine behavior could be taken into account, exceeding the UML limitations. It could be, for example, original Harel-based statemachine as specified within the STATEMATE environment [39]. However, within this paper we would only discuss approaches that are consistent with the UML specification.

Many of UML specification variants correspond to orthogonal regions in a state machine. As an example, we consider entering a composite state using a history pseudostate. We examine the possible interpretations and show how one of semantic mutation operators was specified.

A composite state can be entered via a history pseudostate, as *State3* in **Fig. 1**. If *State3* were not active before, or its last active substate were a final state, then after entering *State3* the substate *State3_1* will be active.

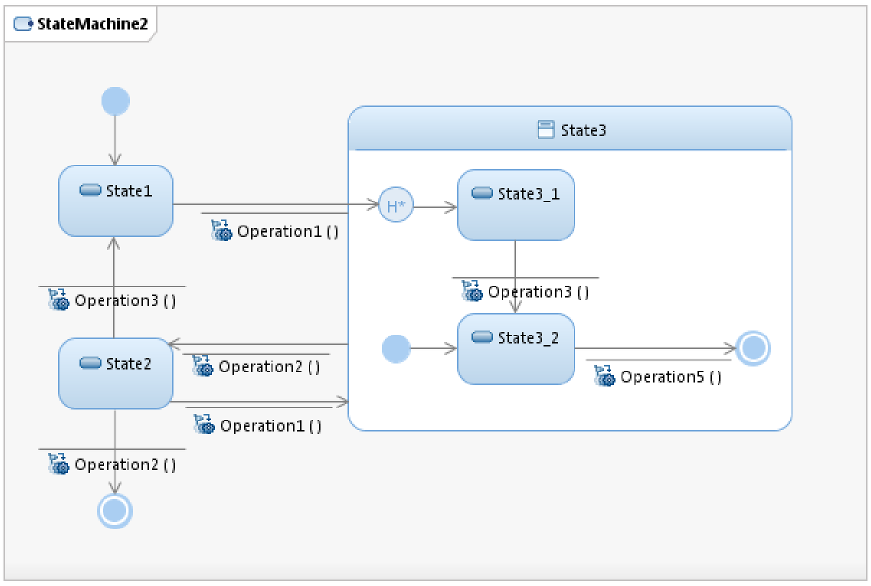

Fig. 1. A state machine example with a *default history state* included in a single region composite state (*State3*)

This is determined by general specification rules. If (i) a composite state was not active before, or (ii) a last active substate included in this composite state was a final state, entering the composite state via a history pseudostate means entering via a *default history state*. However, specification of such a state in the model is not obligatory. This description is sufficient if a composite state has only single region.

Though, when a composite state comprises many orthogonal regions then entering via a history can be interpreted in different ways. This follows from various UML constrains:

- A composite state can include only one history pseudosate, regardless of the number of its regions.
- Only one transition can be outgoing a history pseudostate.
- Between substates included in different orthogonal regions of a composite state no transitions can be specified.
- Entering a composite state causes entering all its orthogonal regions.

Therefore, a default history state can only be defined in one orthogonal region that includes the history pseudostate. The remaining orthogonal regions of this composite state have no default history states. Behavior of a region without its default history state can be specified in the following ways:

1. *Default entry* The region of the composite state is entered according to its default rule.

2. *Automatic completion* The region of the composite state is treated as realized and finished.
3. *Ill-formulation* The model is considered to be ill-formulated. This situation could abandon processing of the state machine due to an error.

These interpretations are illustrated with an example (**Fig. 2**). We can consider a situation when *State2* is active and the next event is processing of *Operation2*. The transitions terminate the composite state *State3* using the history pseudostate. Therefore, for the upper region of *State3*, the default history state will be entered, in this case *State4_2*. For the bottom region of *State3* the default history state could not be specified, and one of the above interpretations could be selected. In case of the first interpretation, *State5_1* will be entered. In the second case, this region is counted to be completed. Finally, according to the third interpretation, the state machine is ill-formulated. The discussed interpretations are further used in a definition of a semantic mutation operator (IV.2 in **Table 4**).

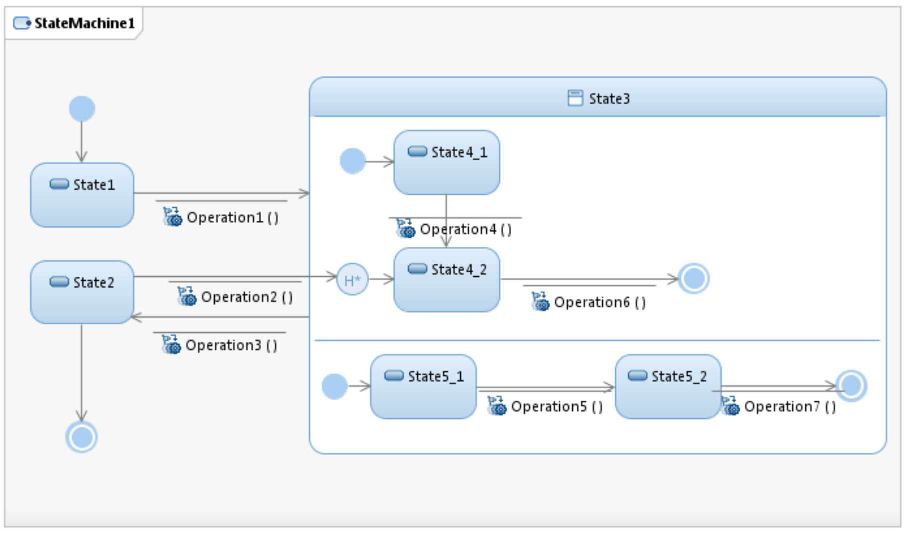

Fig. 2. A state machine example with a *default history state* included in a composite state with many orthogonal regions (*State3*)

4.2 Semantic Mutation Operators of State Machine Behavior

The proposed operators can be divided into several groups referring to different aspects of state machine behavior. In this section we discuss these operators that are summarized in tables (**Table 1-Table 4**).

Mutation Operators of Event Processing State machine behavior is defined by a notion of steps, as in a general labelled transition system. One of basic fundamentals of semantics of UML state machine is *run-to-completion step*. One single event is processed during a single step of a state machine behavior. Management of events is supported by a queue, *event pool*, that stores encountering events. The queue is specified for each state machine of a model.

Policy of a queue is not determined in the UML specification. Selection of five different queue strategies establishes the first operator (I.1 in **Table 1**).

There are different kinds of events that are considered in a state machine. One of them is a *change event*. A change event is associated with a Boolean expression. An event occurs any time when its value changes from False to True. However, the specification does not determine more details, i.e. when the expression is evaluated, what happens if the value changes back to False before an event is detected, etc. Different kinds of a change event management are considered in two mutation operators (I.2 and I.3 in **Table 1**). Operator I.2 covers three policies of detecting an expression change. A change event could be placed into an event pool and waits for processing. Operator I.3 determines policies whether and when the event should be removed from the queue if its value changes to False before processing of the event.

Moreover, some events can be *deferred* in a state. Occurrences of such an event remain in a queue until the event is no longer deferred for all active states from a current state configuration, or the deferred event is explicitly accepted in a trigger of a transition under concern. Two operators (I.4, I.5) deal with the semantics of deferred events. Policy of placing deferred events in a queue can be selected using operator I.4. Policy of selecting deferred event to be processed are considered in operator I.5.

Table 1. Semantic mutation operators dealing with event processing.

ID	Operator	Considered semantics
I.1	Queue policy for selection of events stored in an event pool for a state machine	1) FIFO queue of events 2) FIFO queue of events, with exception of competition event and time events 3) Different priorities assigned to different event types (*MessageEvent*, *ChangeEvent*, *TimeEvent*). FIFO policy within events of the same type. 4) Priority queue for all events 5) LIFO queue of events
I.2	Policy for detection of a change event trigger associated with an expression	1) Evaluation of an expression value periodically for a given time interval, and its comparison 2) An expression is calculated and checked once during a single StateMachine step 3) An expression value is constantly monitored and its change (from *False* to *True*) triggers immediately the corresponding change event.
I.3	Policy of removal of a change event from a state machine event pool	1) An event is removed from an event pool any time the expression associated with the event has changed to *False*. 2) The corresponding expression is calculated during processing of the event. The event is removed from the queue if its expression amounts to *False*. 3) Further changes of the associated expression have no impact the

		change event processing, are disregarded.
I.4	Selection of handling of a deferred event for a state machine	1) A deferred event is placed again in the corresponding event queue, as if the event has encountered once again. 2) A deferred event is added to a special pool of deferred events, globally defined for the whole state machine. 3) A deferred event is placed in a special pool of deferred events which is defined individually for each state.
I.5	Queue policy for processing deferred events of a state	1) FIFO queue of deferred events 2) Different priorities assigned to different event types (*MessageEvent*, *ChangeEvent*, *TimeEvent*). FIFO policy within events of the same type. 3) Priority queue for all events 4) LIFO queue of deferred events

Mutation Operators of Time Management In the general UML specification there are only limited notions of time management. No time delay intervals between time events are established. Event processing time is not predefined neither bounded, by for example some minimal or maximal time. The time issues are open, in order to meet requirements of different semantic variants that could be associated with different application domains. Time concerns might be specified with the MARTE profile [40]. In this paper we have only referred to basic clocks defined in MARTE, logical clock and chronometric clock. Therefore, only one mutation operator for selection of a time processing strategy is proposed (**Table 2**). More details of MARTE are not considered in this paper.

Table 2. Semantic mutation operators dealing with time management.

ID	Operator	Considered semantics
II.1	Time processing policy in a state machine	1) Time events are processed one after another 2) Logical clock is used for time evaluation and processing of time events. 3) Chronometric clock is used for time evaluation and processing of time events

Mutation Operators of Handling Composite States with Orthogonal Regions
States in a state machine can be simple or composite. A composite state may have one or more orthogonal regions. Using many orthogonal regions, we can model concurrent behavior within a state. Semantic of transition realization into and from such a composite state undergoes the run-to-completion step principle. However, some details are left open in the UML specification.

One of unspecified issues is a kind of concurrency, i.e. how are performed actions considered to be executed simultaneously. The following three approaches have been taken into account. In “truly” concurrent execution, separate physical units are involved, as e.g. different cores of a processor. In this case actions can be realized in the same time. The second variant is a parallel execution, which could be implemented by separate parallel software units, e.g. processes or threads. These software units can be run on different physical units but also on a single core unit. The third approach corresponds to sequential interleaved execution of concurrent actions. These approaches are applied as semantic variants in operators III.1, III.2, and III.3 (**Table 3**). The operators are related to execution of actions associated with a transition connected to a state with orthogonal regions. Operator III.1 deals with execution of *exit* actions when many source states are left concurrently. Operator III.2 selects strategy for execution of actions of concurrent transitions. Finally, operator III.3 manages *entry* actions in many target states. The actions are supposed to be concurrently executed.

Another undefined issue concern entering a composite state with many orthogonal regions. In UML, a composite state can be entered via an internal initial pseudostate, or an internal substate can be used as a direct target of a transition. There might be a problem, when many orthogonal regions are used. We would call a region to be ambiguous, if it has no initial pseudostate, nor any substate is directly pointed as a starting point. Different strategies resolving this problem are specified in the mutation operator III.4 (**Table 3**). In this case it is also assumed that no history pseudostate is used.

Table 3. Semantic mutation operators dealing with composite states with orthogonal regions.

ID	Operator	Considered semantics
III.1	Execution policy of *exit* actions to be executed simultaneously	1) Concurrent execution (physically true concurrent) 2) Parallel execution 3) Sequential execution
III.2	Execution policy of *transition* actions to be executed simultaneously	1) Concurrent execution (physically true concurrent) 2) Parallel execution 3) Sequential execution
III.3	Execution policy of *entry* actions to be executed simultaneously	1) Concurrent execution (physically true concurrent) 2) Parallel execution 3) Sequential execution
III.4	Policy for default entry to a composite state with at least one region without an initial pseudostate	1) Abandonment of an ill-defined model 2) Behavior realization in ambiguous regions is omitted. 3) Ambiguous regions are treated as executed (final sates are reached if appropriate). 4) Initial states are selected and entered in ambiguous regions.

Mutation Operators of Handling History

Using of history in a composite state is a powerful modeling notion of UML state machines. However, if orthogonal regions are used, the situation might be interpreted in different ways. We propose two mutation operators related to history. The first one (IV.1) selects an interpretation of a default history pseudostate in orthogonal regions in which it is not defined explicitly. The second operator deals with entering an orthogonal state via a history pseudostate. This problem has been discussed in the previous section.

Table 4. Semantic mutation operators dealing with history pseudostates.

ID	Operator	Considered semantics
IV.1	Selection of an interpretation of a default history psudostate	1) A history pseudostate refers to all regions of the composite orthogonal state in which it is included 2) A history pseudostate only refers to the region in which it is included 3) A history pseudostate refers to the region in which it is included, and also to other regions of its orthogonal state to which no concurrent direct entry exists. 4) A history pseudostate is accepted to be valid only if there are concurrent direct entries to all other regions of the orthogonal state, in which it is included. Otherwise, the model is counted to be ill-modelled.
IV.2	Default entry to an orthogonal state via a history pseudostate	1) Default entering a region 2) A region is considered to be executed (a final substate is reached).

4.3 Operators for Semantic Consequence-Oriented Mutation

Mutation testing in general can be aimed at revealing weaknesses of test sets. A typical motivation derives from the fact that test cases could not detect all test data that should be determined as errors. In this section, we consider another weakness when test data that should be correct might be treated as errors.

In a level of state machine behavior, such tests could refer to consequences of dispatching the same sequence of events. For a given semantics, behavior of a state machine could be different in some random cases. However, it could be still correct and consistent with the semantics.

In particular, this situation can be observed for composite states with orthogonal regions. Many actions could be performed during one transition. The order of these actions is undefined. There are various correct flows that preserve appropriate order of *entry/exit* and transition actions within a single transition.

Mutation operators that are *oriented to semantic consequence* would generate mutants that reflect such different flows. Further, those mutants are used during testing in order to verify whether some test cases do not classify a correct sequence flaw as an erroneous one.

Realization of an operator for semantic consequence-oriented mutation depends of a selected variant of semantics. Three exemplary operators are shown in **Table 5**.

Table 5. Operators for semantic consequence-oriented mutation of state machines

ID	Considered semantics	Operator
V.1	Parallel execution of *entry* actions while incoming orthogonal regions - Semantics 2) for operator III.3	Deterministic order of execution of *entry* actions
V.2	Parallel execution of transitions in orthogonal regions - Semantics 2) for operator III.2	Deterministic order of execution of transitions
V.3	Parallel execution of *exit* actions while outgoing orthogonal regions - Semantics 2) for operator III.1.	Deterministic order of execution of *exit* actions

5 Architectural Support for Semantic Mutation

The Framework for eXecutable UML (FXU) has been designed and implemented as a support for a MDSD process targeted at the C# programming language [20]. Its distinguishing feature is consideration of all notions of UML state machines, including different types of events and actions, various pseudostates, history, orthogonal regions in composite states, etc. The framework consists of two main parts, FXU generator and FXU run-time library. The FXU generator translates UML class and state machine models into a corresponding source code. The FXU run-time library provides implementation for all concepts of state machines. The final application combines a generated code, the library, and additional application-specific code.

It is assumed that a state machine semantics is fixed and independent of an input model. The library includes all necessary interpretations of the state machine behavior. Code generated from a model was used for correct model structure and cooperation of appropriate library elements.

5.1 General Refactored Architecture

Introduction of semantic mutation has required refactoring of the framework architecture. One of the problems is a “code gap”, concerning supplementary code added to the application [41]. When many final applications are created as mutants, the supplementary code should be applied to all the mutants automatically. Other issues concerns performance factors, as e.g. number of projects to be build, number of compilation runs, number of libraries. Four different architecture have been proposed and analyzed [42].The approach based on configurable library proved to be the best solution. It required only single compilation run and one spot where the additional code should be placed for all generated mutants. It gives an easy extensibility with other semantic mutations and simple performing of iterative mutation testing. The selected approach has been implemented in the extended framework. New architecture allows to provide the mutations independently.

The new library includes the following main components: *StateMachineLogic*, *Interfaces*, and *Infrastructure* (**Fig. 3**). An *External Project* cooperates with the interfaces module. It contains a hierarchy of interfaces that corresponds to class hierarchy of different concepts of state machines. Using these interfaces, the state machine elements are accessible in the generated code. The *Interfaces* module includes also other interfaces to cooperate with the internal library objects.

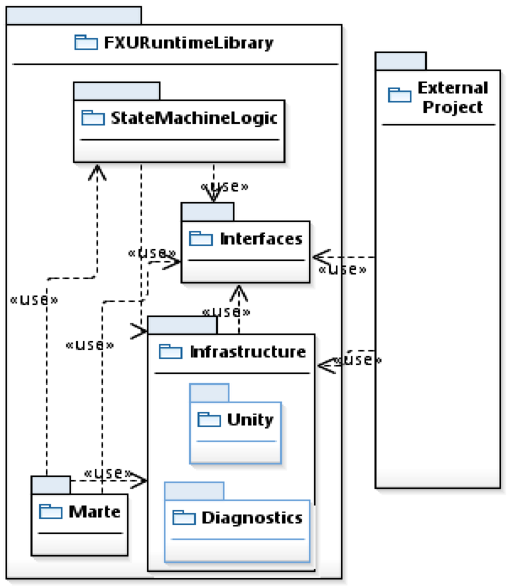

Fig. 3. FXU general architecture

5.2 Container-Based Specification of Mutant Semantics

The *Infrastructure* module consists of two parts: *Unity* and *Diagnostics*. The *Unity* submodule is responsible for dynamic binding of state machine objects according to an interface type or optionally a full name of a context class. The *Diagnostics* submodule provides classes for tracing execution of state machines.

Binding of classes is based on the *dependency injection* pattern realized with dedicated dependency containers. Therefore, all dependencies, except inheritance relations, among classes included in *StateMachineLogic* are deleted. No such dependencies are also between the generated code and *StateMachineLogic* classes. Selection of objects that are provided for a requested interface depends of objects that are registered in a current container of dependency.

A container should be initialized with a set of dependencies. Configuration of a single container represents semantics of a mutant and uses dependency injection based on constructors. It can be stored in an XML-like file. Configurations of all mutants can be specified in a common configuration file. An appropriate configuration has a name which is unique for each mutant. An original, non-mutated program has a default configuration, which corresponds to an unnamed container.

Within a single mutant, various state machines may have different semantics. Therefore, in a configuration of a container, different state machines of a model have to be distinguished. In Appendix I, an example of a configuration file is given. It includes configurations of two containers. The first container describes semantics of an original program. The second one specifies semantics of *Mutant1*. We can observe that the policy of event pool has been changed (in *register* statement) and mapped for one selected state machine.

5.3 Implementation of State Machine Concepts

Realization of different variants of all state machine notions is included in the *State-MachineLogic* module (**Fig. 3**). Different implementations of provided interfaces are given. This code can have only inheritance relations between classes. Different classes that implement the same interface relate to different semantics of the state machine concept.

For example, an interface of a service for default entry to a region is shown in **Fig. 4**. It is implemented by three classes. They correspond to different semantic policies realizing default entry.

Therefore, the module can be easily extended by new implementation variants for the interface, i.e. new semantic variants.

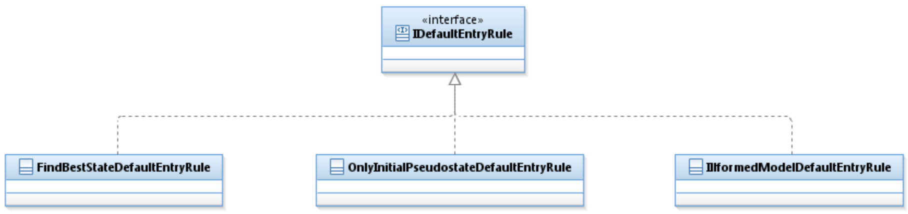
«interface»
IDefaultEntryRule
FindBestStateDefaultEntryRule
OnlyInitialPseudostateDefaultEntryRule
IllformedModelDefaultEntryRule

Fig. 4. Realization of different semantic variants

The *StateMachineLogic* module also realizes a required order of actions in orthogonal regions, according to a given configuration. Execution of a single transition is treated as an asynchronous task from a Task Parallel Library. Its status can be monitored on-line. The order of task execution follows the given semantic configuration or is concurrent if it is not specified explicitly. Events in a main event loop are handled in the similar way, disregarding a direct usage of threads.

A similar architecture has the *Marte* module (**Fig. 3**) that implements selected time concepts from the MARTE profile [40]. It has also been refactored to support semantic mutation, but application of MARTE is beyond the scope of this paper.

5.4 Realization of the Combined Process

General activity flows of code generation and mutation testing supported by the tool are shown in **Fig. 5**. Rounded shaded nodes present activities, and rectangular nodes state for input and output artefacts.

It could be noticed that only one compilation activity is performed regardless of the mutant numbers; models are input for two activities: code generation and generation of configurations; mutation operators influence only generation of configurations.

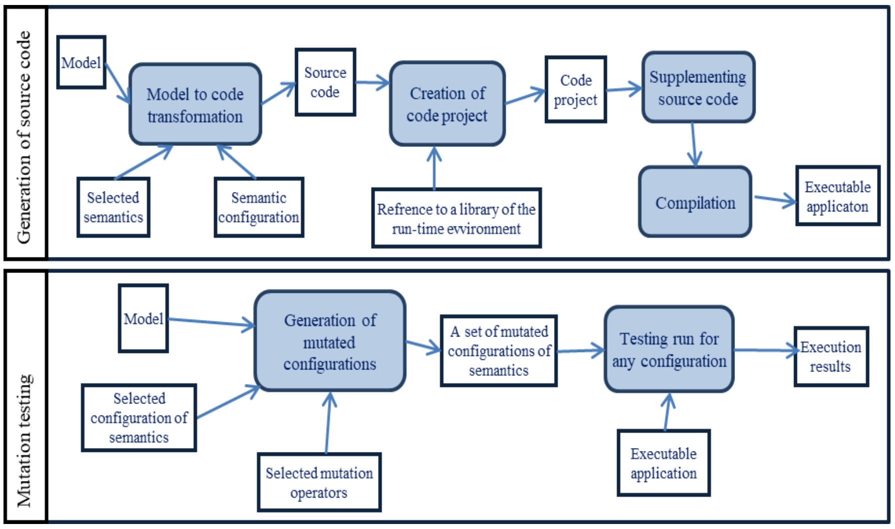
Generation of source code
Model
Model to code transformation
Source code
Creation of code project
Code project
Supplementing source code
Selected semantics
Semantic configuration
Refrence to a library of the run-time ewironment
Compilation
Executable applicaton
Mutation testing
Model
Generation of mutated configurations
A set of mutated configurations of semantics
Testing run for any configuration
Execution results
Selected configuration of semantics
Selected mutation operators
Executable application

Fig. 5. Mutation testing process with configurable library (configurable semantics)

Mutation testing can be performed on two levels:

1. *Class level*. Unit tests of classes are used for direct verification of classes. The class is specified with a state machine. The aim of testing is verification of a class and improvements of its test cases.
2. *Module level*. Test cases are designed for a whole module modelled with classes and their state machines. The testing is focused on the module functionality.

In the first case, the testing should be performed independently for each class. Semantic operators can modify the whole model, but modification of only one state machine will be counted.

The process schema of module testing is similar to the class testing process. However, initial test cases should be focused of verification of the functionality of the whole module. The main difference is dealing with not only one but many state machines. Therefore, it is possible to mutate all state machines with the same mutation operators (i.e. using the same semantic variants), or to mutate each state machine behavior independently with different mutations.

6 Case Study Evaluation

The proposed approach has been evaluated by mutating a case study. The case study focused on a status service for a social network and was used in some previous research on MDSD [43]. The central part of the system is a presence server (PA) (**Fig. 6**). Status data of selected users is stored in the system. The services filter information in accordance to various relationships among users. The presence server manages statuses of users, i.e. creates presence status, accepts and delates subscribers, notifies statuses to a list of subscribers, etc. The presence server can fetch a current status of a user from the SIP (Session Initial Protocol) service. The system is divided into three layers dealing with client communication, status management, and inter-system communication. Each layer has been modeled with a package including further subpackages and classes. Communication between modules has been modeled by appropriate adapters, that could correspond to mock solutions in the system execution.

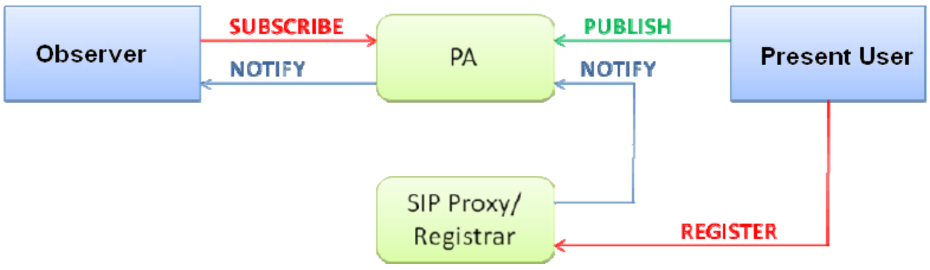

Fig. 6. General structure of the status service with a presence server

The main part of the system, controller of the presence agent, consists of about twenty classes and interfaces. Each class has its state machine that specifies its behavior. In state machines, different modeling structures were applied, including history, *entry/do/exit* actions, transitions with guard conditions and triggers, parallel execution with *for* and *join* pseudostates, composite states with one and many orthogonal regions, etc. Therefore, the comprehensive set of state machine notions has been practically used.

The status service model has been treated as an input model for the MDSD process combined with mutation testing following all its steps. All models have been transformed into code, and apart from code of state machines, some operations have been additionally implemented or mocked. The final project has been supplemented with the semantic configuration.

In experiments, all semantic and semantic consequence-oriented mutation operators have been applied, resulting in a set of corresponding mutants and their semantics.

The application has been tested at the code-level with a set of unit tests devoted to verification of particular classes and functionality of modules of the status service. The main tasks of the presence service have been tested, such as:

– creation of a predefined service status,

— publication of a status to users subscribed in a contact list,
— subscribing to a contact list,
— deleting subscriptions of a presence status.

All the tasks have been realized by a sequence of elementary operations. Different scenarios of handling various requests and diverse time constraints have been taken into account. In result, we could observe and compare behavior for different semantic variants, as primitive tests sometimes do not provide any differences in system behavior. The tests have not revealed any errors of the original application, but this application has been tested beforehand.

However, dealing with different semantic variant, we could verify our expectations of system behavior. Even though, this required creating adequate test scenarios. Consequently, considering semantic mutations encourages developers to build and improve comprehensive test scenario, although, this activity demands still many manual efforts.

7 Conclusion

A process that combines model-driven software development with mutation testing aimed at semantic mutation of behavioral state machines has been presented and realized in a tool support. To the best of authors' knowledge, there is no any other tool realizing semantic mutation for any kind of models. The appropriate architecture enables this kind of mutation operators to introduce efficiently. We have shown details of semantic mutation operators for state machines and their origin. The approach has been verified experimentally on a status service case study.

There are still some process steps that are too laborious, as preparation of semantic configuration, which could be facilitate by more comprehensive tool support. Another challenge to be taken up is assistance in creation of tests to run with mutants. Furthermore, the process support could be extended with other mutations, as structural mutation of models, and code-mutation of C# programs.

References

1. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Transactions on Software Engineering 37(5), pp. 649–678. (2011). doi: 10.1109/tse.2010.62
2. Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Traon, YL., Harman, M.: Mutation testing advances: an analysis and survey. Advances in Computers, 112, 275-378 (2019). doi: 10.1016/bs.adcom.2018.03.015
3. Liddle, S. W.: Model-Driven Software Development. In: Embley, D. W., Thalheim, B. (eds.) Handbook of Conceptual Modeling, pp. 17-54. Springer, Berlin, Heidelberg (2011).
4. Domınguez, E., Perez, B., Rubio, A.L, Zapata, M.A.: A systematic review of code generation proposals from state machine specifications. Information and Software Technology. 54(10), 1045–1066 (2012). doi: 10.1016/j.infsof.2012.04.008.

5. Derezinska A., Zaremba Ł.: Mutating UML state machine behavior with semantic mutation operators. In: Damiani, E., Spanoudakis, G., Maciaszek, L. (eds.) Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE, pp. 385-393. Scitepress, Setubal (2019). doi: 10.5220/0007735003850393
6. UML (Unified Modelling Language), (2017). http://www.omg.org/spec/UML
7. Clark, J. A., Dan, H., Hierons, R. M.: Semantic mutation testing. Science of Computer Programming. 78(4), 345–363 (2013). doi: 10.1016/j.scico.2011.03.011
8. Harel, D.: A visual formalism for complex systems. In: Science of Computer Programming 8(3) 231-274 (1987).
9. Liebel, G., Marko, N., Tichy, M., Leitner, A., Hansson, J.: Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice. Software & Systems Modeling 17(1), pp. 91–113. (2018). doi: 10.1007/s10270-016-0523-3
10. Beeck von der, M.: A comparison of statecharts variants. In: Proc. of the 3rd International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems, London, LNCS, vol. 863, pp. 128-148. Springer, Berlin, Heidelberg (1994). doi: 10.1007/3-540-58468-4_163
11. Badreddin, O., Lethbridge, T.C., Forwared, A., Elaasar, M., Aljamaan, H., Garzon, M.A.: Enhanced code generation from UML composite state machines. In: Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 235-245. SCITEPRESS - Science and Technology Publications, Setubal (2014). doi: 10.5220/0004699602350245
12. Chauvel, F., Jézéquel, J.-M.: Code generation from UML models with semantic variation points. In: 8th International Conference Model Driven Engineering Languages and Systems (MoDELS'05). LNCS vol. 3713, pp. 54–68. Springer, Berlin, Heidelberg (2005). doi: 10.1007/11557432_5
13. Prout, A., Atlee, J.M., Day, N.A., Shaker, P.: Code generation for a family of executable modelling notations. Software & Systems Modeling 11(2), 251–272 (2012). doi: 10.1007/s10270-010-0176-6
14. Derezinska, A., Pilitowski, R.: Interpretation of history pseudostates in orthogonal states of UML state machines. In: Next Generation Information Technologies and Systems. LNCS, vol. 5831, pp. 26–37. Springer, Berlin, Heidelberg (2009). doi: 10.1007/978-3-642-04941-5_5
15. Derezinska, A., Szczykulski, M.: Interpretation problems in code generation from UML state machines - a comparative study. In: Kwater, T. (ed.) Computing in Science and Technology 2011: Monographs in Applied Informatics, Department of Applied Informatics Faculty of Applied Informatics and Mathematics, Warsaw University of Life Sciences, pp. 36-50. (2012).
16. Batouta, Z. I., Dehbi, R., Talea, M., Hajoui, O.: Automation in code generation: tertiary and systematic mapping review. In: 4th IEEE International Colloquium on Information Science and Technology (CIST), pp. 200-205. (2017). IEEE. doi: 10.1109/cist.2016.7805042
17. Sunitha, E. V., Samuel, P.: Object Oriented method to implement the hierarchical and concurrent states in UML State Chart Diagrams. In: Lee, R. (ed.) Software Engineering Research, Management and Applications. Studies in Computational Intelligence vol. 654, pp. 133–149. Springer, Cham (2016). doi: 10.1007/978-3-319-33903-0_10
18. Samek, M., Practical statecharts in C/C++: quantum programming for embedded systems. CMP Books. (2002).

19. Wasowski, A.: Code generation and model driven development for constrained embedded software, Ph.D. thesis, University of Copenhagen (2005).
20. Pilitowski, R., Derezinska, A.: Code generation and execution framework for UML 2.0 classes and state machines. In: Sobh, T. (ed.) Innovations and Advanced Techniques in Computer and Information Sciences and Engineering, pp. 421–427. Springer, Dordrecht (2007). doi: 10.1007/978-1-4020-6268-1_75
21. Iqbal, M. Z., Arcuri, A. and Briand, L.: Environment modeling and simulation for automated testing of soft real-time embedded software. Software & Systems Modeling. 14(1), 483–524 (2013). doi: 10.1007/s10270-013-0328-6
22. Burden, H., Heldal, R.,Siljamaki, T.: Executable and Translatable UML -- How Difficult Can it Be?. In: 18th Asia-Pacific Software Engineering Conference. pp. 5-8, IEEE Comp. Soc., Washington (2011). doi: 10.1109/apsec.2011.37
23. Hoefig, E. Interpretation of Behaviour Models at Runtime: Performance Benchmark and Case Studies. Ph.D. thesis, Berlin Institute of Technology (2011). http://dx.doi.org/10.14279/depositonce-2842, last accessed 2019/08/08.
24. IBM RSA (Rational Software Architect), https://www.ibm.com/developerworks/downloads/r/architect, last accessed 2019/08/08.
25. fUML, Semantics of a Foundation Subset for Executable UML models. formal/2018-12-01, (2018). http://www.omg.org/spec/FUML/
26. IBM RRD (Rational Rhapsody Developer), https://www.ibm.com/developerworks/downloads/r/rhapsodydeveloper/, last accessed 2019/08/08.
27. FXU (Framework for eXecutable UML), http://galera.ii.pw.edu.pl/~adr/FXU/, last accessed 2019/08/08.
28. Derezinska, A., Szustek, A.: Object-oriented testing capabilities and performance evaluation of the C# mutation system. In: Szmuc, T., Szpyrka, M., Zendulka, J. (eds.) Advances in Software Engineering Techniques. CEE-SET 2009. LNCS, vol. 7054, pp. 229–242. Springer, Berlin, Heidelberg (2012). doi: 10.1007/978-3-642-28038-2_18
29. Derezinska, A., Trzpil, P.: Mutation testing process combined with Test-Driven Development in .NET environment. In: Zamojski W., Mazurkiewicz J., Sugier J., Walkowiak T., Kacprzyk J. (eds) Theory and Engineering of Complex Systems and Dependability. DepCoS-RELCOMEX 2015. AISC, vol. 365, pp. 131–140. Springer, Cham (2015). doi: 10.1007/978-3-319-19216-1_13
30. Belli, F., Budnik, C. J., Hollmann, A., Tuglular, T., Wong, W. E.: Model-based mutation testing - approach and case studies. Science of Computer Programming, 120(1), 25-48 (2016). doi: 10.1016/j.scico.2016.01.003.
31. Derezinska, A.: Object-oriented mutation to assess the quality of tests. In: Proceedings of the 29th Euromicro Conference. pp. 417-420. (2003). doi: 10.1109/eurmic.2003.1231626
32. Strug, J.: Applying mutation testing for assessing test suites quality at model level. In: Proc. of the 2016 Federated Conference on Computer Science and Information Systems, FedCSIS, Annals of Computer Science and Information Systems vol.8, pp. 1593-1596. IEEE (2016). doi: 10.15439/2016F82
33. Fabbri, S. C. P. F., Delmaro, M. E., Maldonado, J. C., Masiero, P. C.: Mutation analysis testing for finite state machines. In: Proceedings of the 5th IEEE International Symposium on Software Reliability Engineering. pp. 220-229. . IEEE Comput. Soc. Press (1994). doi: 10.1109/issre.1994.341378
34. Fabbri, S. C. P. F., Maldonado, J. C., Sugeta, T., Masiero, P. C.: Mutation testing applied to validate specifications based on statecharts. In: Proceedings 10th International Sympo-

sium on Software Reliability Engineering (Cat. No.PR00443). (ISSRE'99) pp. 210-219. IEEE Comput. Soc. (1999). doi: 10.1109/issre.1999.809326.
35. Trakhtenbrot, M.: New mutations for evaluation of specification and implementation levels of adequacy in testing of Statecharts models. In: Proceedings of Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007). pp. 151-160. IEEE (2007). doi: 10.1109/taic.part.2007.23
36. Trakhtenbrot, M.: Implementation-oriented mutation testing of Statechart models. In: IEEE International Conference on Software Testing Verification and Validation Workshops (ICSTW), pp.120-125. IEEE (2010). doi: 10.1109/icstw.2010.55
37. Trakhtenbrot, M.: Mutation patterns for temporal requirements of reactive systems. In: IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 116- 121. IEEE (2017). doi: 10.1109/icstw.2017.27
38. Bartolini, C.: Software testing techniques revisited for OWL ontologies. In: Hammoudi S., Pires L., Selic B., Desfray P., (eds.) Model-Driven Engineering and Software Development. MODELSWARD 2016. CCIS, vol. 692. pp. 132-153. Springer, Cham (2017). doi: 10.1007/978-3-319-66302-9_7
39. Harel, D. et al.: STATEMATE: a working environment for the development of complex reactive systems. IEEE Transactions on Software Engineering, 16(4), 403-414 (1990). doi: 10.1109/32.54292
40. Object Management Group, UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. (2018). http://www.omg.org/spec/MARTE/
41. Derezińska, A., Redosz, K.: Reuse of project code in model to code transformation, In: Borzemski, L. at al. (eds.) Information Systems Architecture and Technology, Contemporary Approaches to Design and Evolution of Information Systems, pp. 79-88. Oficyna Wydawnicza Politechniki Wroclawskiej, Wroclaw, Poland (2014).
42. Derezinska, A., Zaremba, Ł.: Approaches to semantic mutation of behavioral state machines in Model-Driven Software Development. In: Proceedings of the 2018 Federated Conference on Computer Science and Information Systems. ACSIS, vol. 15, pp 863–866. (2018). doi: 10.15439/2018f313
43. Derezinska, A., Szczykulski, M.: Towards C# application development using UML state machines: a case study. In: Sobh, T., Elleithy, K. (eds.) Emerging Trends in Computing, Informatics, System Sciences, and Engineering. LNEE, vol. 151, pp. 793–803. Springer, New York (2013). doi: 10.1007/978-1-4614-3558-7_68

Appendix I Configuration File with a Set of Semantics

```
<container>
    <register type="IEventsPool" mapTo="PriorityEventQueue">
    <constructor>
     <param name="callEventPriority" value="1" />
     <param name="changeEventPriority" value="2" />
     <param name="signalPriority" value="3" />
     <param name="afterEventPriority" value="4" />
     <param name="completionEventPriority" value="5" />
    </constructor>
    </register>
```

```
   <register type="IDefaultEntryRule" map-
To="RequiredExactlyOneInitialPseudostate">
    <constructor/>
   </register>
   <register type="IRegion" mapTo="Region">
    <constructor>
     <param name="defaultEntryRule" dependencyType="IDefaultEntryRule"/>
    </constructor>
   </register>
</container>
<container name="Mutant1">
    <register type="IEventsPool" mapTo="EventQueue">
    <constructor/>
    </register>
    <register name="PresenceAgent.utils.Status" type="IEventsPool" map-
To="PriorityEventQueue">
    <constructor>
     <param name="callEventPriority" value="1" />
     <param name="changeEventPriority" value="2" />
     <param name="signalPriority" value="3" />
     <param name="afterEventPriority" value="4" />
     <param name="completionEventPriority" value="5" />
    </constructor>
    </register>
   <register type="IDefaultEntryRule" mapTo="UseMostAppropiateState">
    <constructor/>
   </register>
   <register type="IRegion" mapTo="Region">
    <constructor>
     <param name="defaultEntryRule" dependencyType="IDefaultEntryRule"/>
    </constructor>
   </register>
</container>
```