Mutating UML State Machine Behavior with Semantic Mutation Operators

Anna Derezinska[1][a], Łukasz Zaremba[1]
[1]*Institute of Computer Science, Warsaw University of Technology, Nowowiejska 15/19, Warsaw, Poland*
A.Derezinska@ii.pw.edu.pl

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

Abstract: In Model-Driven Software Development (MDSD), an application can be built using classes and their state machines as source models. The final application can be tested as any source code. In this paper, we discuss a specific approach to mutation testing in which modifications relate to different variants of behavioural features modelled by UML state machines, while testing deals with standard executions of the final application against its test cases. We have proposed several mutation operators aimed at mutating behaviour of UML state machines. The operators take into account event processing, time management, behaviour of complex states with orthogonal regions, and usage of history pseudostates. Different possible semantic interpretations are associated with each operator. The operators have been implemented in the Framework for eXecutable UML (FXU). The framework, that supports code generation from UML classes and state machines and building target C# applications, has been extended to realize mutation testing with use of multiple libraries. The semantic mutation operators have been verified in some MDSD experiments

1 INTRODUCTION

Model-Driven Software Development (MDSD) can be aimed at production of high quality software in a reasonable time or at a rapid prototyping (Liddle, 2011). In both cases the target software should reflect behavioral features introduced in source models. It has been assumed that building an application based on an automatically generated code gains benefits of high-level modelling and analysis. Model to code transformation uses mainly structural models, as UML classes (Batouta et al., 2017).

However, behavioral models, e.g., state machines, are also utilized (Dominguez et al., 2012). State machines are widely used in the embedded system domain, and other application areas (Liebel et al., 2018); though code generation is still not very common in the industrial practice. Moreover, building of such applications impose requirements on their consistency to the source models and verification of the final code.

The latter can be supported by mutation testing. Mutation testing is an approach primarily used for assessment of test set quality and generation of tests satisfying selected criteria (Jia and Harman, 2011). In a standard mutation testing process, syntactic changes, so-called mutations, are injected into a source code and supposed to be detected by test cases. Modified programs, mutants, are run against tests. An evidence of an abnormal program behaviour, i.e. killing of a mutant, admits ability of tests to detect program faults. It could confirm quality of a test suite in regard to the type of faults introduced by mutation

[a] https: //orcid.org/0000-0001-8792-203X

This is a pre-print of a contribution presented on the International Conference on Evaluation of Novel Approaches to Software Engineering- ENASE 2019, http://www.enase.org/?y=2019
The definite authenticated version is available online in SCITEPRESS Digital Library via https://doi.org/10.5220/0007735003850393
Cite this paper as:
Derezinska A. and Zaremba Ł. (2019). Mutating UML State Machine Behavior with Semantic Mutation Operators. In: E. Damiani, G. Spanoudakis, L. Maciaszek (eds.) Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering - Volume 1: ENASE, pages 385-393.

operators during generation of the mutant. The notion of mutations used in this paper follow concepts from mutation testing, and not from genetic algorithms.

Different variations of the standard mutation testing process have been considered, including various software artefacts to be mutated and tested. Not only a program code, but also different kinds of models and specifications have been used as a source in a mutation testing process, (Belli et al., 2016). Among models, mutating of state machines has also been discussed (Trakhtenbrot, 2007).

Moreover, mutation testing operators can refer not only to syntactical changes of an input artefact, i.e. code, model, or specification, but also to its semantic variations or other implementation features (Clark, Dan and Hierons, 2013), (Trakhtenbrot, 2010), (Trakhtenbrot, 2017).

Mutation testing could be combined with MDSD approach in various ways. We propose mutation of semantic features of state machines that are source models in the program development. However, mutation testing results are interpreted not on a model level but on the level of a final application; as in the typical mutation testing of a program. Tests should be capable to detect different kinds of software flaws. Among others they should verify correctness of usage of the state machine formalism to design the project corresponding to given requirements.

Presentation of semantic operators to behavioural features of UML state machines and their possible interpretation variants consistent to the UML specification, is the main contribution of the paper. The proposed operators have been implemented in a tool and verified in a case study. To the best of our knowledge it is the first implementation of mutation operators of such kind.

In this paper we assume an MDSD process in which an executable application is created based on UML classes and hierarchical state machine models (Pilitowski and Derezinska, 2007). The final code project is built with all necessary library notions, so the target application can be run as any other general–purpose application in a standard environment. The variety of semantics is realised by a "multiple library" approach, which has been selected after comparison of four different architectural approaches (Derezinska and Zaremba, 2018).

This mutation testing approach has been implemented in FXU – a tool that supports code generation from UML classes and their behavioral state machines with the target to C# applications (FXU, 2019).

The next Section is devoted to the background of the work. Semantic mutation operators of state machines are discussed in Section III. Section IV describes implementation of the approach in FXU. Section V informs about conducted experiments. Section VI summarises related work and Section VII concludes the paper.

2 SEMANTIC MUTATION IN MDSD

Considering mutation testing in an MDSD process, the following general mutation categories can be recognised, which refer to elements that are mutated:

A) design or construction mutation,
B) semantic mutation,
C) semantic consequence-oriented mutation.

The first category has been mostly used in mutation of source code and UML models, but is not a subject of this paper.

2.1 Semantic Mutation

Introduction of semantic mutation is associated with modification of realization of specific transformations rather than modification of transformation effects. In comparison to the A) category, semantic mutation does not modify a source form of a model or code. Semantic mutations rely on other interpretations of an intermediate form. Usually, transformation rules from a source to an intermediate form have to be modified.

Semantic mutation can be applied to models, therefore transformation of a model to a source code or to another model notation results in another code-model or another meaning-behavior in dependence on an applied mutation. Semantic mutations referring to UML state machines are important as there are many semantic variants consistent with the UML specification (UML, 2017).

Semantic mutation can also be used for a program code. In the contrast to traditional mutation, in this case a source code is not modified but its interpretation is changed. For example, in different programming languages, a scope and precision of embedded types can be different (Clark, Dan and Hierons, 2013).

2.2 Semantic Consequence-Oriented Mutation

This mutation category relates to realization of a specific meaning of a programming concept that was modelled. For example, a final system behavior could be determined by one of system realizations, which is consistent with a given semantics. However, behavior of this system can be nondeterministic. Different

correct scenarios can follow realization of execution of models from orthogonal regions of a state machine. Execution order of operations in orthogonal regions is undefined by the specification. Consequently, many combinations of such operation execution can encounter. Therefore, semantic consequence-oriented mutation could imitate different behavioral combinations in a situation of this kind.

This mutation category principally differs from the previous ones. In this case, a generated mutant corresponds to a correct system behavior, and should not be killed by a test suite. Application of tests with such mutants is based on the integration of the mutated system with other parts.

This kind of mutation was treated as an implementation-oriented mutation (Trakhtenbrot, 2010) specified in the context of the Harel statecharts (Harel, 1987). However, the approach to realization of such mutations proposed in this paper is different to those from Trachtenbrot.

3 SEMANTIC MUTATION OPERATORS FOR STATE MACHINE BEHAVIORS

Different aspects of state machine behaviour have been considered. Therefore, we have distinguished several groups of semantic mutation operators:

I. Event processing.
II. Time management.
III. State behavior in regions belonging to the same orthogonal complex state.
IV. Processing of history pseudostate.

It is assumed that all variants of the possible state machine behavior are consistent with the UML specification (UML, 2017). We discuss exemplary selected operators belonging to all above groups, and possible interpretation variants that might be associated with the operators. The selection of operators originates from the previous research (Derezinska and Pilitowski, 2009), (Derezinska and Szczykulski, 2012) and covers a wide range of possible topics and their interpretations, but of course is not exhaustive.

3.1 Operators of Event Processing

The semantics of UML requires processing of one event per time according to the rule of *run-to-completion step*. Encountering of events is stored in a queue, so-called an *event pool*. A policy of the queue is undefined, hence various queue policies could be considered as interpretation variants.

Moreover, there exists a possibility to defer an event. An event could be deferred, if no transition exists that could be traversed due to this event consumption and in the same time this event is denoted as deferred by at least one of states that belong to the current *active configuration*. The deferred event is kept in the event pool until it can trigger a transition, or a configuration is reached in which it is not deferred any more. Processing of deferred events requires deciding of some interpretation issues and resolving of conflicts about an order of deferred event evaluation.

In result we have considered the five following mutation operators of event processing and their several interpretation variants:

I.1. *Selection of queue policy of events*:
- I.1.1) FIFO,
- I.1.2) FIFO, except completion and time events,
- I.1.3) priority queue – priorities are associated with the different types of events,
- I.1.4) priority queue – priorities are associated with different events,
- I.1.5) LIFO.

I.2. *Detection policy for a change event*:
- I.2.1) periodical checking of the expression value,
- I.2.2) checking of the expression value once during a completion step,
- I.2.3) immediate reaction to detection of change of the expression value (i.e. from False to True).

I.3. *Removal of a change event from an event pool:*
- I.3.1) changing of the expression value to False causes removal of an event associated with this expression from the event pool,
- I.3.2) the expression is assessed during evaluation of the associated event. The event is removed if the expression is False.
- I.3.3) further changes in a value of the associated expression have no impact on the event processing.

I.4. *Interpretation of an event deferment*:
- I.4.1) an event deferment is realized as a placement of the event again in the event pool (as if it has encountered again),
- I.4.2) an event deferment causes adding the event to a special pool of deferred events, which is global for the whole state machine,
- I.4.3) due to an event deferment the event is placed in a special pool of deferred events, which is defined locally for each state of the state machine.

I.5. *Processing of events deferred by the same state:* options I.5.1) - I.5.4) are the same as in I.1.

3.2 Operators of Time Management

The UML specification does not assume any specific time delays between consecutive time events, or any predefined event processing time (neither a minimal time, nor a maximal one). Due to such universal foundation, different semantic variants could be applied to time processing in dependence to a selected application domain. We have identified the following operator and its three interpretation variants:

II.1 S*election of time processing policy in a state machine*:

II.1.1) processing of consecutive events one after another,

II.1.2) processing based on a logical clock for time measurement,

II.1.3) processing based on a chronometric clock for time measurement.

3.3 Operators of Orthogonal States

A complex orthogonal state consists of many regions. An event processing in such a state may cause execution of many transitions during a *run-to-completion step*. Only one transition can be executed in a region. Transitions in the orthogonal regions are executed simultaneously, which could be differently interpreted. In realization of a transition we consider three actions: exiting a source state, transition execution, and entering a target state. These actions have been referred in the following three operators dealing with transitions in orthogonal regions:

III.1. *Execution policy of actions to exit from source states which are executed simultaneously*:

III.1.1) concurrent execution (physically true concurrent – e.g. using different cores, or different processors),

III.1.2) parallel execution (e.g. might be realized by many threads on the same core),

III.1.3) sequential execution (might be given an execution order).

III.2. *Execution policy of transitions to be executed simultaneously*:

options III.2.1- 2.3) are the same as in III.1.

III.3. *Execution policy of actions to enter target states which are executed simultaneously:*

options III.3.1- 3.3) are the same as in III.1.

The next operator deals with orthogonal states were not all initial states are directly defined, but no history pseudostate is used.

III.4 *Default entering a complex state including at least one region without an initial pseudostate:*

III.4.1) the model is treated as ill-defined,

III.4.2) the ambiguous regions are omitted,

III.4.3) the ambiguous regions are counted as successfully executed,

III.4.4) initial states are selected in the ambiguous regions.

3.4 Operators of History Pseudostates

In this operator group we consider application of history pseudostates, referring to both cases of a shallow and deep history. There are various situations that might be interpreted in different ways. One is calling of a nonexistent default history pseudostate. Another is entering a complex orthogonal state via history. Hence, there are two operators:

IV.1. *Selection of a history pseudostate interpretation*:

IV.1.1) a history pseudostate refers to all regions of the complex orthogonal state in which it is included,

IV.1.2) a history pseudostate only refers to the region in which it is included,

IV.1.3) 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,

IV.1.4) 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:*

IV.2.1) lack of a default history pseudostate results in a default entering a region(s),

IV.2.2) regions that do not include default history pseudostates are considered to be executed.

3.5 Operators for Semantic Consequence-Oriented Mutation of State Machines

Some operators from the third category have also been considered. A mutant of the second category can be regarded as a pair: a final code and its semantics. In the case of the third category, a mutant could be specified by a 3-tuple: a code, a semantics, and a constraint of the semantics.

Introduction of a mutation operator results here not in changing of a semantics, as in the second category, but in applying some limits to a selected semantics. Therefore, an operator is specified in relation to a chosen semantic variant.

We have proposed the following mutation operators of this kind:

V.1 *Deterministic order of execution of concurrent entries into orthogonal regions* (it refers to the III.1.1 semantic variant).

V.2 *Deterministic order of execution of concurrent transitions in orthogonal regions* (it refers to the III.2.1 semantic variant).

V.3 *Deterministic order of execution of concurrent exits from orthogonal regions* (it refers to the III.3.1 semantic variant).

4 IMPLEMENTATION OF SEMANTIC MUTATION OPERATORS IN FXU

The discussed approach has been incorporated into an MDSD process supported by the Framework for eXecutable UML (FXU, 2019). The FXU tool has been designed to transform UML models into executable code of a general purpose language (Pilitowski and Derezinska, 2007). It was focused on code generation for all notions of behavioral state machines, including complex states with orthogonal regions, different pseudostates, also with history pseudostate, etc. It was the first tool that supported the C# language as a transformation target, and it has still been treating state machines in the most comprehensive way within this technology domain. The framework was extended with different versions, e.g. considering time concepts from the OMG MARTE profile (Derezinska and Szczykulski, 2017).

The framework includes two main parts: FXU Generator and FXU Run-Time Library. The Generator transforms UML classes and their behavioral state machines into the corresponding C# code. The Library contains implementation of all state machine concepts. After code transformation, a final application is built as a project including the generated code and the library. Consequently, it can be run independently in the .NET environment as any other general-purpose application.

UML models are statically verified during model to code transformation (Pilitowski and Derezinska, 2007). Furthermore, FXU runtime library supports dynamic verification completed during a program execution. Apart from this, the mutation operators are another means of model verification that have been incorporated into the framework.

Four different architectural approaches to realization of semantic mutation have been examined (Derezinska and Zaremba, 2018). The basic complexity metrics were analyzed and compared. After this evaluation, a solution based on a "configurable library" was selected and implemented in FXU. In this case, semantics is expressed in a set of configurable rules that is combined with a target application. Each mutant refers to configurable semantic rules and the target application can be executed in accordance to its semantics. Moreover, two strategies, i.e. *all-state machines* and *one selected state machine*, that could be chosen by a user, were implemented.

Realizations of all state machine details and semantic variants are encapsulated in the run-time library. Source code is generated from the class and state machine models. The model-to-code transformation is responsible only for correct construction of state machines from the components provided by the library. Compilation of the code is performed only once, regardless of the applied semantic mutation operators.

Different semantic variants are driven by selected mutation operators providing, in result, appropriate semantic configurations. Based on these configurations, the desired library versions are used during the application run-time. The application can be executed for any semantic configuration and any test suit, if necessary.

A reconfigured architecture of the FXU Library supports different approaches to state machine behavior, including semantic mutation operators and semantic consequence-oriented mutations. The refactored Run-Time Library consists of the following main components:

`StateMachineLogic` – implements concepts from the state machine domain,

`Interfaces` - includes interfaces for classes of the state machine elements used by generated code, and interfaces for internal communication,

`Infrastructure` – implements an intermediate layer used by generated code,

`Marte` - implements selected time concepts from the OMG MARTE profile.

5 EXPERIMENTS

The MDSD experiments have been conducted combined with mutation testing using semantic mutation. The subject of experiments was a case study that had been used in some previous research on model-driven development (Derezinska and Szczykulski, 2013). The case study concerned modelling of a presence server in a social network. It covered processing user statuses; in particular, user's presence, location, communication possibilities, activities, etc.

The presence server model consisted of three main layers dealing with communication with a client, controlling of presence statuses, and communication with other systems. Each layer was modelled by packages comprising its classes and additional subpackages. Behaviour of the classes was specified by their state machines. The whole model

included about twenty classes and interfaces as well as about seventeen state machines.

The model of presence server was processed by the FXU generator, i.e. class and state machine models were transformed into C# source code, and a code project and appropriate semantic configuration files were created. The selected server functionality was implemented or simulated by refinement of appropriate methods.

The aim of current experiments was not only evaluation of an MDSD process, but also verification of the final application. Different variants of the application and their behaviour could have been compared. The variants corresponded to different semantic variants of UML state machines. In result, behavioural correspondence of an application variant to an interpretation of a related model could have been examined.

A set of unit tests for the application was developed and placed into a test project that belonged to the same VS solution. The test project included also a configuration file of a state machine semantics. Each test class was extended with a method initializing the FXU run-time environment.

There were tests developed to check correctness of only one class and its behavior specified by its state machine. Other kinds of tests were devoted to verification of a whole subsystem, for example servicing of a data publishing request. A test started with an initialization of an object of the presence server. Next, a request for status publishing was created and delivered to the server via TCP. Then, it was verified whether an expected status was set in places of concern.

In order to perform semantic mutation testing, configurations of semantic mutants were used. Configurations mutated semantics for the whole execution of a single test, if a test checked only one class and its state machine. If a test referred to a whole subsystem, two types of mutants were configured, namely:

1) All state machines of the involved classes behaved according to the same semantic variant within the same test run.

2) Different state machines of the involved classes used various semantic variants within the same test run.

The mutated applications were run against all test cases. The mutation testing process was supported by an add-in to VS that managed execution of mutants with tests.

The created tests have finished with correct results in the created environment. We have not observed any discrepancies between requirements expressed in the input models and behaviour of the final applications, assuming given semantic variants. However, it should be noted, that this model was formerly evaluated with its basic semantics and thoughtfully verified (Derezinska and Szczykulski, 2013). The main goal of the experiment was to verify the semantic mutation testing process combined with MDSD and the tool support, and not to find model errors or semantic flaws in the case study.

6 RELATED WORK

There are different areas of research that have contributed to the presented work: model-to-code transformation (in particular from state machines), variation points in behaviour of state machines, and processes of mutation testing.

6.1 Code generation from State Machines

A straightforward transformation from UML models to code is based on class models. However, while dealing with behavioral specification, a transformation can be extended with state machine models. There are many approaches to reproduce these models in a code, such as replicating states by attributes, using state design patterns, and others (Dominguez et al., 2012), (Sunitha and Samuel, 2016), (Samek, 2002), (Badreddin et al., 2014), (Pilitowski and Derezinska, 2007). A special attention has been devoted to transformation of advanced modelling features of state machines, including composite states (Sunitha and Samuel, 2016), (Badredddin et al., 2014), or states with history pseudostates (Derezinska and Pilitowski, 2009).

Contemporarily, several tools support code generation from UML state machines (IBM RSA, 2018). They usually respect only a subset of state machine concepts, while more advanced notions, such as complex states, in particular orthogonal regions in states, deep and shallow history pseudostates, deferred events, entry/do/exit actions or internal transitions might be omitted (Samek, 2002).

There are some solutions that apply more comprehensive set of state machine concepts, as IBM Raphsody (IBM RRD, 2018), Umple (Badreddin et al., 2014), FXU (FXU, 2019), although most of them do not support the C# language.

In this paper we discuss an approach based on the full UML state machine specification. The target is an application built in a general-purpose programming language. Concerning implementation issues we used C# and Visual Studio environment.

6.2 Interpretation Issues of State Machines

Models of UML state machines have been originated from the concepts proposed by Harel (1987). The official UML specification (UML, 2017) has always been imprecise, and included some unspecified places, previously called semantic variation points (Beeck, 1994). All in all, they should be resolved in different ways when a model has to be interpreted or a model-based application has to be built and executed.

There are various ways to handle these problems. In (Chauvel and Jezequel, 2005) authors stipulate different variants to be decided by a user. Selected decisions, about event handling and queuing policies, can be also taken by a user in the Umple tool (Badreddin et al., 2014). Another generic approach to creation of a code generator parametrized with semantic variants has been discussed in (Prout et al., 2012). However, in most of implemented solutions, there are different resolutions of behavioral interpretation problems, but often without precise statements about selections taken.

During development of the FXU tool, different problems of state machine interpretation have been faced and decided (Derezinska and Pilitowski, 2009), (Derezinska and Szczykulski, 2012). Moreover, it could be possible to incorporate different variants of state machine behaviour into solutions offered to a user, and they could be treated as possible modifications in mutation testing.

6.3 Mutation Testing

Mutation testing approach has been employed to applications written in different programming languages (Jia and Harman, 2011), also including C# (Derezinska and Szustek, 2012), (Derezinska and Trzpil, 2015). This methodology was also used to mutate UML models (Belli et al., 2016).

Some research was also devoted to mutation of automata-based models. Many of these works were dealing with syntactical changes of diagrams (Trakhtenbrot, 2007).

Behavioral models, mainly state machines, have been also studied as an object of semantic mutation (Clark, Dan and Hierons, 2013), in some variants called also an implementation mutation (Trakhtenbrot, 2007), (Trakhtenbrot, 2010). In this kind of mutation there are no changes introduced into a model graph structure, but different semantic interpretations are considered (Trakhtenbrot, 2017), (Bartolini, 2017).

7 CONCLUSIONS

Different operators to semantic mutation of state machines have been introduced. The operators and their selected possible interpretations have been implemented in the FXU, the framework that supports building C# applications from class and state machine models. The semantic mutations were applied in mutation testing experiments. Their behaviour was consisted with the expectations.

There are many possibilities of the future enhancement of the approach. Basing on the mutation facility developed in the framework, other mutation operators corresponding to different behavioral variants of state machines can be added. These variants could deal with other interpretations of UML state machines, or solutions consistent with other semantics than derived from the UML specification.

Moreover, structural mutations of state machine models could also be incorporated into the framework. Such mutations deal with other flaws of models than the ones covered in this paper.

While using the current FXU add-in for the Visual Studio, unit tests can be automatically executed for any number of mutants. However, defining of tests and their configuration requires a manual work, which could be automated.

Finally, the concerned applications were developed in the C# language. The model-driven development of a program could be combined with the mutation testing performed at the source code level with standard and object-oriented operators of C# (Derezinska and Szustek, 2012) or with operators applied at the intermediate code (CIL - Common Intermediate Language of .NET) (Derezinska and Trzpil, 2015).

REFERENCES

Badreddin, O. et al., 2014. Enhanced code generation from UML composite state machines. In: *Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD). SCITEPRESS - Science and Technology Publications. pp. 235-245. doi: 10.5220/0004699602350245.

Bartolini, C., 2017. 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. Springer, Cham, pp. 132-153. doi: 10.1007/978-3-319-66302-9_7.

Batouta, Z. I. et al., 2017. Automation in code generation: tertiary and systematic mapping review. In: 4th *IEEE International Colloquium on Information Science and*

Technology (CIST), IEEE. art. no. 7805042, pp. 200-205. doi: 10.1109/cist.2016.7805042.

Beeck von der, M., 1994. 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 863, pp. 128-148.

Belli, F. et al., 2016. Model-based mutation testing—approach and case studies. *Science of Computer Programming*. Elsevier BV, 120(1), pp. 25–48. doi: 10.1016/j.scico.2016.01.003.

Chauvel, F. and Jézéquel, J.-M., 2005. Code generation from UML models with semantic variation points. In: *Proceedings of 8th International Conference Model Driven Engineering Languages and Systems*. LNCS vol. 3713 Springer Berlin Heidelberg, pp. 54–68. doi: 10.1007/11557432_5.

Clark, J. A., Dan, H. and Hierons, R. M., 2013. Semantic mutation testing. *Science of Computer Programming*. Elsevier BV, 78(4), pp. 345–363. doi: 10.1016/j.scico.2011.03.011.

Derezinska, A. and Pilitowski, R., 2009. Interpretation of history pseudostates in orthogonal states of UML state machines. In: *Next Generation Information Technologies and Systems*. LNCS vol. 5831, Springer Berlin Heidelberg, pp. 26–37. doi: 10.1007/978-3-642-04941-5_5.

Derezinska, A. and Szczykulski, M., 2012. Interpretation problems in code generation from UML state machines - a comparative study. In: T. Kwater, 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.

Derezinska, A. and Szczykulski, M., 2013. Towards C# application development using UML state machines: a case study. In: T. Sobh, K. Elleithy, eds., *Emerging Trends in Computing, Informatics, System Sciences, and Engineering*. LNEE. vol. 151 Springer New York, pp. 793–803. doi: 10.1007/978-1-4614-3558-7_68.

Derezinska, A. and Szczykulski, M., 2017. Advances in transformation of MARTE profile time concepts in Model-Driven Software Development. In: *Software Engineering Trends and Techniques in Intelligent Systems (CSOC 2017)*, AISC. Vol. 575 Springer International Publishing, pp. 385–395. doi: 10.1007/978-3-319-57141-6_42.

Derezinska, A. and Szustek, A., 2012. Object-oriented testing capabilities and performance evaluation of the C# mutation system. In: *Advances in Software Engineering Techniques*. LNCS, vol. 7054, Springer Berlin Heidelberg, pp. 229–242. doi: 10.1007/978-3-642-28038-2_18.

Derezinska, A. and Trzpil, P., 2015 Mutation testing process combined with Test-Driven Development in .NET environment. In: *Proceedings of the 10th International Conference DepCoS-RELCOMEX*, Advances in Intelligent Systems and Computing. Vol. 365. Springer International Publishing, pp. 131–140. doi: 10.1007/978-3-319-19216-1_13.

Derezinska, A. and Zaremba, Ł., 2018. 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, IEEE. doi: 10.15439/2018f313.

Domınguez, E. et al., 2012. A systematic review of code generation proposals from state machine specifications. *Information and Software Technology*. Elsevier BV, 54(10), pp. 1045–1066. doi: 10.1016/j.infsof.2012.04.008.

FXU (Framework for eXecutable UML). [Online] Available from: http://galera.ii.pw.edu.pl/~adr/FXU/ [Accessed: 2nd Jan 2019].

Harel, D., 1987. A visual formalism for complex systems. In: *Science of Computer Programming*, Amsterdam, pp. 231-274.

IBM RSA (Rational Software Architect). [Online] Available from: https://www.ibm.com/developerworks/downloads/r/architect [Accessed: 7th Dec 2018]

IBM RRD (Rational Rhapsody Developer). [Online] Available from: https://www.ibm.com/developerworks/downloads/r/rhapsodydeveloper/ [Accessed: 7th Dec 2018]

Jia, Y. and Harman, M., 2011. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, Institute of Electrical and Electronics Engineers (IEEE), 37(5), pp. 649–678. doi: 10.1109/tse.2010.62.

Liddle, S. W., 2011. Model-Driven Software Development. In: D.W. Embley, B. Thalheim, eds., *Handbook of Conceptual Modeling*, Springer, pp. 17-54.

Liebel, G. et al., 2018. Model-based engineering in the embedded systems domain: an industrial survey on the state-of-practice. *Software & Systems Modeling*. Springer Nature, 17(1), pp. 91–113. doi: 10.1007/s10270-016-0523-3.

Pilitowski, R. and Derezinska, A., 2007. Code generation and execution framework for UML 2.0 classes and state machines. In: T. Sobh, ed., *Innovations and Advanced Techniques in Computer and Information Sciences and Engineering*. Springer Netherlands, pp. 421–427. doi: 10.1007/978-1-4020-6268-1_75.

Prout, A. et al., 2012. Code generation for a family of executable modelling notations. *Software & Systems Modeling.* 11(2), Springer Nature, pp. 251–272. doi: 10.1007/s10270-010-0176-6.

Samek, M., 2002. *Practical statecharts in C/C++: quantum programming for embedded systems*. CMP Books.

Sunitha, E. V. and Samuel, P., 2016. Object Oriented method to implement the hierarchical and concurrent states in UML State Chart Diagrams. In: *Software Engineering Research, Management and Applications*. Vol.654, Springer International Publishing, pp. 133–149. doi: 10.1007/978-3-319-33903-0_10.

Trakhtenbrot, M., 2007. 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)*. IEEE. pp. 151-160. doi: 10.1109/taic.part.2007.23.

Trakhtenbrot, M., 2010. Implementation-oriented mutation testing of Statechart models. In: *IEEE International Conference on Software Testing Verification and Validation Workshops (ICSTW)*, pp.120-125. IEEE. doi: 10.1109/icstw.2010.55.

Trakhtenbrot, M., 2017. Mutation patterns for temporal requirements of reactive systems. In: *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. pp. 116- 121. IEEE. doi: 10.1109/icstw.2017.27.

UML (Unified Modelling Language), 2017. [Online] Available from: http://www.omg.org/spec/UML, [Accessed: 7th Dec 2018]