Proceedings of the Federated Conference on Computer Science and Information Systems pp. 863–866
DOI: 10.15439/2018F313
ISSN 2300-5963 ACSIS, Vol. 15

Approaches to Semantic Mutation of Behavioral State Machines in Model-Driven Software Development

Anna Derezińska
Warsaw University of Technology
Institute of Computer Science
Nowowiejska 15/16, 00-665 Warsaw Poland
Email: A.Derezinska@ii.pw.edu.pl

Łukasz Zaremba
Warsaw University of Technology
Institute of Computer Science
Nowowiejska 15/16, 00-665 Warsaw Poland

***Abstract*—Behavior of UML state machines can be a source of interpretation problems in model to code transformation. Different solutions to the semantic variants could be defined as a special kind of mutations, similarly as in the mutation testing. State machines together with class models can be a source of an Model-Driven Software Development process aimed at building an executable application. We have compared several approaches to creating applications based on models in which semantic mutation operators of state machine behavior are used. The most promising approach has been utilized to extend the Framework for eXecutable UML (FXU) with semantic mutation facilities. The framework supports code generation from UML classes and their state machines as well as developing C# applications according to selected mutations of state machine behavior. The tool has been used in evaluation of a case study.**

I. Introduction

AUTOMATIC code generation in Model Driven Software Development (MDSD) can be based not only on structural models, like UML classes, but also on behavioral models, e.g. state machines [1]. One of the obstacles is not sufficient support for generation and verification of applications of this kind.

Mutation testing is used for evaluation of test suites and generation high-quality tests [2]. Syntactic changes injected into a source code are supposed to be detected by test cases. Modified programs, so-called ***mutants***, are run against tests. An abnormal program behavior confirms ability of the tests to detect the type of faults introduced by ***mutation operators*** during mutant generation.

Mutation testing approach has been extended for different software artefacts to be mutated and tested in a software development life cycle. A mutated source can be a model or specification [3], including state machines [4], [5]. A special kind of mutation testing is focused not on syntactical changes of an input, i.e. code, model, or another artefact, but on changes in its semantic or other implementation features [5]-[7].

In this paper we focus of different architectural approaches to combine semantic mutation of state machines into an MDSD process. It is assumed that an executable application is created based on UML classes and hierarchical state machine models [8]. 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 developed application should reflect system requirements that are specified in the input models, therefore the final testing is performed at application level, and not at model level.

Presentation of this idea and possibility of its realization is the main contribution of the paper. We compare different approaches to performing such semantic mutation in regard to their applicability in practice and complexity of realization (Sec. III). Complexity analysis of the approaches helps to select the best one which has been implemented in FXU – a tool that supports MDSD from UML classes and their behavioral state machines with the target to C# applications (Sec. IV). Therefore, we have also shown how the semantic mutation testing can be practically combined into an MDSD process. To the best of our knowledge it is the first implementation of such mutation approach.

II. Related Work

The main background of this work originate from areas of code generation from state machines, interpretation of state machine behavior, and mutation testing.

UML model to code transformation based on class models can be extended with state machine models [1]. Tools that support this usually respect only a subset of notions, omitting complex concurrency issues. Some solutions that apply comprehensive set of state machine concepts do not support C#, apart from FXU [8].

The UML specification has included some semantic variation points, in particular concerning behavior of state machines. They should be resolved in different ways while a model has to be interpreted or a model-based application executed. In most of implemented solutions, there are different resolutions of behavioral interpretation problems, often without precise statements about selections taken.

Mutation testing approach has been used to applications in different programing languages [2], including C# [9]. This idea was also used to mutate UML models, e.g. class

models [3], or automata-based models, mainly dealing with syntactical changes of diagrams [4].

Behavioral models, including state machines, have been also considered as an object of semantic mutation [6], in some variants called also an implementation mutation [4], [5]. In this kind of mutation there are no changes introduced into a model graph structure, but different semantic interpretations are considered [7].

III. Different Approaches to Combining Semantic Mutation of State Machines into MDSD

In a model-based process of software development mutation testing can be used at different levels, applied to various software artefacts, and with evaluation of effects in different process stages. Realizations of mutation of model semantics can be classified into three main types [6]:

1) Simulation/interpretation of a model with different parameters mimicking semantic variants.
2) Semantic expressed in a set of configurable rules that is combined in an executable model or a target application.
3) Imitation of semantic mutations using syntactic changes of elements in a model or in a code.

In this paper we focus on approaches that belong to the second realization type, discuss mutating of input model semantic, but testing the final code application.

A. Categories and Strategies of Mutation Operators

The following general mutation categories which refer to elements mutated in MDSD can be distinguished:

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

The first category includes typical mutation testing defined for programming languages, as well as modifications of input models. However, in this paper we do not deal with this category.

Semantic faults can be imitated by semantic mutation, or semantic consequence-oriented mutation, or such structural mutations that reproduce semantic faults [5]. In comparison to design mutation, semantic mutations do not modify an intermediate source form of a model or code but apply another interpretation of it. Transformation rules from a source to an intermediate form are modified.

The third mutation category is associated with realization of a given meaning of modelled programming concepts. System realization consistent to a given semantic determines a final system behavior. However, according to a semantic, behavior of a system or its part can be nondeterministic. This mutation category is aimed at imitation of different behavioral combinations.

This kind of mutation was considered as implementation-oriented mutation [5] specified in the context of the Harel statecharts. However, an approach to realization of such mutations proposed in this paper is different to those from Trachtenbrot [5]. The details of the semantic operators are beyond the scope of this paper and will be published in [10].

A mutation operator could be applied in many places of a program, but in the ***first order mutation,*** code is changed in one place per one mutant [2]. In general, the number of generated mutants will be denoted as *MN*, and equal to:

$$MN = \sum_{i=1..N} OP_i \quad (1)$$

where N is a number of operators and OP_i is the number of program places in which the *i-th* operator can be applied.

Considering behavior variants, we generally assume that only one operator is applied. However, the application of such operators can be mutually dependent. It means, for example, that only after operator OP_x had been selected, a variant determined by another operator OP_y could be used. Taking into account such dependency of operators, the final mutant can still be counted as a first order mutant under application of a composite operator OP_xOP_y.

A model usually includes many state machines, thus the same operator could be used to one or many state machines at the same time. Hence two strategies could be considered:

1) ***all state machines***, i.e. the same mutation operator refers to all state machines in a model to be transformed,

2) ***one or selected state machines***, i.e. only selected state machines (usually one) have different behavior determined by the operator.

In the first strategy, the number of generated mutants depends linearly on the number of operators and is lower than in the second approach. The interpretation of the behavior is also simpler. The latter strategy could result in higher number of mutants, especially for complex systems with many state machines. The number of all possible mutants is of order of $N*K$ where N is a number of mutation operators and K is the number of state machines.

B. Approach I – Multiple Code Generation

This is a simple approach based on a straightforward creation of code in a MDSD process. For each mutant, i.e. for a pair <model, semantic>, a separate process towards a target application will be performed. A result of the transformation would be a code that implements model with the semantic. The application code might be slightly different for each mutant. Each mutant has its own code project and requires to be compiled.

The main process metrics are summarized in Table I. Approach I is simple and independent of a code generator, but have many disadvantages. The new code has to be written in each mutant while a method body is supplemented. Moreover, it could be repeated before adjusting a mutant to any test run. Therefore, the approach could be used for a quick verification of a semantic mutation operator, but it not convenient for the mutation testing in MDSD.

C. Approach II – Multiple Libraries

Drawbacks of the first approach imply that mutation testing process should be independent from model-to-code transformation and from supplementing of the generated code. This idea has already been partially supported if we

separate code generation and run-time libraries, where the library deliver the semantic of state machines.

The rules of a state machine behavior could be encapsulated in a library, therefore we could mutate the library and not a generated code that could be independent. Model transformation and code supplementing would belong to one process which is independent from the mutation testing process. The model to code transformation is performed only once, and we could also extend some code one time, if necessary. In dependence of a selected mutation, an appropriate library could be chosen and used to build the final target application.

The main drawback of this approach is necessity to maintain many versions of the library (Table I). The number of versions is equal to the product of supported mutation operators and their possible interpretations. Hence, the complexity rises up quite fast.

D. Approach III – Mutants with Common Base

This approach extends the library with different variants of classes that represent various concepts of state machine. The whole idea is similar to the *Strategy* design pattern. Each class implements a single interface that corresponds to one state machine notion. All classes are gathered into one library that could be added to a final application. Based on an input model and selected mutation operators, multiple classes are created for each state machine.

A base class specifying behavior is generated for each model class that has its state machine. This base class could include methods originated from the model class as well as additional methods for initialization of the state machine of a default semantic. Moreover, for each mutant a class derived from the base class is created. Each such inherited class redefines methods for initialization of the state machine corresponding to the semantic of a given mutant.

If a method body is supplemented in the generated code, it can be done once in the base class. Creation of an executable application requires one compilation of all mutants together. Tests should be defined for a base class in order to be run for all generated mutants.

An advantage of this approach is a single implementation of additional code. Selection of mutation operators in experiment iterations, i.e. updating a variant, could be realized by substitution of a constructor in the generated mutant code that would require changing in the code generator, which is not a flexible solution.

E. Approach IV – Configurable Library

In the fourth approach we combine advantages of approaches II and III. Considering a set of mutants, we can use a pair <model, a set of semantics> instead of a set of pairs <model, semantic>. Moreover, the source code generated from a model is explicitly separated from the library code. The generated code can use the library only by dedicated interfaces, placed in an additional intermediate layer. The generated code does not depend on the library classes that implement those interfaces.

In result, one version of code is generated for each state machine of a model. It would be used in an original application and a mutated one. Therefore, supplementing of a method code body is performed only once. We need also only one compilation of the application.

Furthermore, mutation testing process uses one consistent run-time library. Consequently, maintenance and extending of the semantic mutation operators would be uncomplicated. In a single mutant, various state machines can be executed according to different semantic variants, if desired.

This approach has many advantages, but creating of objects of state machines could take more time due to reflection mechanism used in the intermediate layer. On the other hand, this activity is performed only once during the live time of an object that includes a state machine.

F. Comparison of Approaches

The approaches are summarized in Table I. We have compared some relevant metrics of the process and product complexity. Only in the first primitive approach we have to build many code projects (row 1) and supplement the same code in many applications (row 5). In other cases one project is used for all mutants regardless of the number of operators and the number of their interpretations.

When multiple variants are introduced into libraries, only one class in the source code corresponds to one model class (row 2). The second approach requires many libraries (row 3), while the others can use a single one. An important time overhead is associated with multiple compilation, which is necessary for two first approaches only (row 4).

Summing up quantitative data in the upper part of the Table (rows 1-5), we can conclude that the fourth approach has the lowest complexity (1 in all metrics).

The bottom part of the Table assesses the mutation testing process flexibility and extensibility. Here, also the last approach would be the most beneficial. Iterative mutation testing can be easily performed, and new semantic mutation ideas could be easily introduced.

IV. Realization of Semantic Mutation with FXU

Framework for eXecutable UML (FXU) creates executable C# from UML models [8]. It was the first tool that supported transformation of state machines to C#, and still belongs to comprehensive tools that covers all notions of behavioral state machines, with complex, orthogonal states, different pseudostates, also history, etc. [1]. The FXU Generator transforms UML classes and their state machines into C# code. The FXU Library contains implementation of all state machine concepts. The final application is built as a project including the generated code and the library.

Basing on the analytic evaluation of the approaches, FXU has been extended to support semantic mutation of state machines using the fourth approach - configurable library. Both strategies, all-state machines and one selected state machine, have been implemented. The reconfigured FXU Library provides versatility of state machine semantic mutation opera-

TABLE I.
COMPARISON OF APPROACHES I-IV (MN – NUMBER OF MUTANTS)

Metric	I	II	III	IV
1 Number of generated projects	MN	1	1	1
2 Number of code classes originated from a model class which has its state machine	MN (one in a project)	1	MN +1	1
3 Number of run-time libraries	1	MN	1	1
4 Number of compilation runs	MN	MN	1	1
5 Number of spots where the same code is placed in project(s)	MN	1	1	1
6 Mutant creation process independent of code generation	No	Yes	No	Yes
7 Separate compilation needed to create any executable mutant	Yes	Yes	Yes	No
8 Easy extensibility with other semantic mutations	High	Med ium	Med ium	High
9 Difficulty in performing an iterative mutation testing	Low	High	Med ium	Low

tors, including semantic mutations and semantic consequence-oriented mutations (Sec III).

Evaluation of the mutation testing process with the extended FXU has been performed on a case study used in the previous MDD experiments [11]. It referred to modeling of a presence server in a social network. Here, we have focused on the application verification, showing different application alternatives reflecting activities consistent with various semantic variants of UML state machines.

A set of unit tests for the application was developed. The test project was supplemented with a configuration file of state machine semantic. The tests followed two schemata, in which (i) we checked a correctness of only one class and its behavior specified by its state machine, or (ii) a whole subsystem was verified. An example of the latter case could be servicing of a data publishing request. It was verified if a valid status was set in appropriate places. In case of tests that check one class and one state machine, semantic for the whole was mutated. In case of subsystem tests, two types of mutants were configured. (A) All state machines behaved according to the same semantic variant within the same test run. (B) Different state machines of the involved classes used various semantic variants within the same test run.

All tests were run against the created mutants and positively evaluated in the environment. The behavior of the mutants corresponded to expectations given in the input models and semantic variants.

V. CONCLUSION

Different approaches to introducing semantic mutation of state machines have been compared. The best solution in terms of complexity and flexibility has been implemented in the FXU, the framework transforming class and state machine models into C# applications. While using this tool support selected behavioral variants to state machines were accomplished and verified in mutation testing experiments.

REFERENCES

[1] E. Dominguez, B. Perez, A.L. Rubio, and M.A. Zapata, "A systematic review of code generation proposals from state machine specifications," *Information & Software Technology*, 54, no. 10, 2012, pp. 1045-1066. http://dx.doi.org/10.1016/j.infsof.2012.04.008

[2] M. Harman and Y. Jia, "An analysis and survey of the development of mutation testing," *IEEE Transactions Software Engineering*, vol. 37, no. 5, 2011, pp. 649-678, http://dx.doi.org/10.1109/TSE.2010.62

[3] A. Derezińska, "Object-oriented mutation to assess the quality of tests," in *Proc. of the 29th Euromicro Conf., IEEE Comp. Society*, Los Alamitos, California, 2003, pp. 417-420. htpp://dx.doi.org/10.1109/EURMIC.2003.1231626

[4] M. Trakhtenbrot, "New mutation for evaluation of specification and implementation levels of adequacy in testing of statecharts models," in *Proc. of the 3rd Workshop on Mutation Analysis (MUTATION'07)*, Windsor, 2007, pp. 151-160. http://dx.doi.org/10.1109/TAIC.PART.2007.23.

[5] M. Trakhtenbrot, "Implementation-oriented mutation testing of statechart models", *Proc. 3rd Int'l. Conf. on Software Testing, Verification, and Validation Workshops,* Paris, 6-9 April 2010, pp.120-125. http://dx.doi.org 10.1109/ICSTW.2010.55

[6] J.A. Clark, H. Dan, and R.M. Hierons, "Semantic mutation testing," in *Science of Computer Programming*, no. 78 pp. 345-363, 2013. http://dx.doi:10.1016/j.scico.2011.03.011

[7] M. Trakhtenbrot, "Mutation patterns for temporal requirements of reactive systems," in *Proc. of 10th IEEE Intern. Conf. on Software Testing, Verification and Validation Workshops*, 2017, pp. 116-121. http://dx.doi.org 10.1109/ICSTW.2017.27

[8] A. Derezińska and R. Pilitowski, "Realization of UML class and state machine models in the C# Code Generation and Execution Framework," *Informatica* vol. 33, no 4, pp. 431-440, Nov. 2009.

[9] A. Derezińska and A. Szustek, "Object-Oriented Testing Capabilities and Performance Evaluation of the C# Mutation System," in *Proc. 4th IFIP TC2 Central and Eastern European Conference on Software Engineering Techniques CEE-SET 2009*, LNCS, vol. 7054, pp. 229-242, Springer, 2012. http://dx.doi.org/:10.1007/978-3-642-28038-2_18

[10] A. Derezińska, "Mutating state machine behavior", unpublished.

[11] A. Derezińska, M. Szczykulski, "Towards C# application development using UML state machines – a case study," in *Emerging Trends in Computing, Informatics, System Sciences, and Engineering*, T. Sobh, K. Elleithy, Eds. LNEE vol. 151, Springer, 2013, pp. 793-803, htpp://dx.doi.org /10.1007/978-1-4614-3558-7_68