Mutation Testing in Model Accuracy Assessment

Joanna Strug

Abstract Abstract models built during a development of software systems play an important role in producing a high quality system. Any modeling mistakes, if not corrected, will propagate to the further development stages decreasing the quality of the final system and increasing costs of correcting them. It is of primary importance to make sure that the model conforms to all requirements of the stakeholders and ensures proper work of the future system under various conditions. This paper describes a mutation testing based approach to accuracy assessment of conceptual models built at the beginning of a system development. The approach focuses on providing test cases for assessing and measuring accuracy of such model with respect to its ability to handle unexpected and erroneous situations. Mutation testing is usually used to assess quality of test cases, but it can also help to provide, in a systematic and human-unbiased way, a number of test cases representing wide range of unexpected situations.

Keywords Model evaluation · Mutation testing · Software modeling · UML · OCL

1 Introduction

Abstract models, focusing on different characteristics of a system to be designed, are built at all stages of the system development process [1]. Such model, if being accurate reflections of system characteristics of interest for a particular level of abstraction, contributes to successful development of a flawlessly working system. However, modeling involves taking decisions regarding elaboration and even refinement of requirements, definition of the system structure and operations, selection of components performing these operations, and so on. As modeling

J. Strug (✉)
Faculty of Electrical and Computer Engineering, Cracow University of Technology, Kraków, Poland
e-mail: pestrug@cyf-kra.edu.pl

© Springer International Publishing AG 2017

A. Grzech et al. (eds.), *Information Systems Architecture and Technology: Proceedings of 37th International Conference on Information Systems Architecture and Technology—ISAT 2016—Part II*, Advances in Intelligent Systems and Computing 522, DOI 10.1007/978-3-319-46586-9_16

activities are errors prone, accuracy of models cannot be taken for granted. Thus, time and effort should be invested into developing models conforming to requirements of the stakeholders and ensuring proper work of the system under various conditions [2].

Development of a model can be seen as an iterative process involving, after building an initial model, iterative assessment of the model accuracy and its improvements. The assessment step is essential in determining the degree to which the model fulfills the expectations and in providing useful data for its improvement.

Main causes of inaccuracy of a conceptual model include missing and incorrectly modeled requirements, as well as lack of proper handling of unexpected, erroneous situations and data. Faults caused by any of the first two of them can usually be detected by positive test cases reflecting, in a representative way, the intended ways of using the system that are given by a system specification [3]. However, due to the fact that early prediction of unexpected and potentially erroneous situations is limited to some obvious cases, the test cases ability to check if a model can handle them properly is very limited [4]. Thus, an effective approach to an assessment of a model accuracy should also provide a subset of negative test cases [3] designed specifically for checking how the model handles the unexpected situations.

A general approach to mutation testing based negative testing was presented in [5] and in [6] a concept of applying mutation testing to assess a model accuracy, focusing on this second aspect of the accuracy, was introduced. This paper builds on the concept by adding more details, shows how to use information coming from the evaluation to improve a model, and presents a real life case study.

The paper is organized as follows. Section 2 briefly presents background information and related works. The approach to an evaluation of a model accuracy is described in details in Sect. 3, and Sect. 4 presents the case study. In the last section conclusions and directions for future works are given.

2 Background and Related Work

A great number of research deals with various problems related to software testing, but only some of them regard negative testing [3, 7–9]. Techniques, such as equivalence partitioning [8] or stress testing [7, 9] provide advises on selecting test cases with invalid input values or on creating and testing some extreme conditions for the system. However, test cases selected by mean of such techniques may not be able to reflect a wide range of unexpected, but not necessarily extreme, situations.

Mutation testing, a technique that in general serves the derivation of faulty artifacts from correct ones, seems to be adequate for contributing to the solving the problem of providing negative test cases. The technique was originally introduced to evaluate quality of test suites provided for programs [10]. Its application involved generation of a number of faulty version of a correct program (called mutants) by introducing small syntactic changes into the code of the original program and then

running the mutants with tests from the evaluated test suite. The ratio of the number of mutants detected by the tests over the total number of non-equivalent mutants generated for the original program (called a mutation score), determined the quality of the evaluated test suite in terms of its ability to detect faults.

A test quality assessment is still the main application area of mutation testing, but works showing its application at different levels of abstractions and to different artifacts are more and more common (examples of such works are given in [11]). However, papers on using mutation testing for deriving negative test cases are still rare. To the author's best knowledge only a few other researchers have also studied this topic [12–14]. The authors of the work in [13] mutated a model of a system and used model-checking techniques to generate counter-examples showing violation of certain properties. The counter-examples made a set of negative test cases, but due to the fact that mutants of the model represent incorrect behaviour of the system, some of the test cases may in fact detect specification related inconsistencies that can also be detected by positive test cases. Another example of an approach, where tests cases were derived from mutated model was presented in [14]. The approach described in [12] is the most relevant to the one presented in this paper, as it also aims at modifying tests directly. However, the modifications proposed by the authors are random and affect only data processed by a program. The approach presented in [5] and the one presented here provide much wider range of changes introduced in a controlled way by using a set of mutation operators defined by the author specifically to target test cases [5, 6].

3 An Approach to Model Accuracy Assessment

A conceptual model is built upon requirements that specify what a system is expected to do, what kind of data it should process, under which conditions it should operate, and so on. The requirements do not provide a clear answer to a question of how the system should behave when subjected to conditions and data out of its normal scope of operations. Issues arising from such a question need to be resolve during development of the conceptual model, otherwise they will propagate to further development stages and finally would make the final system vulnerable to failures.

Development of an accurate model, handling properly various unexpected and erroneous situations, is not easy, because a developer's ability to predict situations that may lead to failures is rather limited. Thus, it is necessary to run the model with negative test cases being able to trigger a wide range of unexpected use scenarios for the system and observe how the model behaves in these situations. Unfortunately, test cases selected basing on requirements are positive, that is they represent only intended ways of using the system and do not support detection of this kind of problems.

The approach presented here focuses on assessing a model, during its development, with the aim to detect its weaknesses caused by lack of proper handling of

unexpected situations. It deals with this problem by providing and using negative test cases that trigger such situations. The main idea behind this approach is to use mutation testing to generate a number of test cases, each being a modified version of any of the positive test cases. Each mutant is a negative test case, as it represents some unintended use scenario for the system. The model, when run with a mutant, will either behave as if the mutant represented some expected use scenario, or it will fail showing its ability to recognize the fact that the mutant triggered an unexpected way of using the system.

The following subsections outline the approach to a model evaluation and describe its main stages in details.

3.1 An Outline of the Approach

Before the assessment of a conceptual model starts the model, representing requirements defined for a system, and a suite of test cases selected basing on these requirements, have to be provided. The assessment is carried out in three stages (Fig. 1):

1. generation of mutated test cases,
2. execution of mutated test cases, and
3. analysis of assessment results.

The first stage involves generation of mutated test cases. The mutants are obtained by introducing small changes into the original test cases accordingly to a predefined rules, called mutation operators. Then, in the second stage, each mutant is executed against the model and a verdict (accepted or rejected) is assigned to it. Finally, in the last stage, accuracy of the model is calculated basing on the number of mutants accepted and rejected by the model and the mutants are analyzed to provide feedback for improving the model.

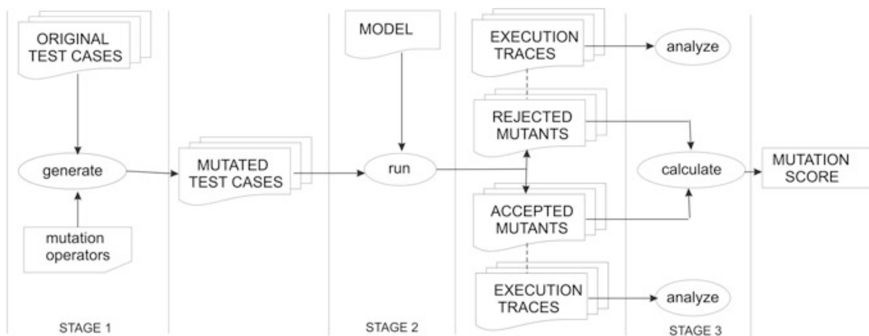

Fig. 1 Outline of the approach

3.2 *Stage 1—Generation of Mutated Test Cases*

The expected input artifact for the first stage is a suite of positive test cases representing all intended ways of using a system, as given by the requirements. A single test case should specify conditions, data and steps (usually represented by operations calls) needed to force a model to behave in a particular way, as well as expected results that the model should produce after executing the test case.

To generate mutants of the test cases a set of mutation operators is needed. Here, a mutation operators is a rule specifying an element of a test case that can be changed and defining the ways it can be changed. A set of mutation operators applicable to test cases consists of the following 9 operators [5, 6]:

- Condition Part Deletion (CPD)—deletes a part of a condition expected to hold before executing a test case,
- Condition Part Replacement (CPR)—replaces a part of a condition by another, compatible element of a condition,
- Operation Call Deletion (OCD)—deletes an operation call (i.e. a step) from a sequence of operation calls in a test case,
- Operation Call Replacement (OCR)—replaces an operation call in a sequence of operation calls in a test case by another operation call,
- Operation Call Insertion (OCI)—inserts an extra operation call in a sequence of operation calls,
- Operation Call Swap (OCS)—changes the order in which two subsequent operation calls should be performed,
- Operation Parameter Replacement (OPR)—replaces a value of a parameter in an operation call (i.e. data passed within a given step) with another, compatible value,
- Operation Parameter Swap (OCS)—changes the order of two values of parameter in a given operation call,
- Operation Target Replacement (OTR)—replaces a target of an operation call with another one.

The mutation operators cover all elements of a typical test case, so the set can be seen as sufficient for generating mutants triggering wide range of unintended ways of using a system. Algorithms outlined in [5] can be used to generate the mutants.

3.3 *Stage 2—Execution of Mutated Test Cases*

The expected input artifacts for the second stage are: the set of mutants generated in the first stage and the model of system undergoing the assessment.

In general the model may represent a system at any level of abstraction, but in this approach a focus is on a conceptual model built at an early stage of the system development. It is therefore expected that the model represents requirements

specified for the system by giving conceptual architecture and functionality of the designed system. The model has to be executable, thus the formalism used to describe the model should include means to define processing. The model, before being assessed with respect to its handling of unexpected situations, should be tested with all original test cases to ensure that the specified requirements have been modeled correctly.

Once the prerequisites concerning the model are satisfied the model is run with each mutated test case. When a mutant is executed its execution trace is generated and linked to the mutant, and after it has been executed a verdict is assigned to the mutant. A verdict rejected is assigned to a mutant when the model fails, otherwise a verdict accepted is assigned to the mutant.

3.4 Stage 3—Analysis of Assessment Results

The third stage uses the results obtained in the second stage: the verdicts assigned to mutants and the execution traces of mutants, to determine the accuracy degree of the assessed model and to provide feedback for refining the model.

The accuracy of a model, in term of its ability to recognized unexpected situation [6], is indicated by a mutation score. Let's for the rest of this paper M denotes the assessed model, T' denotes the set of mutants generated for a set of positive test cases denoted by T, T'_A and T'_R denote subsets of T' consisting of accepted and rejected mutants respectively, and T'_E denotes a subset of equivalent mutants [11].

The mutation score (denoted by MS) for a model M is defined as follows [Eq. (1)]:

$$MS(M) = \frac{|T'_R|}{(|T'_A| + |T'_R|) - |T'_E|} \tag{1}$$

where:

$|T'_A|$ is the number of mutants accepted by the model M,
$|T'_R|$ is the number of mutants rejected by M, and
$|T'_E|$ is the number of equivalent mutants.

The mutation score expresses, in a quantitative way, the degree of a model accuracy. Basing on the value of mutation score one can decide whether the development of the model should be continued or may be finished. The highest possible value of mutation score is 1—it means that all mutants were rejected.

When the mutation score implies that the model accuracy is not acceptable yet, the mutants and their execution traces should be analyzed. An execution trace of a mutant shows, in details, what the model really did when it was run with the mutant. So, an examination of an execution trace of a mutant should help to find out why the mutant was rejected or accepted and prepare a report describing the results of examination.

Rejection of a mutant shows that an evaluated model poses the ability to recognize an unexpected situation defined by the mutant and fails to work. Thus, an examination of an execution trace of the rejected mutant helps to identify the operations that drove the model to fail and to decide (possibly together with stakeholders) what the proper handling of a class of erroneous situations represented by the mutant should be like.

When an evaluated model accepts a mutant, it shows that the model is not able to recognize an unexpected situation and works further producing some, seemingly correct, results. Examination of an execution trace of the accepted mutant helps to identify faulty constraints specifying applicability range of operations performed when the model is run with the mutant, and thus suggests ways of improving the model.

Once, a model has been improved accordingly to the suggestions, it should be assessed again to see if the mutation score has reached an assumed level.

4 Case Study: A HVAC System

The case study demonstrates application of the described approach to evaluation of conceptual models on a software controlling a Heating, Ventilation and Air Conditioning system (HVAC) [15].

In general, the HVAC system maintains the room temperature within an assumed range. A user of the system can turn it on and off and set the temperature range. The system, when turned on, displays its current status, check the temperature in the room and, basing on the current value of the temperature, cools or heats the room.

The approach, as described in Sect. 3 can be applied to assess models described by means of various formalisms, but this work is aimed at models of object-oriented systems. Thus, UML/OCL class diagram was used to model the system [16, 17]. The class diagram describes structure of a system by giving elements (classes) of the system, their properties (attributes), functionalities (operations) and relations between these elements [16], as well as constraints specifying the operations in the form of their pre-conditions and post-conditions [17]. The pre- and post-conditions, are here of particular importance. They define conditions that should hold before an operation starts and when it ends, respectively. Missing or incorrectly specified pre- and post-conditions make the model vulnerable to unexpected behavior. The class diagram representing structure of the HVAC system is given in Fig. 2. The constraints defined for the system are not depicted in this figure for clarity, but an example pre-condition of the operation *regulate()* in class Controller is shown as an annotation in the Fig. 2.

A suite of test cases, the second element required by the assessment approach, was prepared manually. In the case study the suite consisted of only one test case, shown in Fig. 3a. It is given in a format required by the USE simulator [18].

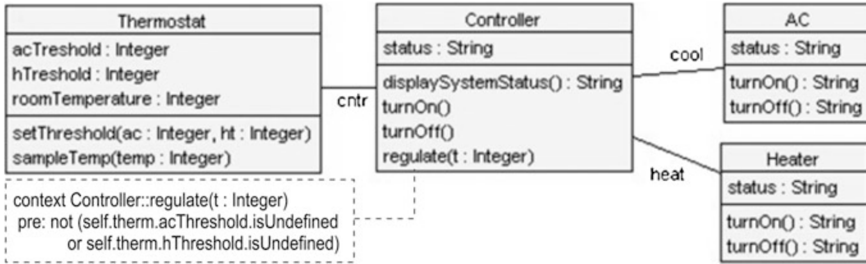

Fig. 2 A class diagram for HVAC system (screenshot from USE)

(a)

```
!therm:=new Thermostat
!cntr:=new Controller
!ac:=new AC
!ht:=new Heater
!insert (therm,cntr) into cntr
!insert (cntr, ac) into cool
!insert (cntr, ht) into heat
!cntr.status:='off'
!ac.status:='off'
!ht.status:='off'

!cntr.turnOn()
!therm.setThreshold(28,18)
!therm.sampleTemp(15)
!therm.sampleTemp(25)
!therm.sampleTemp(30)
!therm.sampleTemp(25)
!cntr.turnOff()
```

(b)

```
!therm:=new Thermostat
!cntr:=new Controller
!ac:=new AC
!ht:=new Heater
!insert (therm,cntr) into cntr
!insert (cntr, ac) into cool
!insert (cntr, ht) into heat
!cntr.status:='off'
!ac.status:='off'
!ht.status:='off'

!cntr.turnOn()
!therm.setThreshold(28,18)
!therm.sampleTemp(15)
!therm.sampleTemp(30)      OPR
!therm.sampleTemp(30)
!therm.sampleTemp(25)
!cntr.turnOff()
```

(c)

```
!therm:=new Thermostat
!cntr:=new Controller
!ac:=new AC
!ht:=new Heater
!insert (therm,cntr) into cntr
!insert (cntr, ac) into cool
!insert (cntr, ht) into heat
!cntr.status:='off'
!ac.status:='off'
!ht.status:='off'

!cntr.turnOn()
OCD
!therm.sampleTemp(15)
!therm.sampleTemp(25)
!therm.sampleTemp(30)
!therm.sampleTemp(25)
!cntr.turnOff()
```

Fig. 3 **a** An example test case t for HVAC, **b** and **c** examples of mutants for t

4.1 Generation of Mutants

The test case (denoted by t) used in the case study describes a scenario for checking if the model makes it possible to turn on and off the system, and to turn on and off the Heater and the AC, when the temperature rises and then falls down.

For the test case 51 mutants were generated. Detailed information on the number of mutants generated by applying each of the mutation operators is given in the second column of Table 1. Two of the mutants generated for t are shown in Fig. 3b, c.

4.2 Execution of Mutants

The mutants generated for test case t were executed within a USE environment [18]. The USE allows to run a model with a specified test case by simulating calls of the model operations (according to a scenario provided by the test case). While running the model, the tool checked if the pre- and post-conditions of called operations were

Table 1 Statistics by operators for mutants generated for t

Mutation operator	# of mutants for t ($\lvert T' \rvert$)	# of rejected mutants ($\lvert T'_R \rvert$)	# of accepted mutants ($\lvert T'_A \rvert$)	# of equivalent mutants ($\lvert T'_E \rvert$)
CPD	3	0	3	0
CPR	5	3	2	0
OCD	6	1	5	2
OCR	7	2	5	4
OCI	14	5	9	2
OCS	5	2	3	1
OPR	10	0	10	5
OPS	1	0	1	0
OTR	0	0	0	0

satisfied. An unexpected situation represented by a mutated test case was recognized if a violation of any condition was reported or the model failed. After executing a mutant, depending on the simulation results, a verdict rejected or accepted was assigned to it. Table 1 summarizes results of running the mutants. It gives, for each mutation operator, the number of accepted, rejected and equivalent mutants.

The results of executing mutants were presented within the USE in a form of a sequence diagrams [16]—these diagrams show the execution traces for the mutants.

4.3 Analysis of the Assessment Results

While most activities of the two previous stages of the model assessment can be automated, the analysis has to be performed manually.

First the accepted mutants were analyzed to identify and remove the equivalent mutants and the mutation score for the initial model was calculated as given by Eq. (1). Here, a mutation score of 0.35 was reached, what indicates rather low accuracy of the initial model and need for improvements.

To provide a feedback for the necessary improvements the execution traces of the remaining accepted mutants were analyzed to find out why they were accepted and what the model did wrong. For example, the mutant given in Fig. 3b was accepted. It represents a situation when the room temperature rises, in an instance, from below the lower threshold to above the upper threshold. While such situation seems unlikely to occur it is not entirely impossible. The examination of the mutant showed that the model turned on the AC, but it did not turn off the Heater. It is clearly an erroneous behavior of the model that needs to be fix, as both the Heater and the AC should never be working at the same time.

The rejected mutants were also analyzed to identify the operation that caused the model to fail. The mutant from Fig. 3c was rejected. It represents a situation when the user did not set the thresholds. The precondition defined for the regulate()

operation was violated, so the model was able to recognized erroneous situation but the model developer (or stakeholders) should still propose some solution that would allow the model to handle this situation.

Although the analysis performed in this stage are quite laborious, especially in case of accepted mutants, a careful inspection of the execution traces of mutants helps to gather information that in turn helps to improve the model, and thus the final systems that is to be developed basing on the model.

5 Conclusions and Future Work

A high quality system should always work flawlessly, thus it should never, in any situation, provide incorrect results and should be able to manage some unexpected situation without crashing, so early modeling of proper handling of such situations will significantly contribute to improvement of the quality of the final system. The approach presented in this paper contributes to the domain of developing models, as it provides a way to assess their accuracy with regard to their ability to recognized erroneous situations. This aspect of a model accuracy is rarely considered [13].

The approach uses mutation testing to generate, in a systematic and human-unbiased way, a number of test cases being able to trigger a wide range of unexpected situations. While mutation testing is a very effective assessment technique, it is also quite expensive in term of costs of generating and executing mutants [11]. Several cost reduction techniques were proposed, so far (a survey is given in [19]). A study on applying such techniques, especially the selective and structure dependant ones [20–24], in this context should also be a part of future work. Another problem that needs to be addressed is the identification of equivalent mutants [25]. It may also be worth to explore the possibility to use higher order mutations [11, 25] that could help to overcome both above problems.

In this paper the approach was studied in the context of models represented by means of UML and OCL. Still, the general approach to mutation testing based negative testing [5] may be easily adapted for other modeling formalisms or to other levels of system descriptions, as the general structure of a test cases will remain unchanged.

References

1. Brambilla, M., Cabot, J., Wimmer, M.: Model-Driven Software Engineering in Practice. Morgan & Claypool Publishers, San Rafael (2012)
2. Schamai, W., Helle, P., Fritzson, P., Christiaan, J., Paredis, J.: Virtual verification of system designs against system requirements. In: Models in Software Engineering. LNCS, vol. 6627, pp. 75–89. Springer, Heidelberg (2010)
3. Roman, A.: Testing and Software Quality. PWN, Warsaw (2015) (in polish)

4. Fernandez, J.-C., Mounier, L., Pachon, C.: A model-based approach for robustness testing. In: Testing of Communication Systems. LNCS, vol. 3502, pp. 333–348. Springer, Heidelberg (2005)
5. Strug, J.: Mutation testing approach to negative testing. J. Eng. **2016**, 13 p. (2016)
6. Strug, J.: Mutation testing approach to evaluation of design models. Key Eng. Mater. **572**, 543–546 (2014)
7. Briand, L.C., Labiche, Y., Shousha, M.: Stress testing real-time systems with genetic algorithms. In: Conference on Genetic and Evolutionary Computation, pp. 1021–1028, Washington, DC (2005)
8. Reid, S.C.: An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In: International Software Metrics Symposium, pp. 64–73, Albuquerque, NM (1997)
9. Zhang, J., Cheung, S.C.: Automated test case generation for the stress testing of multimedia systems. Softw. Pract. Exp. **32**, 1411–1435 (2002)
10. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: help for the practicing programmer. Computer **11**, 34–41 (1878)
11. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**, 649–678 (2011)
12. Bolazar, K., Fawcett, J.W.: Measuring component specification-implementation concordance with semantic mutation testing. In: International Conference on Computers and Their Applications, pp. 102–107, New Orleans (2011)
13. Fraser, G., Wotawa, F.: Using model-checkers for mutation-based test-case generation, coverage analysis and specification analysis. In: International Conference on Software Engineering Advances, pp. 16–21, Tahiti (2006)
14. Belli, F., Budnik, C.J., Hollmann, A., Tuglular, T., Wong, W.E.: Model-based mutation testing—approach and case studies. Sci. Comput. Program. **120**, 22–48 (2016)
15. Bahill, T., Daniels, J.: Using objected-oriented and UML tools for hardware design: a case study. Syst. Eng. **6**, 28–48 (2003)
16. Unified Modeling Language http://www.omg.org/spec/UML/2.5
17. Object Constraint Language http://www.omg.org/spec/OCL/2.4
18. Gogolla, M., Buttner, F., Richters, M.: USE: a UML-based specification environment for validating UML and OCL. Sci. Comput. Program. **69**, 27–34 (2007)
19. Usaola, M.P., Mateo, P.R.: Mutation testing cost reduction techniques: a survey. IEEE Softw. **27**, 80–86 (2010)
20. Strug, J.: Classification of mutation operators applied to design models. Key Eng. Mater. **572**, 539–542 (2014)
21. Ammann, P., Delamaro, M.E., Offutt, J.: Establishing theoretical minimal sets of mutants. In: IEEE International Conference on Software Testing, Verification and Validation, pp. 21–30, Cleveland Ohio, USA (2014)
22. Strug, J., Strug, B.: Machine learning approach in mutation testing. In: Software and Systems. LNCS, vol. 7641, pp. 200–214. Springer, Heidelberg (2012)
23. Strug, J., Strug, B.: Using structural similarity to classify tests in mutation testing. Appl. Mech. Mater. **378**, 546–551 (2013)
24. Strug, J., Strug, B.: Classifying mutants with decomposition kernel. In: Artificial Intelligence and Soft Computing. LNCS, vol. 9692, pp. 644–654. Springer, Heidelberg (2016)
25. Madeyski, L., Orzeszyna, W., Torkar, R., Józala, M.: Overcoming the equivalent mutant problem: a systematic literature review and a comparative experiment of second order mutation. IEEE Trans. Softw. Eng. **40**, 23–42 (2014)