Hindawi Publishing Corporation
Journal of Engineering
Volume 2016, Article ID 6589140, 13 pages
http://dx.doi.org/10.1155/2016/6589140

Research Article

Mutation Testing Approach to Negative Testing

Joanna Strug

Department of Electrical and Computer Engineering, Cracow University of Technology, Warszawska 24, 31-155 Krakow, Poland

Correspondence should be addressed to Joanna Strug; pestrug@cyf-kr.edu.pl

Received 29 November 2015; Accepted 8 June 2016

Academic Editor: Luis Carlos Rabelo

Negative testing deals with an important problem of assessing a system ability to handle unexpected situations. Such situations, if unhandled, may lead to system failures that in some cases can have catastrophic consequences. This paper presents a mutation testing-based approach for generation of test cases supporting negative testing. Application of this approach can provide, in a systematic and human-unbiased way, test cases effectively testing wide range of unexpected situations. Thus, it can contribute to improvement of a tested system. The paper formally defines mutation operators used to control the generation process, describes a generic framework for the generation and execution of the test cases, and explains how to interpret results.

1. Introduction

Testing plays an important role in developing dependable software systems working to the satisfaction of their users [1, 2]. A thorough testing should include both positive and negative testing. The positive testing [2, 3] comprises various activities related to checking if a system fulfills all requirements of stakeholders. A number of testing approaches offer ways to accomplish this task. Although the approaches vary in details, they usually share the concept of using some form of specification capturing the requirements for selecting test cases and then executing the test cases against the system to see if its responses match the expected and specified ones. Results of thoroughly performed positive testing can usually be considered a clear and dependable indication of the degree to which a system is correct in terms of fulfilling the specified requirements.

However, a system working correctly under normal conditions may still be unable to handle some untypical, unexpected, and basically undesired situations in an adequate way. Lack of adequate handling of such situations can cause the system to crash or to fail in providing proper outcomes, what in turn may lead to serious consequences, including damage in property or human life. Hence, there is need to perform also a negative testing [2–4]. The negative testing focuses on assessing the behaviour of a system while subjecting it to conditions out of its normal scope of operations. As a typical requirements specification states what a system is expected to do in certain situations rather than describing unexpected situations the system may encounter and their adequate handling, it cannot be used directly to support negative testing. Thus, it is usually left to the expertise and creativity of a tester to determine all the untypical and unexpected situations that may be of interest and provide test cases checking how the system will react to them.

In this paper, a mutation testing-based approach addressing the problem of providing negative test cases for testing software systems is presented. Originally, mutation testing was used to assess a quality of suites of positive test cases by checking their ability to detect faulty versions of a tested program (so-called mutants) generated from a source code of the program by changing it in a small ways [5, 6]. However, as it provides a general guidance for producing some "faulty" artifacts, it can also be applied for generating negative test cases by modifying the positive ones. Test cases obtained in this way can reflect a wide variety of unexpected usage scenarios for a system and thus contribute to better testing. The concept of mutating test cases rather than a program was outlined in [7] as a support for model evaluation. This paper builds on this concept, but it formalizes it and focuses on

object-oriented software systems. In particular, it describes formally rules (so-called mutation operators [5]) controlling the generation of negative test cases for such systems and outlines a generic framework for providing and using the test cases.

The paper is organized as follows. In Section 2, essential background information and related research concerning mutation testing and negative testing are presented. Section 3 gives the problems related to generation of negative test cases and briefly explains the reasons behind the proposed solution. Section 4 provides key information about positive and negative test cases and introduces an example system used in the subsequent section to illustrate the approach. In Section 5, mutation operators are formally defined and discussed, and Section 6 presents the generic framework for applying mutation testing to generate negative test cases and briefly discusses possible way of assessing a system within the framework. Section 7 states the main problem related to the approach and the last section concludes the work and indicates direction for future works.

2. Background and Related Research

The approach presented in this paper deals with applying mutation testing for generating negative tests; hence, these areas are of particular interest for the research.

2.1. Negative Testing. Negative testing aims at revealing weak points of a system, checking if it is able to handle unexpected situations adequately or at least recover gracefully in case of a failure [2–4]. There is no much work dedicated to the problem solely, but some general testing techniques provide advices on dealing with this problem. The most prominent example of such techniques is equivalence partitioning [2, 8, 9]. The idea behind this technique is to analyze the input domain for a system and partition it into several equivalence classes, including classes representing invalid values, and then select one value from each class to define test cases. Test cases defined based on invalid classes can be considered to be negative ones. Nevertheless, such approaches target only input values; thus, they deal only with one type of invalidating normal usage scenarios of a system. Another testing technique that can be related to negative testing directly is stress testing [10–12]. Stress testing focuses on checking how a system behaves under extreme conditions; thus, typical test cases provided to this aim try to overwhelm the system with resources or take the resources away. However, such approaches do not cover less extreme situation that may result from more subtle changes in using a system.

2.2. Mutation Testing. Mutation testing is a fault-based software testing technique introduced originally to assess quality of a suite of test cases with respect to their ability to detect faults in programs [5, 6]. Application of this technique involves generation of a number of faulty versions of the program, so that each of them differs from the original one by only one, small change. The faulty versions, called mutants, are then run with the assessed suite of test cases. Once all test cases are executed, the so-called mutation score for the suite is calculated as a ratio of mutants detected by the test cases over the total number of nonequivalent [13] mutants generated from the original program. The mutation score expresses, in a quantitative way, the quality of the suite of test cases being assessed.

Nowadays, the application area of mutation testing is not limited to assessing test cases at implementation level only. A number of works have shown examples of applying it to different formalisms, at various levels of software development, and for assessment, as well as for generation of test cases [14–24]. Works dedicated to mutation testing-based generation of test cases usually share the idea of mutating a system, its model, or specification and then selecting test cases being able to detect the mutants [15, 18, 19, 23–26]. Different methods are used to select the test cases based on the mutants. In the research presented in [18], constraint solving was used to derive positive test cases from mutated constraints imposed on functions. A conversion of a graph-based model to a formal grammar-based description was proposed in [26]. The grammar was further processed to obtain test cases. An interesting approach was presented also in [25]. Its authors applied mutations to a model of a system or to specified properties and used model-checking technique to generate counterexamples showing violation of certain, desired, or mutated properties. In both cases, the counterexamples were seen as test cases, either negative or positive. As it seems, such approach can provide both positive and negative test cases. However, as mutants can behave in ways incorrect with respect to a system specification or in entirely unpredictable way, some of negative test cases may actually detect specification related inconsistencies that can be detected by positive test cases.

The concept of providing negative test cases by using mutation testing on test cases directly is not widely explored. To the author's best knowledge, so far, only a few other researchers have followed the idea. The approach described in [27] is the closest to the approach outlined in [7] and the one further presented in this paper. The approach in [27] deals with measuring a specification-implementation concordance, but it also aims at modifying test cases directly. However, the changes introduced by its authors applied only to data processed by a program and were random. This approach provides wider range of changes introduced in a controlled way.

Several studies have demonstrated that mutation testing is an effective and trustworthy technique supporting test generation and assessment at various levels of a software development [28–31]. The strength of mutation testing lies in its systematic and human-unbiased way of generating mutants; thus, it seems to be well suited to accomplish also the task of providing negative test cases being able to detect various undesired situations.

3. Problems and Solution

The main purpose of negative testing is to check whether a tested system is able to handle properly any unexpected and undesired situations. Thus, selection of negative test cases serving this purpose poses a challenge. In particular, two

issues need to be resolved. The first one is to decide what situations are unexpected and how to trigger them, and the second issue is to determine what their proper handling should be.

Two assumptions help to set basis for dealing with the above issues:

(1) Any situation that is not given by a requirement specification is accounted as an unexpected and undesired one.

(2) Proper system response to any of these situations is to follow an error handling procedure.

In context of negative testing, both assumptions are justified, because a system should only work and be used, in ways described by the specification, and it should not do anything else. An attempt to use the system in any other way should be considered incorrect and be forbidden.

Taking into account the first assumption, it can be further assumed that situations of interest for negative testing can be defined upon the specified ones by contradicting expected ways of using a system. The contradiction can be achieved by invalidating either inputs triggering the expected behaviours of the system or the conditions that are expected to hold for it.

A typical test case defines conditions and inputs needed to force the tested system to behave in a given way and usually expected outcomes the system should produce [2]. Thus, to define negative test cases, one needs to select representative combinations of invalid conditions and inputs. Accurate outcomes for negative test cases cannot be determined. However, the second assumption helps to decide if actual outcomes resulting from executing a negative test case indicate whether the system is able to prevent unwanted behaviour or not.

A majority of unexpected situations are caused by slightly incorrect use (by human users or external systems) of a system; therefore, a suite including negative test cases varying only slightly from the positive ones should be adequate for triggering wide range of unexpected situations. The approach presented in this paper proposes to use mutation testing to generate such suite. Application of this technique supports systematic and human-unbiased way of introducing small modification of various kinds.

4. Positive and Negative Test Cases for Object-Oriented Systems

The approach presented in this paper aims at object-oriented software systems. An object-oriented system consists of interacting objects, but from a user's perspective, such system can be seen as a single entity (an object) having its own properties (attributes) and providing specific services (operations) to the users. A simple model of ATM cash machine, used in this paper to illustrate the approach, is shown in Box 1. The system has been presented in a form of a class defining the attributes and operations. For the attributes and operations, short descriptions regarding their meaning were included.

To use the system, the user is required to follow certain scenarios. For example, to successfully withdraw money, one needs to insert a card, to enter a valid PIN, to select the withdrawal option, to enter correct amount of money to withdraw, to confirm it, then to take the disposed money, assuming that the requested amount is currently available, and to take the ejected card. So, to test if the system is able to perform according to such a scenario, a positive test case reflecting the scenario is required. Hence, a positive test case should describe the events triggering certain system operations (i.e., inputs) and expected system responses (i.e., outputs) and initial settings ensuring that the scenario can be successfully accomplished (i.e., conditions). Formally, a test case for an object-oriented system is defined as follows (Definition 1).

Definition 1. A test case t_j for object-oriented system S is a triple (C_j, I_j, and O_j) where

(i) C_j is a vector $(c_{j1}c_{j2}, \ldots, c_{|C|})$ of conditions defining the state of S where

(a) $c_{ji} = \{\text{obj}, S(X_{ji}, Y_{ji})\}$ is a call of a constructor instantiating and initializing an object representing the system S where

(1) obj is an instantiated object of S,

(2) $S(X_{ji}, Y_{ji})$ is the constructor defined for S, and $X_{ji} = (x_{ji1}, x_{ji2}, \ldots, x_{ji|X|})$ and $Y_{ji} = (y_{ji1}, y_{ji2}, \ldots, y_{ji|Y|})$ are sequences of values and types, respectively, of the same length (e.g., $|X| = |Y|$),

(b) $|C|$ is the length of C_j,

(ii) I_j is vector $(i_{j1}, i_{j2}, \ldots, i_{|I|})$ of inputs, where

(a) $i_{ji} = \{\text{obj}, f_k(X_{ji}, Y_{ji})\}$ is a single input triggering some behaviour of S, where

(1) obj is an object of S targeted by $f_k(X_{ji}, Y_{ji})$,

(2) $f_k(X_{ji}, Y_{ji})$ is a call of method $f_k \in F_S$ (a suite of operations defined for S), and $X_{ji} = (x_{ji1}, x_{ji2}, \ldots, x_{ji|X|})$ and $Y_{ji} = (y_{ji1}, y_{ji2}, \ldots, y_{ji|Y|})$ are sequences of values and types, respectively, of the same length (e.g., $|X| = |Y|$),

(b) $|I|$ is the length of I_j,

(iii) O_j is vector $(o_{j1}, o_{j2}, \ldots, o_{|O|})$ of outputs produced by S where

(a) o_{ji} is an output provided for i_{ji} (o_{ji} may by empty, denoted as 0, when no output is expected for i_{ji}),

(b) $|O|$ is the length of O_j, and $|O| = |I|$.

An example of a positive test case representing the scenario for successful withdrawal of money, as generally described earlier in this section, is shown in Box 2. While the format of the test case in Box 2 does not imply the use of a specific formalism, it follows the object-oriented principles and it can be easily adapted for any object-oriented approach.

As given by Definition 1, the test case from Box 2 is denoted as $t_1 = (C_1, I_1, O_1)$, where

```
ATM

mode: int                                   // defines whether the ATM is in use
currentAmount: double                       // defines the available amount of money in ATM

getCard():Boolean                           // requests the card be inserted
getPin(pin: int): int                       // requires a PIN be inserted
validateCard(cardNo: int, pin: int):Boolean // validates the card and the PIN
displayMenuChoice():void                    // displays the possible actions available for a user
enterMenuChoice(choice: string):Boolean     // requests an action be selected
checkBalance(amount: double):Boolean        // checks if the user has the requested amount
enterAmount(amount: double):double          // requests an amount to withdrawn be entered
confirm():Boolean                           // passes confirmation of withdrawal
quit():Boolean                              // passes quitting of withdrawal
dispenceCash():double                       // pays out the money
ejectCard():string                          // returns the card
```

Box 1: A model of an ATM system.

```
atm = new ATM(1, 100000.00);

atm.getCard();
atm.getPIN(1234);
atm.displayMenuChoice() / "check balance, withdraw cash, quit";
atm.getMenuChoice("withdraw cash");
atm.enterAmount(500);
atm.confirm();
atm.ejectCard() / "card ejected";
atm.dispenceCash() / 500;
```

Box 2: An example of a positive test case for ATM system.

(i) $C_1 = (c_1)$, such that $c_1 = \{\text{obj}, \text{ATM}(X_{11}, Y_{11})\}$, where obj = atm, S = ATM, $X_{11} = (x_{111}, x_{112}) = (1, 100000.00)$, and $Y_{11} = (y_{111}, y_{112}) = (\text{string}, \text{int})$,

(ii) $I_1 = (i_{11}, i_{12}, \ldots, i_{18})$ such that

(a) $i_{11} = \{\text{obj}, f_1(X_{11}, Y_{11})\}$, where obj = atm, f_1 = getCard, and X_{11} and Y_{11} are empty,

(b) $i_{12} = \{\text{obj}, f_2(X_{12}, Y_{12})\}$, where obj = atm, f_2 = getPIN, $X_{12} = (x_{121}) = (1234)$, and $Y_{12} = (y_{121}) = (\text{int})$,

(c) The remaining inputs are defined in the same way as shown in (a) and (b) above.

$O_1 = (o_{11}, o_{12}, \ldots, o_{18})$ such that o_4 is "check balance, withdraw cash, and quit," o_7 = "card ejected," $o_8 = 500$, and the remaining outputs are empty.

However, positive test cases can check only scenarios that are defined by specification. A small user's mistake or lack of resources needed to successfully run a certain scenario may cause the system to behave in an unknown and most likely unacceptable way, to crash or to hang. Some typical deviations from a given scenario, such as entering an invalid PIN or unavailability of requested amount of money while running the withdrawal scenario, are usually defined in a requirements specification; thus, they are well defined and not unexpected and therefore should be tested by positive test cases.

However, numerous other situations being the results of a small deviation from some known scenarios could be truly unexpected and hard to predict. Although such situation may occur rarely, a high quality system should be prepared to handle them by following some error handling procedure. To check a system "readiness" to handle the unexpected situation, negative test cases have to be used. As it was stated in the previous section, a negative test case can be obtained by changing a part of a positive test case. For example, to check what will happen when a user does not confirm the withdrawal after entering the amount, a negative test case should be obtained by simply removing the call for the confirmation from a positive test case defined for the withdrawal scenario. A suite of negative test cases, generated by means of mutation testing, can check a system against a wide range of unexpected situations that even an experienced test developer would not be able to design.

A negative test case has the same structure as a positive one, so it will not be defined separately. Examples of negative test cases are given in Section 5.

```
atm = new ATM(1, 100000.00);

atm.getCard();
atm.getPIN(1234);
atm.displayMenuChoice() / “check balance, withdraw cash, quit”;
atm.getMenuChoice(“withdraw cash”);
atm.enterAmount(500);
atm.ejectCard() / “card ejected”;
atm.dispenceCash() / 500;
```

Box 3: An example of a mutant generated by applying operator op_{OCD}.

```
atm = new ATM(1, 100000.00);

atm.getCard();
atm.getPIN(1234);
atm.displayMenuChoice() / “check balance, withdraw cash, quit”;
atm.getMenuChoice(“withdraw cash”);
atm.enterAmount(500);
atm.quit();
atm.ejectCard() / “card ejected”;
atm.dispenceCash() / 500;
```

Box 4: An example of a mutant generated by applying operator op_{OCR}.

5. Mutation Operators

A mutation operator is a transformation rule that defines how to modify certain features of the artifact undergoing mutations [5]. There is a subset of the so-called traditional mutation operators that are fairly universal and can be easily adapted for different programming or modeling languages, but in general mutation operators are formalism-specific [7, 14, 16, 19, 32–34]. Therefore, application of mutation testing in different context, as in this approach, should always include defining an adequate suite of mutation operators.

An adequate suite of mutation operators should at least

(i) cover all features of the targeted formalism,

(ii) generate syntactically correct (feasible) mutants.

The syntax of test cases is rather simple, when comparing to the syntax of programming languages, but it is also significantly different from it. Therefore, the mutation operators defined for programming languages are not applicable in context of mutating test cases. A suite of 9 mutation operators targeting test cases was introduced and informally described in [7]. One, new mutation operator (Condition Part Swap) is here introduced, and the operator Operation Target Replacement mentioned in [7] was here discarded, because of the fact that this approach assumed only one object representing the system to be created, so the operator will be of no use. All the operators target conditions and inputs. Their formal definitions are given by Definitions 2–10.

Let op_{tp} be a mutation operator of type tp and let t be a test case, such that $t = (C, I, O)$ as given by Definition 1. Formally, for a given test case t, a mutation operator op_{tp} produces a mutated test case t^M what is denoted by $\text{op}_{\text{tp}}: t \to t^M$, where $t^M = (C^M, I^M, O^M)$ is a mutated test case.

Definition 2. Operation Call Deletion operator (op_{OCD}) is a mutation operator that deletes one input from the sequence of inputs given by t, what is formally defined: $\text{op}_{\text{OCD}}: t \to t^M$, where $t^M = (C^M, I^M, O^M)$, $C^M = C$, $I^M \neq I$, $O^M = O$, and $I^M = (i_1, i_2, \ldots, i_{i-1}, i_{i+1}, \ldots, i_{|I|})$; that is, I^M is obtained by removing i from I.

An example of mutated test cases obtained by applying this operator to the test case given in Box 2 is shown in Box 3. The mutant was generated by removing the call for operation `confirm()`. The mutants in Box 3 should be able to check what would happen when a user does not confirm entering the amount of money to withdraw.

Definition 3. Operation Call Replacement operator (op_{OCR}) is a mutation operator that replaces one input in the sequence of inputs given by t by another input, what is formally defined: $\text{op}_{\text{OCR}}: t \to t^M$, where $t^M = (C^M, I^M, O^M)$, $C^M = C$, $I^M \neq I$, $O^M = O$, and $I^M = (i_1, i_2, \ldots, i_{i-1}, i_i', i_{i+1}, \ldots, i_{|I|})$ and $i_i' \in F_S$; that is, I^M is obtained by replacing i_i with i_i' which is also a call of operation of S.

An example of mutated test cases obtained by applying this operator to the test case given in Box 2 is shown in Box 4. The mutant was generated by replacing operation `confirm()` with operation `quit()`. Such mutated test case should be able to check how the system will further process if different activity was requested (especially if it will continue

```
atm = new ATM(1, 100000.00);

atm.getCard();
atm.getPIN(1234);
atm.displayMenuChoice() / “check balance, withdraw cash, quit”;
atm.getMenuChoice(“withdraw cash”);
atm.enterAmount(500);
atm.confirm();
atm.quit();
atm.ejectCard() / “card ejected”;
atm.dispenceCash() / 500;
```

Box 5: An example of a mutant generated by applying operator op_{OCI}.

```
atm = new ATM(1, 100000.00);

atm.getCard();
atm.getPIN(1234);
atm.displayMenuChoice() / “check balance, withdraw cash, quit”;
atm.getMenuChoice(“withdraw cash”);
atm.confirm();
atm.enterAmount(500);
atm.ejectCard() / “card ejected”;
atm.dispenceCash() / 500;
```

Box 6: An example of a mutant generated by applying operator op_{OCS}.

withdrawal scenario or provide outputs expected for scenario with aborted withdrawal).

Definition 4. Operation Call Insertion operator (op_{OCI}) is a mutation operator that inserts additional input into the sequence of inputs given by t, what is formally defined: op_{OCI}: $t \to t^M$, where $t^M = (C^M, I^M, O^M)$, $C^M = C$, $I^M \neq I$, $O^M = O$, and $I^M = (i_1, i_2, \ldots, i_i, i_k, i_{i+1}, \ldots, i_{|I|})$ and $i_k \in F_S$; that is, I^M is obtained by inserting into it i_k which is kth operation of S.

An example of mutated test cases obtained by applying this operator to the test case given in Box 2 is shown in Box 5. The mutant was generated by adding the call for operation `quit()` after the call for operation `confirm()`. Such mutated test case should be able to check what would happed when a user tries to abort the withdrawal of money after confirming entering of the amount.

Definition 5. Operation Call Swap operator (op_{OCS}) is a mutation operator that changes the order of two subsequent inputs of the sequence of inputs given by t, what is formally defined: op_{OCS}: $t \to t^M$, where $t^M = (C^M, I^M, O^M)$, $C^M = C$, $I^M \neq I$, $O^M = O$, and $I^M = (i_1, i_2, \ldots, i_{i-1}, i_{i+1}, i_i, \ldots, i_{|I|})$; that is, I^M is obtained by swapping inputs i_i and i_{i+1}.

An example of mutated test cases obtained by applying this operator to the test case given in Box 2 is shown in Box 6. The mutant was generated by swapping calls for operations `enterAmount(500)` and `confirm()`. Such mutated test case should be able to check what would happen when users do not enter the amount of money to withdraw before confirming it and then they will try to enter the amount later (especially if the system will continue scenario for withdrawal).

Definition 6. Operation Parameter Replacement operator (op_{OPR}) is a mutation operator that replaces a value of one parameter of an operation call with another value of the same type, what is formally defined: op_{OPR}: $t \to t^M$, where $t^M = (C^M, I^M, O^M)$, $C^M = C$, $I^M \neq I$, $O^M = O$, and $I^M = (i_1, i_2, \ldots, i'_i, \ldots, i_{|I|})$, where $i'_i = \{\text{obj}, f_k(X'_i, Y_i)\}$ and $X'_i = (x_1, x_2, \ldots, x'_i, \ldots, x_{|X|})$ and x'_i is a value of type y_i.

An example of a mutated test case obtained by applying this operator to the test case given in Box 2 is shown in Box 7. The mutant was generated replacing the value 500 of the parameter of call for operation `enterAmount()` with value 0. Such mutated test case should be able to check what the system will do when a user does not enter a valid value.

Definition 7. Operation Parameter Swap operator (op_{OPS}) is a mutation operator that changes the order of two values of parameters of an operation call, what is formally defined: op_{OPS}: $t \to t^M$, where $t^M = (C^M, I^M, O^M)$, $C^M = C$, $I^M \neq I$, $O^M = O$, and $I^M = (i_1, i_2, \ldots, i'_i, \ldots, i_{|I|})$, where $i'_i = \{\text{obj}, f_k(X'_i, Y_i)\}$ and $X'_i = (x_1, x_2, \ldots, x_{i-1}, x_k, x_{i+1}, \ldots, x_{k-1}, x_i, x_{k+1}, \ldots, x_{|X|})$ for $k \neq i$ and $y_k = y_i$.

```
atm = new ATM(1, 100000.00);

atm.getCard();
atm.getPIN(1234);
atm.displayMenuChoice() / "check balance, withdraw cash, quit";
atm.getMenuChoice("withdraw cash");
atm.enterAmount(0);
atm.confirm();
atm.ejectCard() / "card ejected";
atm.dispenceCash() / 500;
```

Box 7: An example of a mutant generated by applying operator op_{OPR}.

The principles behind applying this operator are the same as those for Condition Part Swap operator given by Definition 10. The example test case, used here to illustrate application of mutation operator, will not undergo modifications defined by op_{OPS}, but the example in Box 10 can be referred to to see the idea of swapping parameters.

Definition 8. Condition Part Deletion operator (op_{CPD}) is a mutation operator that deletes one element of a condition in the condition sequence, what is formally defined: op_{CPD}: $t \to t^M$, where $t^M = (C^M, I^M, O^M)$, $C^M \neq C$, $I^M = I$, $O^M = O$, and $C^M = (c_1, c_2, \ldots, c_i', \ldots, c_{|I|})$, where $c_i' = \{\text{obj}, S(X_i', Y_i')\}$ and $X_i' = (x_1, x_2, \ldots, x_{i-1}, x_{i+1}, \ldots, x_{|X|})$ and $Y_i' = (y_1, y_2, \ldots, y_{i-1}, y_{i+1}, \ldots, y_{|X|})$.

An example of mutated test cases obtained by applying this operator to the test case given in Box 2 is shown in Box 8. The mutant was generated by removing one value in the call of constructor `ATM(1, 100000.00)`. Deletion of a value reflects the concept of a missing suiting defining a state of the system that should be met to run some scenario. Such mutant should be able to check how the scenario for withdrawal of money will run when the amount of money available is not set.

Definition 9. Condition Part Replacement operator (op_{CPR}) is a mutation operator that replaces a value of one parameter of a constructor call with another value of the same type, what is formally defined: op_{OCR}: $t \to t^M$, where $t^M = (C^M, I^M, O^M)$, $C^M \neq C$, $I^M = I$, $O^M = O$, and $C^M = (c_1, c_2, \ldots, c_i', \ldots, c_{|I|})$, where $c_i' = \{\text{obj}, S(X_i', Y_i)\}$ and $X_i' = (x_1, x_2, \ldots, x_i', \ldots, x_{|X|})$ and x_i' is a value of type y_i.

An example of mutated test cases obtained by applying this operator to the test case given in Box 2 is shown in Box 9. The mutant was generated by replacing the value 1000000.00 in the constructor `ATM(1, 100000.00)` with 0. Such mutant should be able to check how the scenario for withdrawal of money will run when there is no money available in the ATM.

Definition 10. Condition Part Swap operator (op_{CPS}) is a mutation operator that changes the order of two parameters values of a constructor call, what is formally defined: op_{OCR}: $t \to t^M$, where $t^M = (C^M, I^M, O^M)$, $C^M \neq C$, $I^M = I$, $O^M = O$, and $C^M = (c_1, c_2, \ldots, c_i', \ldots, c_{|I|})$, where $c_i' = \{\text{obj}, S(X_i', Y_i')\}$ and $X_i' = (x_1, x_2, \ldots, x_{i-1}, x_k, x_{i+1}, \ldots, x_{k-1}, x_i, x_{k+1}, \ldots, x_{|X|})$ for $k \neq i$ and $y_k = y_i$.

An example of mutated test cases obtained by applying this operator to the test case given in Box 2 is shown in Box 10. The mutant was generated by swapping the values in the constructor `ATM(1, 100000.00)`. Such mutant should be able to check what will happen when certain conditions setting the correct state of ATM do not hold.

Negative test cases generated by applying operators op_{OPR} or op_{CPR} may have a counterpart in test cases obtained by using methods based on equivalence partitioning. Other kinds of mutants rather do not have their equivalents in test cases obtained by means of other test generation methods.

The examples in Boxes 3–10 show that some mutants generated by the mutation operators reflect unexpected, but still viable, scenarios, while others seem to represent quite unrealistic scenarios. It is a general characteristic of mutation testing; mutants rarely represent errors that can happen in reality. However, it was shown by several researchers (see references in [13]) that mutated programs and models imitated real live errors good enough to be considered reliable base of assessment of test cases. Thus, similar tendency can be expected in this context. These seemingly impossible to occur scenarios that are represented by mutated test cases may be able to detect serious problems that will never be in focus of a test designer, even an experienced one.

The suite of mutation operators covers all features of a typical test case that are of interest in context of negative testing. Thus, the suite should be sufficient to provide negative test cases being able to trigger wide range of unexpected situation.

6. A Generic Framework for Generating Negative Test Cases

An outline of the generic framework for generating and using negative test cases is presented in Figure 1. The main steps performed within this framework follow the general principles of applying mutation testing: generation of mutants (i.e., the negative test cases) and their execution.

Let us for the rest of this section suppose that ${}^P T = \{{}^P t_i : 1 \leq i \leq |{}^P T|\}$ denotes a suite of $|{}^P T|$ positive test

```
atm = new ATM(1);

atm.getCard();
atm.getPIN(1234);
atm.displayMenuChoice() / “check balance, withdraw cash, quit”;
atm.getMenuChoice(“withdraw cash”);
atm.enterAmount(500);
atm.confirm();
atm.ejectCard() / “card ejected”;
atm.dispenceCash() / 500;
```

Box 8: An example of a mutant generated by applying operator op_{CPD}.

```
atm = new ATM(1, 0);

atm.getCard();
atm.getPIN(1234);
atm.displayMenuChoice() / “check balance, withdraw cash, quit”;
atm.getMenuChoice(“withdraw cash”);
atm.enterAmount(500);
atm.confirm();
atm.ejectCard() / “card ejected”;
atm.dispenceCash() / 500;
```

Box 9: An example of a mutant generated by applying operator op_{CPR}.

cases provided for a tested system S, ${}^{N}T = \{{}^{N}t_j : 1 \leq j \leq |{}^{N}T|\}$ denotes a suite of $|{}^{N}T|$ negative test cases, where ${}^{N}t_j$ is a negative test case generated for a positive test case by applying a mutation operator, and $\text{OP} = \{\text{op}_{\text{tp}} : \text{tp} = \{\text{OCD, OCR, OCI, OCS, OPR, OPS, CPD, CPR, CPS}\}\}$ denotes a suite of mutation operators defined for test cases.

6.1. Generation of Mutants. The first step, generation of mutated (negative) test cases, requires the suite of positive test cases (${}^{P}T$) to be provided. It is further required that

(i) the suite ${}^{P}T$ is complete with respect to a requirements specification for system S,

(ii) all test cases from ${}^{P}T$ were executed against S and S has passed them all.

The expected outcome of this step is a suite of negative test cases (${}^{N}T$). A generic procedure for generating mutants from a given suite of positive test cases is given in Box 11. It gives the key steps of generating negative test cases, but details concerning their actual implementation will depend on the actual implementation of the theoretical model.

To generate the suite ${}^{N}T$ of negative test cases, each positive test case (${}^{P}t_i$) has to be parsed to recognize and return, one by one, all occurrences of conditions and inputs, the elements that are to be modified by applying adequate mutation operators to them. So, each call of the operation *nextElement()* returns one element (denoted in Box 11 by `e.el`) and its type (denoted in Box 11 by `e.tp`). Depending on the type `e.tp` returned for an element `e.el`, some of the “apply an operator” operations are called. Independent of the specific functionality of the “apply an operator” operation, each of them takes as arguments the element `e.el` returned by current call to nextElement() and currently analyzed positive test case ${}^{P}t_i$ and returns a subset of newly generated negative test cases. A generation of one negative test case always consists in copying ${}^{P}t_i$ and modifying the element `e.el` of the copy according to the change defined by the given “apply an operator” operation. The cost, in terms of computational complexity, of generating one negative test case is a sum of the costs of copying the positive test case ($O(|C| + |I| + |O|)$ (the notations used to express the complexity of all operations in Box 11 are the same as those in Definition 1)) and of introducing one modification ($O(1)$). Considering the fact that $|O|$ is comparable to $|I|$ and $|C|$ should not be greater than $|I|$, the cost of generating one negative test case can be approximated by $O(|I|)$.

There are nine “apply an operator” operations, one for each mutation operator. The cost of one call of a given operation depends on the number of negative test cases it generates and returns. The operations are listed as follows, briefly described, and for each of them the cost of its one call is given:

(i) `applyOCD()` takes, as the `e.el` argument, a system operation call and returns one negative test case obtained by deleting the operation call (e.g., by omitting it while copying the test case ${}^{P}t_i$ given as its second argument into one, new, negative test case).

```
atm = new ATM(100000.00, 1);

atm.getCard();
atm.getPIN(1234);
atm.displayMenuChoice() / "check balance, withdraw cash, quit";
atm.getMenuChoice("withdraw cash");
atm.enterAmount(500);
atm.confirm();
atm.ejectCard() / "card ejected";
atm.dispenceCash() / 500;
```

Box 10: An example of a mutant generated by applying operator op_{CPS}.

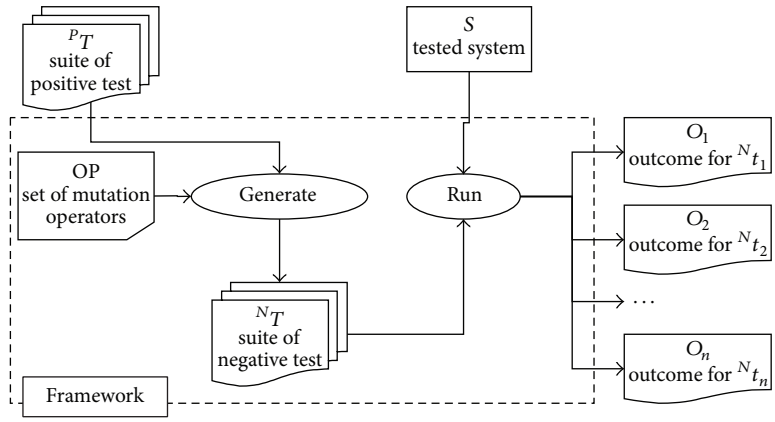

Figure 1: An outline of the framework.

```
INPUT: PT
OUTPUT: NT
NT = ∅
for each Pt_i ∈ PT do
    for each e = nextElement(Pt_i) do
        if (e.tp = "condition") then
            NT = NT ∪ applyCPD(e.el, Pt_i);
            NT = NT ∪ applyCPR(e.el, Pt_i);
            NT = NT ∪ applyCPS(e.el, Pt_i);
        else if (e.tp = "input") then
            NT = NT ∪ applyOCD(e.el, Pt_i);
            NT = NT ∪ applyOCR(e.el, Pt_i);
            NT = NT ∪ applyOCI(e.el, Pt_i);
            NT = NT ∪ applyOCS(e.el, Pt_i);
            NT = NT ∪ applyOPR(e.el, Pt_i);
            NT = NT ∪ applyOPS(e.el, Pt_i);
```

Box 11: An outline of mutants generation procedure.

The cost of one call of `applyOCD()` is equal to the cost of generating one negative test case – $O(|I|)$.

(ii) `applyOCR()` takes, as the `e.el` argument, a system operation call and returns a subset of negative test cases. Each negative test case is obtained by replacing the received call to the system operation f_k by a call to a system operation f_l ($1 \leq l \leq |F_S| \wedge l \neq k$). The number of negative test cases generated in this way equals $|F_S| - 1$; hence, the cost of one call of `applyOCR()` is $O(|I| * |F_S|)$.

(iii) `applyOCI()` takes, as the `e.el` argument, a system operation call and returns a subset of negative test cases. Each negative test case is obtained by inserting a call to a system operation f_l ($1 \leq l \leq |F_S|$) after the received call to the system operation f_k. The number of negative test cases generated in this way equals again $|F_S| - 1$, for all but the first call of `applyOCI()`. In the first call of `applyOCI()`, that is for the first input of $^{P}t_i$, the calls to the remaining system operations are inserted also before the call to f_j. The cost of one call of `applyOCI()` is $O(|I| * |F_S|)$.

(iv) `applyOCS()` takes, as the `e.el` argument, a system operation call and returns one negative test case obtained by reversing the order of two subsequent system operations calls given by $^{P}t_i$: the received call to system operation f_k and the subsequent call to a system operation f_l. The cost of one call of `applyOCS()` is $O(|I|)$.

(v) `applyOPR()` takes, as the `e.el` argument, a system operation call and returns a subset of negative test cases. Each negative test case is obtained by replacing the value x_i passed in the received call to operation f_k

with a value x'_i from a set $x' = \{x'_i: 1 \leq i \leq |x|\}$ of predefined values of type y_i. The number of negative test cases generated by replacing the value x_i with all the values defined for it equals $|x|$; thus, the cost of replacing all values given by the received call to system operation f_k equals $|X| * |x|$, and the cost of one call of `applyOPR()` is $O(|I| * |X| * |x|)$.

(vi) `applyOPS()` takes, as the `e.el` argument, a system operation call and returns a subset of negative test cases. Each test case is obtained by swapping two values x_i and x_l passed in the received call to a system operation f_k. The number of negative test cases generated in this way equals the number of swaps that have to be done; that is, $(|X| * (|X| - 1))/2$ for $|X|$ values passed in the operation call. So, the cost of one call of `applyOPR()` is $O(|I| * |X|^2)$.

(vii) `applyCPD()` takes, as the `e.el` argument, a condition and returns a subset of negative test cases. Each negative test case is obtained by removing a value x_i from the list of values passed in the received call to the system constructor S. The number of negative test cases generated in this way equals the number of values passed in the call; thus, the cost of one call of `applyCPD()` is $O(|I| * |X|)$.

(viii) `applyCPR()` takes, as the `e.el` argument, a condition and returns a subset of negative test cases. The negative test cases are generated basically in the same way as that by the `applyOPR()` operation and the cost of one call of `applyCPR()` is also $O(|I| * |X| * |x|)$.

(ix) `applyCPS()` takes, as the `e.el` argument, a condition and returns a subset of negative test cases. The negative test cases are generated basically in the same way as that by the `applyOPS()` operation and the cost of one call of `applyCPS()` is also $O(|I| * |X|^2)$.

The cost of mutating one positive test case is proportional to the number of negative test cases generated on its base and is $O(|I|^2 * (|F_S| + |X|^2 + |X| * |x| + |X|))$ and the total cost of applying the procedure in Box 11 is $O(|{}^P T| * |I|^2 * (|F_S| + |X|^2 + |X| * |x| + |X|))$.

6.2. Execution of Mutants. The second step uses the negative test cases (${}^N T$) generated in the previous step and additionally requires the system S to be provided. The system should be correct with respect to its specification; that is, it should have passed all positive test cases from ${}^P T$, as it was stated in the previous subsection. The purpose of this step is to gather information helping to assess the system ability to handle unexpected situations in a safe way. A generic procedure for executing mutated test cases follows a typical test execution procedure. Its outline is presented in Box 12.

Each iteration of this procedure consists in resetting the system S to its initial state and then running the system (e.g., a program or an executable model) while feeding it with inputs from one negative test case ${}^N t_i$ and saving a corresponding outcome OT_i. An outcome OT_i should include actual outputs produced by the system for inputs given by test case ${}^N t_i$ and

INPUT: ${}^N T$, S
OUTPUT: OT

for each ${}^N t_i \in {}^N T$ **do**
  reset(S)
  OT_i = run(S, ${}^N t_i$)
$\mathrm{OT} = \mathrm{OT} \cup \mathrm{OT}_j$

Box 12: An outline of mutants execution procedure.

a verdict linked to the test case (rejected or accepted). The cost of executing the negative test cases depends on their number and size ($O(|{}^N T| * |I|)$) and on the cost of running the system (the cost of running a system cannot be given here as it depends on the system itself).

A more detailed procedure for executing negative test cases and an actual format of outcomes are here not defined, as they will depend on a testing environment used to run the system and execute the negative test cases.

6.3. Analysis of Negative Testing Results. After finishing both steps performed within the framework, the outcomes have to be analyzed to draw some conclusions regarding the ability of the assessed system to detect and adequately handle unexpected situations.

When mutation testing is applied in order to assess quality of a suite of test cases, the assessment is simple based on calculating the number of detected (i.e., providing outputs different from the original program for at least one test case from the suite) and undetected (i.e., providing the same outputs as the original program for all test cases from the suite) mutants. Only the undetected mutants need to be further manually analyzed to decide if they are equivalent [6] or were not detected due to the insufficiency of the suite.

In context of applying mutation testing to assess a system, the conclusions cannot be drawn in such a straightforward way. All results of running a system with the mutated test cases have to be analyzed and most of the work needs to be performed manually. The system may respond to a mutated test cases by crashing or hanging or by running the unexpected scenario without breaking and providing erroneous outcomes.

A crash or hanging is here an obvious case of "detecting" (rejecting) a mutated test case. However, in this context, it indicates the system insufficiency in providing any handling of the unexpected situation checked by the test case. Therefore, it should be recommended to study the test cases and execution traces of the system to identify the problem behind the crash and to propose adequate means of fixing the problem.

When a system does not crash, while executing particular mutated test case, the test case is considered "undetected" (accepted). Acceptance of a mutated test case means that the system has provided some error handling, it was unable to recognize unexpected situation, and it actually ran the scenario given by the mutant and provided some (erroneous)

outputs or that the mutant was an equivalent one. Each case of accepting a mutant should be carefully analyzed to see what has caused the acceptance.

Let us consider the mutated test case presented in Box 2 as example. When the test case is executed, the ATM can wait for some period of time and then either abort the withdrawal and eject or withhold the card or first issue a message pointing to the lack of the expected input before aborting the withdrawal if the input is still not provided. Both responses seem to be acceptable ways of handling this unexpected situation; thus, they could be considered adequate ways of error handling. The system may also crash (and switch off) or hang (and be unable to abort the withdrawal and undertake any other actions) pointing to its inability to deal with the situation. Another course of action that may be taken by the system could consist in the continuation of the scenario and disposal of some money, what should be considered to be the case of not recognizing the situation as an unexpected one and providing invalid outcomes. In general, cases like the last one could be dangerous, because they may deceive the system users into thinking that the outcomes they obtained are correct and use them as such.

A mutated test case should be considered to be equivalent if it reflects a scenario represented by some of the positive test cases and forces the system to work in a way expected for the positive scenario. It is however not required that the outcomes will be identical with the outcomes provided by executing the positive test case. For example, for the positive test case presented in Box 1, a mutant that could be obtained by replacing the value 500 in the call of operation `enterAmount(500)` with another valid value (e.g., 100) will be an equivalent mutant. In general, there are no satisfactory solutions to the problem of identifying equivalent mutant [13]. It is possible to apply some approaches helping to reduce the number or equivalent mutant (e.g., by rejecting replacement of values with values belonging to the same equivalence class), but in general they cannot be avoided and they cannot be identified without human assistance.

Results of analyzing the outcomes of executing negative test cases should provide valuable information pointing out weaknesses of the system and indicating possible ways of improving the tested system to make it more dependable and trustworthy.

7. Discussion of the Approach

The mutation testing-based approach to generation of negative test cases presented in this paper contributes to the area of software testing. Application of the approach can significantly improve a system ability to handle unexpected situations that, if unhandled adequately, could cause serious damage. However, there are certain problems that should be addressed in order to make the approach more attractive for practitioners.

The main problem concerning the approach, and mutation testing in general, is the high cost of generating and executing mutants [13]. Several cost reduction techniques were proposed so far (e.g., [35–42]) and they seem to be quite efficient in the context of mutating systems. However, none of them was applied in context of mutating test cases. As it is clear that the number of mutated test cases can be quite large, even for small systems, adaptation of such techniques in this context or development of new, better suited, here, techniques seems to be one of the main issues that needs to be addressed.

Another problem that may affect the practical use of the approach is the selection of adequate replacement values used by operators OPR and CPR. Such a set can be generated by taking all values assigned to a given parameter in all positive test cases and adding values that are out of scope of valid values for the parameter. The invalid values can be determined on the basis of equivalence partitioning [2]. Unfortunately, such set could be large and thus contribute significantly to the high cost of generating mutants. Similar problem, encountered in context of mutating systems, was tackled by rejecting operators replacing values of operands in expressions entirely. However, in this context, this approach seems to be not applicable, as the values passed in operations calls are vital elements of a test case. Future work on this approach should include a study on a possibility of minimizing the size of a such set or on working out a way allowing for using only a subset of these values each time the operators are applied.

Once more mutation testing related issue is the identification of equivalent mutants. A solution to this problem was not found in any context, yet. As it was stated in Section 6.3, the identification of equivalent mutants has to be done manually and it may require quite a large amount of time and effort to be invested. In practical implementation of the approach, it could be possible to add some procedures, for example, comparing the accepted mutants with positive test cases to calculate their similarity. Although it would probably not identify clearly the equivalent mutants, such comparison may point out some candidates and thus facilitate the human work.

The possibility to automate an approach plays an important role in its acceptance for practical use. In this case, the generation and execution of mutated test cases can be fully automated, but the analysis of results needs to be done mostly manually. Some preliminary assessment of the tested system can be made based only on the number of rejected and accepted test cases. However, to identify the causes of inadequate handling of unexpected situations, more detailed analysis of the outcomes of executing mutants is required. Though a detailed analysis needs to be manual and could be rather time consuming, it seems that the benefits of developing a highly dependable system being able to handle various undesired situations are worth the effort, especially in case of safety-critical systems.

8. Conclusions and Future Works

It is expected that a software system will work flawlessly in any situation. However, most testing methods focus only on checking if the system fulfills its specification, leaving the problem of assessing the system behaviour in unexpected situation unaddressed. The approach presented in this paper targets the problem by providing a mutation testing-based method for generating negative test cases that are able

to support an assessment of a system ability to handle a wide range of unexpected situations. The main advantages offered by this approach include a procedural (thus suitable for automation), systematic, and human-unbiased way of defining the negative test cases and no need of any formal or informal description of unexpected situations. The approach seems promising, but as it was stated in Section 7, there are still problems that need to be addressed by further works.

Future work concerning application of mutation testing to test cases should also include development of tools supporting the generation and execution of mutants and experimental evaluation of the approach. Availability of such tools would significantly increase the possibility of adapting the approach in practice.

Competing Interests

The author declares that there are no competing interests regarding the publication of this paper.

References

[1] A. Bertolino, "Software testing research: achievements, challenges, dreams," in *Proceedings of the Future of Software Engineering (FoSE '07)*, pp. 85–103, IEEE, Minneapolis, Minn, USA, May 2007.

[2] A. Roman, *Testing and Software Quality*, PWN, 2015 (Polish).

[3] F. Belli and M. Linschulte, "On 'Negative' tests of web applications," *Annals of Mathematics, Computing & Teleinformatics*, vol. 1, no. 5, pp. 44–56, 2008.

[4] F. Belli, "Finite state testing and analysis of graphical user interfaces," in *Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE '01)*, pp. 34–43, IEEE CS Press, November 2001.

[5] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[6] A. P. Mathur, *Mutation Testing, Encyclopedia of Software Engineering*, Taylor & Francis, Abingdon, UK, 1994.

[7] J. Strug, "Mutation testing approach to evaluation of design models," *Key Engineering Materials*, vol. 572, no. 1, pp. 543–546, 2014.

[8] S. C. Reid, "Empirical analysis of equivalence partitioning, boundary value analysis and random testing," in *Proceedings of the 4th International Software Metrics Symposium*, pp. 64–73, November 1997.

[9] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, John Wiley & Sons, New York, NY, USA, 2004.

[10] J. Zhang and S. C. Cheung, "Automated test case generation for the stress testing of multimedia systems," *Software: Practice and Experience*, vol. 32, no. 15, pp. 1411–1435, 2002.

[11] L. C. Briand, Y. Labiche, and M. Shousha, "Stress testing real-time systems with genetic algorithms," in *Proceedings of the 7th Annual Conference on Genetic and Evolutionary Computation (GECCO '05)*, pp. 1021–1028, ACM, Washington, DC, USA, June 2005.

[12] C. Barna, M. Litoiu, and H. Ghanbari, "Model-based performance testing (NIER track)," in *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*, pp. 872–875, Honolulu, Hawaii, USA, May 2011.

[13] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, article 6, pp. 649–678, 2011.

[14] H. Agrawal, R. Demillo, R. Hathaway et al., "Design of mutant operators for the C programming language," Tech. Rep. SERC-TR-41-P, Software Engineering Research Centre, Hyderabad, India, 1989.

[15] F. Belli, C. J. Budnik, A. Hollmann, T. Tuglular, and W. E. Wong, "Model-based mutation testing—approach and case studies," *Science of Computer Programming*, vol. 120, pp. 25–48, 2016.

[16] S. C. Pinto Ferraz Fabbri, J. C. Maldonado, T. Sugeta, and P. C. Masiero, "Mutation testing applied to validate specifications based on Statecharts," in *Proceedings of the 10th International Symposium on Software Reliability Engineering (ISSRE '99)*, pp. 210–219, November 1999.

[17] R. Schlick, W. Herzner, and E. Jbstl, "Fault-based generation of test cases from uml-models approach and some experiences," in *Computer Safety, Reliability, and Security: 30th International Conference,SAFECOMP 2011, Naples, Italy, September 19–22, 2011. Proceedings*, vol. 6894 of *Lecture Notes in Computer Science*, pp. 270–283, Springer, Berlin, Germany, 2011.

[18] B. K. Aichernig and P. A. Pari Salas, "Test case generation by OCL mutation and constraint solving," in *Proceedings of the 5th International Conference on Quality Software (QSIC '05)*, pp. 64–71, Melbourne, Australia, September 2005.

[19] Y. Jiang, S.-S. Hou, J.-H. Shan, L. Zhang, and B. Xie, "An approach to testing black-box components using contract-based mutation," *International Journal of Software Engineering and Knowledge Engineering*, vol. 18, no. 1, pp. 93–117, 2008.

[20] J. J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, and I. Medina-Bulo, "Evolutionary mutation testing," *Information and Software Technology*, vol. 53, no. 10, pp. 1108–1123, 2011.

[21] J. Strug, "Classification of mutation operators applied to design models," *Key Engineering Materials*, vol. 572, no. 1, pp. 539–542, 2014.

[22] K. Pan, X. Wu, and T. Xie, "Automatic test generation for mutation testing on database applications," in *Proceedings of the 8th International Workshop on Automation of Software Test (AST '13)*, pp. 111–117, May 2013.

[23] B. Aichernig, J. Auer, E. Jöbstl et al., "Model-based mutation testing of an industrial measurement device," in *Tests and Proofs*, M. Seidl and N. Tillmann, Eds., vol. 8570 of *Lecture Notes in Computer Science*, pp. 1–19, Springer, Berlin, Germany, 2014.

[24] B. K. Aichernig, H. Brandl, E. Jöbstl, W. Krenn, R. Schlick, and S. Tiran, "Killing strategies for model-based mutation testing," *Software Testing Verification and Reliability*, vol. 25, no. 8, pp. 716–748, 2015.

[25] G. Fraser and F. Wotawa, "Using model-checkers for mutation-based test-case generation, coverage analysis and specification analysis," in *Proceedings of the International Conference on Software Engineering Advances*, pp. 16–21, 2006.

[26] F. Belli and M. Beyazit, "A formal framework for mutation testing," in *Proceedings of the 4th IEEE International Conference on Secure Software Integration and Reliability Improvement (SSIRI '10)*, pp. 121–130, Singapore, June 2010.

[27] K. Bolazar and J. W. Fawcett, "Measuring component specification-implementation concordance with semantic mutation testing," in *Proceedings of the 26th International Conference on Computers and Their Applications (CATA '11)*, pp. 102–107, New Orleans, La, USA, March 2011.

[28] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, pp. 402–411, May 2005.

[29] M. Daran and P. Thévenod-Fosse, "Software error analysis: a real case study involving real faults and mutations," *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 3, pp. 158–171, 1996.

[30] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '14)*, pp. 654–665, 2014.

[31] B. H. Smith and L. Williams, "Should software testers use mutation analysis to augment a test set?" *Journal of Systems and Software*, vol. 82, no. 11, pp. 1819–1832, 2009.

[32] P. Black, V. Okun, and Y. Yesha, "Mutation operators for specifications," in *Proceedings of the 15th ASE IEEE International Automated Software Engineering Conference*, pp. 81–88, Grenoble, France, 2000.

[33] Y.-S. Ma, Y.-R. Kwon, and J. Offutt, "Inter-class mutation operators for Java," in *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE '02)*, pp. 352–363, November 2002.

[34] A. Derezińska, "Object-oriented mutation to assess the quality of tests," in *Proceedings of the 29th Euromicro Conference (EUROMICRO '03)*, pp. 417–420, Antalya, Turkey, September 2003.

[35] A. Mathur, "Performance, effectiveness, and reliability issues in software testing," in *Proceedings of the 15th Annual International Computer Software & Applications Conference (COMPSAC '79)*, pp. 604–605, Tokyo, Japan, 1991.

[36] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 2, pp. 99–118, 1996.

[37] M. P. Usaola and P. R. Mateo, "Mutation testing cost reduction techniques: a survey," *IEEE Software*, vol. 27, no. 3, pp. 80–86, 2010.

[38] L. Zhang, T. Xie, L. Zhang, N. Tillmann, J. de Halleux, and H. Mei, "Test generation via dynamic symbolic execution for mutation testing," in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM '10)*, pp. 1–10, IEEE, Timisoara, Romania, September 2010.

[39] G. Kaminski, U. Praphamontripong, P. Ammann, and J. Offutt, "A logic mutation approach to selective mutation for programs and queries," *Information and Software Technology*, vol. 53, no. 10, pp. 1137–1152, 2011.

[40] J. Strug and B. Strug, "Machine learning approach in mutation testing," in *Testing Software and Systems*, B. Nielsen and C. Weise, Eds., vol. 7641 of *Lecture Notes in Computer Science*, pp. 200–214, Springer, Berlin, Germany, 2012.

[41] J. Strug and B. Strug, "Using structural similarity to classify tests in mutation testing," *Applied Mechanics and Materials*, vol. 378, pp. 546–551, 2013.

[42] J. Strug and B. Strug, "Classifying mutants with decomposition kernel," in *Artificial Intelligence and Soft Computing*, vol. 9692 of *Lecture Notes in Computer Science*, pp. 644–654, Springer, 2016.