Mutation Operators for Object Constraint Language Specification

Kunxiang Jin, Kevin Lano

King's College London, Strand, London, WC2R 2LS, UK

Abstract

Mutation testing is a fault-based software testing technique for checking the effectiveness of a test suite through artificial defects. The mutation testing produces a satisfaction score, which is typically called the mutation score, to represent the quality of the input test suite. In mutation testing, from a programme p, a set of faulty programmes p' called mutants, is generated by making, for each p', a single simple change to the original programme p. To the best of our knowledge, no well-defined mutation operators for Object Constraint Language (OCL) specification has been proposed so far. The choice of mutation operators is an essential activity to ensure the accurate results of mutation testing. In this paper, we present a set of mutation operators to OCL specification. Since OCL is more and more popular in the scope of Model-Based Testing (MBT), the proposed mutation operators will help the mutation testing process, which involves the OCL specification. The paper presents the experimental results of an Android-platform financial application that is modelled by OCL specification. The results demonstrate the effectiveness of the proposed mutation operators for OCL specification.

Keywords

Object Constraint Language (OCL), Mutation Testing, Mutation Operators

1. Introduction

Software testing only can prove that the existence of system faults and software failure, but testing can never confirm the absence of defects because it is nearly impossible to perform exhaustive testing under the time and resource constraints [1]. Also, the system faults are the aggregation of simple defects. We can validate the System Under Testing (SUT) correctness by mutation testing based on the assumption that we can change the specification to simulate the real system defects.

Mutation testing is a fault-based software testing technique. The technique has been widely studied and used for nearly 50 years since a student paper can backtrack to 1971 [2]. Mutation testing provides a range of methods, tools, and reliable results for the software testing process. In mutation testing, from a programme p, a set of faulty programmes p' called mutants, is generated by making, for each p', a single simple change to the original programme p.

OCL 2021: 20th International Workshop on OCL and Textual Modeling, June 25 2021, Bergen, Norway

Kunxiang.jin@kcl.ac.uk (K. Jin); kevin.lano@kcl.ac.uk (K. Lano)

https://kclpure.kcl.ac.uk/portal/en/persons/kunxiang-jin(d183dcb5-a906-4a9d-af59-36ea9ea8fe78).html (K. Jin); https://www.kcl.ac.uk/people/kevin-lano (K. Lano)

0000-0002-9706-1410 (K. Lano)

MBT is a testing method in software testing to validate whether the SUT satisfies the specification by using design models. One key observation is that even well-constructed manual test cases can only exercise the programme feature by the limited amount of test data [3]. Many types of models can be used in MBT, such as Unified Modelling Language (UML), Labelled Transition System (LTS), Petri-Net and Markov Chains. In the MBT community, UML is a popular approach to perform the testing process. Object Constraint Language (OCL) is a type of declarative language that adds more details to UML models and now is a part of UML standard [4]. OCL has many benefits, and the most critical point is that OCL can add pre- and post-condition [5] to methods, operations and models.

Mutation testing is a testing method initially designed for evaluating the quality of the test suite by modifying the programme, and the majority of existing studies deal with kinds of programming languages [6]. However, these approaches are language-dependent, which means the mutation operators are specific to one particular programming language. Suppose we can perform the mutation testing on a higher abstract level than a specific implementation language, like OCL specification. In that case, the mutation operators can be platform-independent, and the mutants can be used for multiple implementation platforms.

Especially in MBT and regression testing, we prefer to evaluate the test suite before the actual implementation. When the design specification (OCL) is improved, then the mutation operators will be applied. The system implementation should always conform to the design specification. In this premise, the mutated OCL specification corresponds to the faulty programme. Therefore, the result against mutated OCL specification can reflect the quality of the test suite.

Although there are some studies worked on mutation operators, like [7] [8] etc., they are limited to a small subset of OCL specification. To the best of our knowledge, there is still no well-defined set of mutation operators for OCL specification so far, especially for complex operations, like operations to *Collection*. In this paper, we propose a set of mutation operators to OCL specification and all these mutation operators are based on first-order mutation testing, which means the mutants are created by only applying one mutation operator per mutant [9].

The remaining parts of this paper are structured as follows. Section 2 discusses the background of mutation testing and OCL specification, also some related works. In section 3, we present a set of mutation operators to OCL specification. Section 4 demonstrates a case study of the proposed mutation operators based on an Android-platform financial application modelled by OCL specification. Finally, section 5 concludes this paper and proposes future works.

2. Background & Related Works

There are two primary hypotheses within mutation testing designed to find valid test cases and real errors in the programme. The two hypotheses are: Competent Programmer Hypothesis(CPH) and Coupling Effect(CE). CPH means: assuming that programmers are capable, they try to develop programs better and achieve the right results, not the faulty ones. It focuses on the programmer's behaviour and intentions. CE(coupling effect) is more concerned with the category of errors in the variation test. A simple error is often caused by a single variation (such as a syntactic error), while a large and complex error is often caused by more variation. Complex variants are usually composed of many simple variants.

In addition to assessing the adequacy of the test suite, mutation testing can also simulate the real defects of the SUT by using variation defects, thus assisting in evaluating the effectiveness of the testing method proposed by researchers. The effect of variation defects generated by the mutation operator is similar to real defects in effectiveness evaluation [10].

OCL takes a compromise between natural language and mathematical notation to balance intelligibility and formality. OCL specification does not give a complete formal semantics, although some approaches attempt to formalise OCL, like [11]. Due to OCL being on the same abstract level as the system model, it is not dependent on any implementation language. Based on this platform-independent character, there is a need to perform mutation testing and propose mutation operators based on OCL specification.

In [12], the authors presented a detailed survey of mutation testing, which includes a concise description of the problems, methods, tools and common practices within the field of mutation testing. This work indicates that compared to code-based mutation testing, model-based mutation testing has not been researched much over the last years, but there is a growing interest in this direction.

In [13], the authors demonstrated a model-based mutation testing approach. They present two elementary mutation operators, insertion and omission, and the combinations of these two first-order operators to perform mutation testing. Three case studies from industry and real-life system are used to validate the effectiveness of their proposed approach. However, the proposed approach is hard to apply to the OCL specification.

As aforementioned, in [7] [8], the authors proposed some sets of mutation operators to OCL specification. Furthermore, they validated the effectiveness of the proposed mutation operators through case studies. However, these operators are only related to the primary and a small subset of the OCL specification. There is still a need to propose mutation operators related to complex data types, like *Collection*, and operations to these types.

In [14], the authors presented an approach using OCL mutation and aspect-oriented programming to validate the correctness of implementation and specification faults. But the mutant operators for this study are only limited to predefined primitive types, which are *Integer*, *Real*, *Boolean* and *String*. The proposed approach still not support collection types and operations to collections types, although these concepts are essential to OCL specification.

3. Mutation Operators

One major problem in software testing is the inability to know the number of system defects practically or theoretically, and mutation testing tries to solve this challenge by introducing mutants. Mutants are introduced by simple syntax change to the original specification. The mutant operators are the transition rules that define how to perform the syntactic changes. Based on the set of mutant operators, we can generate a set of mutated specification that allows us to perform mutation testing analysis.

Since all possible mutant operators to a particular specification are enormous, defining a basic set of mutant operators, which is typically considered the minimum set of mutant operators for mutation testing, is necessary. The proposed mutation operators are presented in **Table 1**.

All mutant operators are used in this work are first-order operators, which means the mutated

Table 1
Mutant Operators

NO.	Original Specification	Mutated Specification
1	=, <>	<>, =
2	≥, >	<, ≤
3	≤, <	>, ≥
4	*true*, *false*	*false*, *true*
5	p@pre	p
6	$forAll(p)$	$exists(not\ p)$
7	$exists(p)$	$forAll(not\ p)$
8	$include(p)$, $exclude(p)$	$exclude(p)$, $include(p)$
9	$select(p)$	$reject(p)$
10	$min()$, $max()$	$max()$, $min()$
11	$first()$, $last()$	$last()$, $first()$
12	$select(p) \rightarrow isEmpty()$	$select(p) \rightarrow notEmpty()$
13	*A and B*	*not A or not B*
14	*A or B*	*not A and not B*
15	$if(expression)$, $if(not\ expression)$	$if(not\ expression)$, $if(expression)$
16	**if** *con* **then** *block*1 **else** *block*2 **endif**	**if** *con* **then** *block*2 **else** *block*1 **endif**

specification is created by only applying exactly one mutant operator. The reason why we only apply first-order mutation testing is that high-order mutation testing is more like to produce equivalent mutants. High-order mutation testing applies multiple mutant operators to specification at the same time. While equivalent mutant is a mutant that has the same behaviour as the original specification [15]. For example, the original OCL specification is $if(expression)$, if we random apply the mutant operator twice, the specification may change to $if(not\ expression)$ then change back to the original one.

Table 2
Frequency of Expression

Expression	=	and	exists	or	implies	>	<	<>	sum	forAll
UML-C	1152	815	98	89	229	34	29	17	10	4
UML-Python	800	367	16	91	329	31	9	10	10	3

The proposed set of mutant operators is only the basic set that defined the necessary operators, and we consider these operators are the minimum set of mutant operators for mutation testing based on OCL specification. In deciding the mutation operators, we also estimate the frequency of different OCL expressions from two significant specification examples: UML to C and UML to Python generator. **Table 2** shows the frequency of the major used operators.

The overall idea of the mutant operators is the negation of the original specification. The first to third mutant operators are related to the arithmetic operator, and we only use the negation version instead of the random replacement by another arithmetic operator based on the CPH hypothesis. The programmer seldom uses an arbitrary operator instead of expecting one. The fourth mutation operator is the Boolean operand. The *@pre* is only allowed within the post-condition, which is the operator accesses the value before executing the operation.

The sixth to twelfth mutation operators change the specification to the operations for the complex collection data type. At the same time, the following two mutation operators are related to the logical relationship. The fifteenth mutation operator negates the expression to a conditional statement, and the final one is the mutant operator to the control flow statement.

4. Case Study

In this section, we present a case study to evaluate the proposed mutant operators. The case study is an Android-platform financial application, which is modelled by OCL specification. Due to the space limitation, the full details about the case study is available at [16], **Figure 1** demonstrates the fragment of case study specification. Because the case study lacks some OCL expressions, we also designed some specific specifications to perform the experiment.

```
operation discount(amount : double , r : double , time : double ) : double
pre: r > -1 & time >= 0
post: result = amount / ( ( 1 + r )->pow(time) );

operation value(r : double ) : double
pre: r > -1
post: upper = ( term * frequency )->floor() & c = coupon / frequency &
period = 1.0 / frequency & result = Integer.subrange(1,upper)->collect( i |
self.discount(c,r,i * period) )->sum() + self.discount(100,r,term);
```

Figure 1: Fragment of Case Study Specification

Table 3
Experiment Result

NO.	CF	MS	II
1	0.292	0.583	-0.061
2	0.246	1.00	0.087
3	0.024	1.00	0.0067
4	0.049	0.00	-0.037
5	0.073	0.00	-0.058
6	0.049	1.00	0.014
7	0.024	1.00	0.0067
8	0.000	-	-
9	0.024	0.00	-0.018
10	0.000	-	-
11	0.000	-	-
12	0.024	1.00	0.0067
13	0.073	1.00	0.021
14	0.024	1.00	0.0067
15	0.049	1.00	0.014
16	0.049	1.00	0.014

We first generate the test suite from the original OCL specification by using AgileUML, then

we inject mutants to the original specification [17]. For each mutated version OCL specification, we execute the test suite against the mutant to record the corresponding results. Finally, the results are collected into **Table 3**. We use three assessment metrics, which are contribution factor (CF), mutation scores (MS) and impact indicators (II), for each mutant operator [7]. These measures are defined to validate the effectiveness of the mutation operators, and reflecting the basic characteristics of these operators.

CF shows the percentage of generated mutants contributed by a specific mutation operator to the total created mutants. MS gives the proportion of mutants of each kind detected by the test suite generated from the original OCL specification. II shows how the mutation score obtained from the original test suite changes when a specific mutation operator is not applied.

The majority of the mutated version of OCL specifications are "killed" by the test suite generated by AgileUML. Some mutation operators fail to be detected because they do not change the behaviour of the original specification, like operator no.4, if the post-condition only uses the variables and not change their assignments, then with or without *@pre* modifier will not change the output of the operation. One interesting discovery is operator no.9. When we want to count the number of elements that satisfy specific criteria within the set, *select*() operator will be used, the mutated version is *reject*(). However, if the satisfying element is the exact half of the set, the count of the mutated version will exact the same as the original one, although the chosen elements are complementary.

Moreover, the impact indicators show how the mutation operators impact the assessment of the test suite. When we assess the quality of the test suite for the mutation operators that have the "negative" II, if we not apply any of them, the quality of the test suite will be overestimated. The overestimate of the test suite may lead the developers to be too confident to decrease the opportunity to find the potential system defects during the development process.

5. Conclusion & Future Works

Mutation testing is a testing method that validates the quality of the test suite by introducing artificial defects. Since OCL is more and more popular in the scope of MBT, the mutant operators to OCL specification will help the mutation testing process. In this paper, we present a set of mutant operators to OCL specification. The operators are suitable for primitive types and can also be applied to collection types and corresponding operations. We also performed a real-world case study to validate the proposed mutant operators.

In the future, we are planning to use AgileUML to generate mutated OCL specification automatically. Moreover, the performed experiment is only based on a small case study from the real world and lacks many OCL specifications, so we should validate our mutant operators on further case studies. Finally, we should supplement and complete our set of mutant operators according to further experimental results.

Acknowledgments

This research project is supported by China Scholarship Council (CSC) and King's College London joint scholarship. (K-CSC No. 201908060026)

References

[1] S. Dalal, K. Solanki, et al., Challenges of regression testing: A pragmatic perspective., International Journal of Advanced Research in Computer Science 9 (2018).

[2] R. J. Lipton, Fault diagnosis of computer programs, 1971.

[3] P. D. Marinescu, C. Cadar, make test-zesti: A symbolic execution solution for improving regression testing, in: 2012 34th International Conference on Software Engineering (ICSE), IEEE, 2012, pp. 716–726.

[4] S. Ali, T. Yue, M. Z. Iqbal, R. K. Panesar-Walawege, Insights on the use of ocl in diverse industrial applications, in: International Conference on System Analysis and Modeling, Springer, 2014, pp. 223–238.

[5] S. Weißleder, D. Sokenou, Automatic test case generation from uml models and ocl expressions, Software Engineering 2008 (2008).

[6] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE transactions on software engineering 37 (2010) 649–678.

[7] J. Strug, Classification of mutation operators applied to design models, in: Key Engineering Materials, volume 572, Trans Tech Publ, 2014, pp. 539–542.

[8] M. F. Granda, N. Condori-Fernández, T. E. Vos, O. Pastor, Mutation operators for uml class diagrams, in: International Conference on Advanced Information Systems Engineering, Springer, 2016, pp. 325–341.

[9] A. J. Offutt, Investigations of the software testing coupling effect, ACM Transactions on Software Engineering and Methodology (TOSEM) 1 (1992) 5–20.

[10] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, 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, 2014, pp. 654–665.

[11] A. D. Brucker, F. Tuong, B. Wolff, Featherweight ocl: A proposal for a machine-checked formal semantics for ocl 2.5, Archive of Formal Proofs (2014). https://isa-afp.org/entries/Featherweight_OCL.html, Formal proof development.

[12] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, M. Harman, Mutation testing advances: an analysis and survey, in: Advances in Computers, volume 112, Elsevier, 2019, pp. 275–378.

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

[14] L. C. Ascari, S. R. Vergilio, Mutation testing based on ocl specifications and aspect oriented programming, in: 2010 XXIX International Conference of the Chilean Computer Science Society, IEEE, 2010, pp. 43–50.

[15] L. Fernandes, M. Ribeiro, L. Carvalho, R. Gheyi, M. Mongiovi, A. Santos, A. Cavalcanti, F. Ferrari, J. C. Maldonado, Avoiding useless mutants, in: Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, 2017, pp. 187–198.

[16] K. Lano, K. Jin, S. Tyagi, Model-based testing and monitoring using agileuml, Procedia Computer Science 184 (2021) 773–778.

[17] AgileUML repository, 2021. URL: https://github.com/eclipse/agileuml/.