Quality Evaluation of Test Oracles Using Mutation

Ana Claudia Maciel, Rafael Oliveira and Márcio Delamaro
ICMC/USP
University of São Paulo
São Carlos, BRA
anamaciel@usp.br, rpaes@icmc.usp.br, delamaro@icmc.usp.br

***Abstract*—In software development, product quality is directly related to the quality of the development process. Therefore, Verification, Validation & Test (VV&T) activities performed through methods, techniques, and tools are needed for increasing productivity, quality, and cost reduction in software development. An essential point for the software testing activity is its automation, making it more reliable and less expensive. For the automation of testing activities, automated test oracles are crucial, representing a mechanism (program, process, or data) that indicates whether the output obtained for a test case is correct. In this paper, we use the concept of program mutation to create alternative implementations of oracles and evaluate their quality. The main contributions of this paper are: (1) propose specific mutation operators for oracles; (2) present a useful support tool for such mutation operators; and (3) establish an alternative to evaluate assertion-based test oracles. Through an empirical evaluation, our main finding is that mutations may help in assessing and improving the quality of test oracles, generating new oracles and/or test cases and decreasing the rate of test oracles errors.**

Keywords–Test Oracles; Mutation Testing; Mutation Operators;

I. INTRODUCTION

Automated test oracles are essential components in software testing activities. Defining a test oracle involves synthesizing an automated structure that is able to offer the tester an indicative verdict of system accuracy [1]. Thus, oracle is the mechanism that defines and gives a verdict about the correctness of a test execution [2]. Despite the importance of the oracle mechanisms, there is no systematic way to evaluate their quality [3].

In some cases, the results of running a test suite may have unwanted results, not due to problems in test data or program under test, but because of errors in the oracle implementation. Accordingly, test oracles correctness is as important as the selection of test inputs and, therefore, should be systematically implemented according to well-defined requirements [2].

This study aims to provide an alternative to improve the quality of test oracles, proposing an automated strategy for assessing quality of oracles, inserted in the cost amortization of realization of software testing. We extended the idea of mutation testing, applying it to evaluate the quality of test oracles implemented using the JUnit framework [4], a test framework which uses annotations to identify methods that specify a test. The main idea is to use test oracles to verify whether the oracles with mutations may contribute to reveal defects in programs.

We designed and created mutation operators to assertion-based test oracles written in JUnit format, based on the method assert signatures and its parameters. Operators have been developed to generate assertions that the tester did not create, or to correct oracles that have been written in the wrong way. Following the concepts of mutation test, oracles can be evaluated automatically. Thus, this work provides specific mutation operators to test oracles in order to systematize the evaluation of oracles.

The main contributions of this paper are related to the context of automation of processes associated with software engineering. In view of this, four contributions are provided through the following work:

- The definition and evaluation of mutation operators specific to assertion-based test oracles;

- MuJava 4 JUnit: a tool to generate the mutant oracles;

- Using the approach and tools with real programs of different functions, showing main operating characteristics and limitations of the proposed strategy; and

- Discussion on automated quality assessment of automated oracles and its importance for the improvement of automated tests.

The remainder of this paper is organized as follows: In Section II, we present the background with the main concepts related to this research. In Section III, we describe our mutation operators for JUnit assertion-based test oracles and our tool: MuJava 4 JUnit. In Section IV, we explain our empirical evaluation by describing the experiment design, research questions, research design and our experiment procedure. In Section V and Section VI we discuss the results of the experiment and threats to validity, respectively. Finally, we present the final remarks of our study in Section VII.

II. BACKGROUND

This section presents and discusses the concepts related to test oracles and mutation testing.

A. Test Oracles

Test oracles can be defined as a tester ("human oracle") or an external mechanism that can decide whether the output produced by a program is correct [5]. Typically, a test oracle is composed of two parts: (1) the expected behavior that is used to check the actual behavior of the System Under Test (SUT); and (2) a procedure to check if the actual result matches the expected output [2]. In this context, one can define that test oracle is a software testing technology, which can be associated with different processes and test techniques [6].

The "oracle problem" happens in cases when, depending on the SUT, it is extremely difficult to predict expected behaviors to be compared against current behaviors [5]. Depending on the oracle, problems like false positives and false negatives may occur:

 ISBN: 978-1-61208-553-1

- False positive: a test execution is identified as failing when in reality it passed, or the functionality works properly; and
- False negative: a test execution is identified as passing when in reality it failed, or there is some problem in functionality.

In this work, we use oracles in JUnit classes format. In JUnit framework, test oracles are written in the form of assertions [7] and tests are units, contributing to expose flaws in the current version of the program or regression faults introduced during maintenance.

B. Mutation Testing

Mutation [8] is a fault-based testing technique. The program being tested is changed several times, generating a set of alternative versions with syntactic changes. This technique measures the fault-finding effectiveness of test suites, on the basis of induced faults. The general principle underlying Mutation Testing is that the faults used by Mutation Testing represent the mistakes that programmers often make [9].

A transformation rule that generates a mutant from the original program is known as mutation operator [10]. Typical mutation operators are designed to modify variables and expressions by replacement, insertion or deletion operators [9].

III. MUTATION OPERATORS FOR ASSERTION-BASED TEST ORACLES

This section presents the mutation tool and a novel mutation operators set, which is specifically designed for test oracles written as JUnit classes.

A. MuJava 4 JUnit - a mutation testing tool for JUnit test oracles

We have adapted MuJava [11] to create a tool (MuJava 4 JUnit) to include our new mutation operators to test oracles, in order to systematize the evaluation of the oracles written using JUnit assertions. Operators were included in MuJava, using the existing code structure. The tool MuJava 4 JUnit is publicly available in [12].

B. Definition of "MuJava 4 JUnit's" operators

We defined generic mutation operators to introduce changes in the most common types of assertions of JUnit. Signature variations of the statements were created adding, removing, modifying, or replacing some setting values. In order to automate and systematize the evaluation of test oracles, we establish four classes of operators:

- **Adding:** parameters are added to the method `assert`;
- **Modifying:** parameters from the method `assert` are changed;
- **Replacing:** the method `assert` is replaced with another method `assert`; and
- **Removing:** parameters are removed from the method `assert`.

The mutation operators for assertion-based test oracles were classified in two levels:

- **Signature level:** changes are made on the type of method assert, or on the parameters received by the `assert` method; and
- **Annotation level:** changes are applied by replacing annotations, removing, or replacing its parameters.

1) Signature-based mutation operators: These mutation operators to test oracles were defined by combining the signatures of assert methods adding or removing parameters, or replacing the assert method by other assert method, improving the quality of test oracles through the creation of new oracles, or even adding new test cases.

The operators of this level are described in Table I. These operators were created according to the JUnit's specifications and can simulate problems, made by the tester, at the coding test oracles.

TABLE I. SIGNATURE LEVEL MUTATION OPERATORS.

Signature Level			
#	Class	Description	Acronym
1	Adding	Adding Threshold Value	ATV
2	Modifying	Decrement Constant from Threshold Value	DCfTV
3	Modifying	Increment Constant to Threshold Value	ICtTV
4	Replacing	Replace Boolean Assertion	RBA
5	Removing	Removing Threshold Value	RTV

2) Annotation-based mutation operators: We created the operators at the level of annotation changing or removing the timeout value, and adding possible exceptions that may occur in the execution of the oracles which were not previously thought by the tester. The operators from annotation level are presented in Table II.

TABLE II. ANNOTATION LEVEL MUTATION OPERATORS.

Annotation Level			
#	Class	Description	Acronym
1	Adding	Adding Expected Class	AEC
2	Modifying	Decrement Constant from Timeout	DCfT
3	Modifying	Increment Constant to Timeout	ICfT
4	Removing	Removing Timeout	RTA

C. Discussion analysis of each individual mutation operator

Next, we present an individual analysis of the effect of each mutation operator. The operators and their effects are:

ATV: adds the delta parameter, which is the third parameter of *assertEquals(expected, actual, delta)* method and kills mutants in two situations: (i) deprecated assert; and (ii) depending on the test value and the constant value.

The purpose of the delta parameter is to determine the maximum value of the difference between the numbers *expected* and *actual* so that they are considered the same value.

The ATV operator is a signature-level operator and belongs to the addition class. It adds the delta parameter. With this, one has a mutated version of the original oracle, in which the result is accepted as correct, considering an error rate. However, it is not always easy to know the acceptable value for a particular application. Currently, only the value 0001 is used as delta, but other values could be considered, taking into account the actual expected value. For example: *expected*/2, *expected*/10, *expected*/100, *expected*/1000, etc.

Figure 1 calculates a function of the second degree by means of the Bhaskara formula in which the coefficients are 1, 2 and 1. Depending on the value of the coefficients, the roots can generate values with several decimal places, so it is important to add the delta value (Figure 1, Line 5). Implementations with and without the delta value may have the same or different results depending on the value of the

 ISBN: 978-1-61208-553-1

delta and the coefficients in question. If the difference between oracles is never revealed, this may indicate that the fragility is in the test case or the error may be directly in the program being executed by the oracle.

```
1  @Test
2  public void testBhaskara() {
3      Bhaskara B = new Bhaskara();
4      double raiz = B.raiz(1,2,1);
5      assertEquals(-1.0, raiz , 0.1);
6  }
```

Figure 1. ATV example.

DCfTV: decrements the delta parameter, which is the third parameter of the method *assertEquals(expected, actual, delta)*. It kills the mutant depending on the decrement value and the value obtained during testing. If the oracle is designed with a case such that changing the precision value will change the result by applying this operator, the mutant oracle will have different results from the original oracle.

Figure 2 uses the *assertEquals(expected, actual, delta)* method in line 7, and a calculation of a rate over the value of a given product is being tested. The DCfTV operator allows the tester to adjust the delta value, decrementing it, according to his/her needs.

```
1  @Test
2  public void taxValue() {
3      Product prod=new Product("TV",600,Product.
           ELETRONIC);
4      CalTaxs calculatorTax=new CalTaxs();
5      double tax=calculatorTax.getTax(prod);
6      double finalPrice=prod.getPrice()*(1+tax);
7      assertEquals(660, finalPrice , 0.001);
8  }
```

Figure 2. DCfTV example.

In the example, the tester should provide a test case that has an error less than the initial error, but near it, ie: $0.0001 < error \leq 0.001$. For one such case, the original oracle indicates that the test passes but the mutant oracle indicates a failure. Thus, the mutant helps the tester verify his oracle or plan new test cases that exercise his oracle.

As in the case of the ATV operator, it is difficult to define how much the delta value decreases. Thus, one can think of extending the DCfTV operator using values such as *error*/2, *error*/10, *error*/100, *error*/1000, etc.

ICfTV: increments the delta parameter, the third parameter of the method *assertEquals(expected, actual, delta)*. It kills mutants depending on the incremented value. If the oracle is designed in the sense of changing the precision value, it will affect the result by applying this operator, then the mutant oracle will have different results from the original oracle.

The ICfTV operator follows the same logic as the DCfTV operator. However, one increment the value of the delta (ICfTV) and another decrement the value of the delta (DCfTV). In Figure 3, the oracle is on line 4, where the *assertEquals* method checks the result of a multiplication with the value of delta 0.1. By applying the ICfTV operator, a mutant oracle is generated with this increased delta value. The two implementations, original oracle and mutant oracle, may have the same or different results depending on the incremental value, which is set by the tester. If the difference between oracles is never revealed, this indicates the fragility of the test oracle.

```
1  @Test
2  public void assertSum(){
3      Calculator c = new Calculator();
4      assertEquals(4.0,c.mult(2,2),0.1);
5  }
```

Figure 3. ICfTV example.

As in the case of the ATV and DCfTV operators, it is difficult to define how much the delta value decreases. Thus, one can think of extending the ICfTV operator using values as *error*/2, *error*/10, *error*/100, *error*/1000, etc.

RBA: replaces boolean assertions (`assertTrue`, `assertFalse`). It produces high rate of dead mutants. Useful to reveal defects in oracles designed to Boolean cases, the replacement of the statements, the mutant oracle can improve the quality of the original oracle.

In Figure 4, the oracle presented in line 4 with the `assertTrue` method checks whether the String "Dog's god" is a palindrome, by applying the RBA operator, the `assertFalse` will be executed. If the result of the mutant oracle is different from the original oracle, the mutant will be considered dead. If the mutant or original oracles present the same result, the tester should check the test case and/or the program being tested.

```
1  @Test
2  public void testPalindrome3(){
3   CheckPalindrome cp=new CheckPalindrome();
4   assertTrue(cp.isPalindrome("Dog's god"));
5  }
```

Figure 4. RBA example.

RTV: removes the delta value, kills mutants depending on the test oracle. If the oracle is designed with a case that changing the precision value it changes the result by applying this operator, the mutant oracle will have different results from the original oracle.

In Figure 5 the arithmetic mean between two numbers is performed, and the oracle in line 5 has the value 0.001 of delta. The RTV operator removes this delta value. The two implementations, with the delta value and no delta value, may have the same or different results depending on the incremental value, which is set by the tester. In this case, we can have two correct implementations, in which it will be up to the tester to perform the analysis of the mutant oracle's correctness.

```
1  @Test
2  public void testAverage(){
3      int x=10, y=7;
4      assertEquals(8.5,calcAverage(10,7),0.001);
5  }
```

Figure 5. RTV example.

It is not recommended that an oracle be designed depending on the delta value. Therefore, if removing this value changes the result of the oracle, this can suggest to the tester to design new test cases that do not depend on this error value. In practice, this operator corresponds to changing the delta value to zero.

 ISBN: 978-1-61208-553-1

AEC: adds an expected class in annotation `@Test`. Kills mutants depending on the executed exception and the oracle running.

The AEC operator assists the tester in handling the exceptions that may occur during oracle execution. For example, in Figure 6 the exception`NullPointerException` avoids a month that does not exist be called in the `getAllDays` method.

```
1 @Test(expected=NullPointerException.class)
2 public void testException() {
3     int month = 4;
4     Assert.assertNotNull(calendar.getAllDays(month
          ));
5 }
```

Figure 6. AEC example.

AEC operator can add the exceptions: *IOException*, *NullPointerException*, *IllegalArgumentException*, *ClassNotFoundException*, *ArrayIndexOutOfBoundsException*, *ArithmeticException* and *Exception*.

The tester must add the exception according to the operation being performed, as well as done in Figure 6, where it is possible to avoid calling a null value.

DCfT: decrements a constant value of the timeout. Kills mutants depending on the decrement value and the value of the timeout. If the oracle depends on the previously established timeout value, using this operator, the mutant oracle will have different results from the original oracle;

Figure 7 is set to a value of timeout in 10 seconds. The DCfT operator can reduce this value, depending on the amount of records that are registered in the database, reducing this timeout is a good solution because it decreases the waiting time for the result. However, the tester must make a decision on how much to decrease in order to improve the performance of the test oracle.

```
1 @Test(timeout=10000)
2 public void searchEmployee(){
3     Employee emp=empDAO.findEmployee("12345");
4     Assert.assertEquals("JOHN",emp.getName());
5     verify(transaction,atMostOnce()).execute("
          12345");
6 }
```

Figure 7. DCfT example.

Deciding how much to decrease from this timeout value is not an easy decision, it is necessary to analyze how long it takes to process the method being tested. One solution is to use some predefined values: *timeout* − 10, *timeout* − 100, *timeout* − 1000, *timeout*/2, *timeout*/10, etc.

ICtT: increments a constant value of the timeout. Kills mutants depending on the increment value and the value of the timeout. If the oracle depends on the previously established timeout value, using this operator, the mutant oracle will have different results from the original oracle.

Figure 8 performs a test of a connection in the database, with the timeout of 1000 milliseconds. The mutant oracle generated by the ICtT operator may give a different result from the mutant oracle, causing the timeout value to be sufficient, or the mutant oracle may still live, giving the same result as the original oracle, showing that the problem may not be in the test program, but the program that performs the database connection. In this case, it is up to the tester to check the program and identify the error.

```
1 @Test(timeout=1000)
2 public void testGetConnection() {
3       Connection con = Connection.getConnection();
4       assertNull("Unable to connect!", con);
5 }
```

Figure 8. ICfT example.

In the example presented in Figure 8, the tester must define a test case whose runtime is higher than the original timeout value, but lower than the mutated value, ie 1000 < *runtime* ≤ 10000.

RTA: removes the timeout value. Kills mutants depending on the value of the timeout. If the oracle depends on the previously established timeout value, using this operator, the mutant oracle will have different results from the original oracle.

In JUnit, it is possible for a test to have a maximum time to run. For example, if the tester wants the test to take no more than 500 milliseconds, the following operation can be performed (Figure 9). However, some operations may take longer than the time set in the timeout parameter, and for this, the RTA operator removes this parameter, causing the test run to use the default JUnit timeout time.

```
1 @Test(timeout=500)
2 public void fastTestCase() {
3     assertEquals(1500, calendar.getSize());
4 }
```

Figure 9. RTA example.

This mutation operator causes the mutating oracle to not depend on the execution time of the test case and, in theory, could run for an infinite amount of time. In the case of the mutant being killed, that is, indicating that the test has passed, while the original oracle indicates that it has failed, there is an indication that the test case actually does not depend on the execution time and that the timeout clause was improperly used.

IV. Empirical Evaluation

In this section, we present an empirical evaluation involving the mutation of test oracles and some subject programs. The idea of this study is to apply specific operators to assertion- based test oracles (written with JUnit) and generate mutants. The syntactic modifications provided by the mutant test oracles are minimal. They reproduce faults in the signatures or annotations of assertion methods, as described in the previous section.

The generation of mutated test oracles suggest some repairs in unit tests previously defined. Further, new test cases can be found to improve the quality of the original test set.

A. Experiment Design

The experiment was conducted in order to verify whether the mutated oracles are able to identify failures that were not identified by original oracles, and analyze mutated test oracles for the purpose of their evaluation with respect to effectiveness and efficiency from the point of view of the tester revealing defects in faulty programs.

 ISBN: 978-1-61208-553-1

Figure 10 illustrates the steps performed in the experiment, namely: (1) run the original oracle against the subject program; (2) apply MuJava 4 JUnit mutation operators in the test oracles, generating the mutant oracles; (3) run all mutant oracles against the original subject programs; and (4) analyze the results.

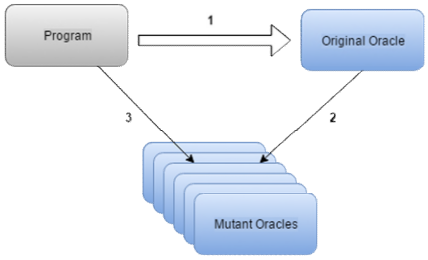

Figure 10. Step-by-step followed in this experiment.

B. Research Questions

The following Research Questions has been defined:

RQ1 Are the mutant test oracles able to improve the quality of the original oracle?

RQ2 Does the operator efficiency change depending on the program in test?

Aiming at answer these questions, we applied the mutation operators for test oracles in the assertion-based oracles of 5 subjects programs, which provided mutant oracles that are supposed to improve the original oracle.

C. Subject Programs

We selected five programs with different cyclomatic complexities, ranging from 1 to 6, to verify the effectiveness of the mutants in oracles, so revealing defects in the original oracles. The subject programs and their complexities are presented in Table III.

Each subject program has a test oracle written in JUnit form. Information about test oracles, including the number of failures in each test oracle used in the experiment are shown in Table IV.

TABLE III. SUBJECT PROGRAMS.

Program	#Cyclomatic Complexity	#Lines of code
Calculator	1	19
CheckPalindrome	3	16
BinarySearch	4	31
BubbleSort	4	66
ShoppingCart	6	117

TABLE IV. TEST ORACLES FROM SUBJECT PROGRAMS.

Oracle	#Cyclomatic Complexity	#Lines of Code	#Failures
TestingCalculator	1	58	7
TestingCheckPalindrome	1	61	2
TestingBinarySearch	1	114	2
TestingBubbleSort	1	146	3
TestingShoppingCart	1	212	13

D. Experimental Procedure

The experiment was divided in 3 steps (Figure 10). Five small programs were selected. Each original program had a correspondent testing class with some assertion-based oracle written through JUnit unit tests. Then, our mutation operators for test oracles were applied to each oracle, and the living and dead mutants were analyzed.

V. Results Discussion

In total, MuJava 4 JUnit tool implements 10 mutation operators to oracles from which 5 are signature level and 5 are annotation level. In this section, we provided a detailed analysis on the effects of using slightly modified version of test oracles to improve the quality of the test class.

A. Answers to RQs

[RQ1] Are the mutant test oracles able to improve the quality of the original oracle?

Some operators generate more mutants than others. The generation of mutants will depend on the assertion used, the parameters used in this assertion and which annotation is being employed. In this experiment, we collect data about the mutants generated by each operator implemented in MuJava 4 JUnit.

The percentages of live and dead mutants by each operator of the MuJava 4 JUnit tool are summarized in Table V. It can be observed that some operators kill more mutants than others. It is also observed that the MuJava 4 JUnit tool operators worked well in the generation of the mutant oracles.

TABLE V. MUTANTS ALIVE AND DEAD BY OPERATOR.

	Alive(%)	Dead(%)
ATV	94,74	5,26
DCfTV	80,00	20,00
ICtTV	80,00	20,00
RBA	50,00	50,00
RTV	62,50	37,50
AEC	78,57	21,43
DCfT	87,50	12,50
ICtT	70,00	30,00
RTA	100,00	0,0

[RQ2] Does the operator efficiency change depending on the program in test?

Each operator generates mutants according to the Assert method and their parameters, or the annotations used. Therefore, when performing the experiment, we conclude that the type of data that the subject program is using is what will determine which operator is more efficient in that situation.

In the context of our experiment, `CheckPalindrome` program, for example, works with boolean values, so that the operators which use these values are more efficient, in this case, RBA. The `Calculator` program works with integer and double values, causing the ATV, DCfTV, ICtTV and RTV operators more efficient. The `BinarySearch`, `ShoppingCart`, and `BubbleSort` programs perform operations with boolean, integer and double values, thus using all operators of these genres. Table VI shows the number of mutants generated by each operator in each program used in the experiment.

TABLE VI. MUTANTS GENERATED IN EACH PROGRAM SEPARATED BY OPERATOR.

	Calculator	CheckPalindrome	BinarySearch	BubbleSort	ShoppingCart
ATV	4	0	0	0	34
DCfTV	5	0	5	5	5
ICtTV	4	0	5	5	6
RBA	0	14	14	0	28
RTV	5	0	0	0	3
AEC	78	90	72	78	102
DCfT	3	0	0	0	5
ICfT	4	0	0	0	6
RTA	3	0	0	0	4

The most interesting mutant oracles are those giving results equal to the original oracles. They can suggest

 ISBN: 978-1-61208-553-1

new test cases, indicate weaknesses in the test case, and then identify errors in the program being tested.

In Table V, it is possible to observe that the generated mutants have a higher rate of live mutants compared to the dead mutants. Therefore, the answer to the QP1 research question in the context of this experiment is yes. However, in the future, a detailed analysis of mutants should be carried out for this result to be consolidated.

B. Pros and cons

In this first experiment, the operators performed well and we observed their behavior in different situations. We focused this experiment on operator's behavior, but we also collected some numbers about live mutants and dead mutants for further analysis.

ATV, DCfTV, ICtTV, and RTV are useful when a mutant is dead, because it indicates that the precision value that is making a difference in the outcome of the oracle. In practice, to obtain a mutant in this condition, the tester must pay attention to the fact that the test case is not necessary for the test case, and then change their oracle so that the precision value does not need to interfere to change the final value of the oracle's execution.

The AEC operator, in practice, to obtain a mutant in this condition, the tester must pay attention to the fact that the test case requires the exception added by the operator and it is interesting that the tester designed the oracle taking into account all the situations that may occur for the required exceptions.

DCfT, ICfT and RTA operators generate mutants that can be dead or alive. They are useful when a mutant is dead, because it shows that the timeout should be reconsidered by the tester when designing the test oracle.

VI. THREATS TO VALIDITY

This section presents the threats of this study on four different perspectives:

Internal validity: Our study is designed with a narrow scope – assertion-based test oracles. The experiment was designed to answer our RQs. We believe that the results were consistent to answers our RQs, leading to a high and acceptable internal validity.

External validity: The study evaluates the effectiveness of the mutation operators for assertion-based test oracles in five small Java applications. However, our experiment does not provide results to assume that the behavior of our technique will be the same in industrial-real-world systems. Further work is required in this context. In addition to that, our tool is designed only for Java applications, reducing the generalizations of our results.

Construct validity: The concept of mutation is useful in several contexts, making our construction validity higher. Hence, the size, and complexity, of the chosen applications are suitable to show the mutation operators effectiveness for JUnit-based test oracles.

Conclusion validity: We have presented our methodology in detail and we are providing the code of the tool we have developed. In this context, our results are associated with our results, and we therefore, claim that we have high conclusion validity.

VII. CONCLUSION

There were no systematic ways to assess the quality or accuracy of an automated test oracle. Thus, it is possible that in some cases, the results of running a test suite present unwanted results, not by problems in test data or test program, but because of errors in the implementation of the oracle. Therefore, this study designs mutation operators to oracles, until then, there was no work in this direction. Operators have been developed to test oracles written in JUnit format and defined replacing signatures of assert methods, adding parameters assert method, or removing parameters assert method.

Operators were implemented and included in MuJava tool. The experiment conducted in this study highlights the behavior of the operators when applied to simple programs and different ciclomatic complexities, data were collected from living and dead mutants, as well as detailed data for each operator in different cases.

We can conclude that using mutation test oracles collaborates in improving the quality of test oracles. The work also contributes presenting a systematic way of assessing the quality of oracles, which has not yet found in the literature.

Mutation operators to test oracles do not have a high rate of generation of dead mutants, however, they may reveal weaknesses in the original or even new test cases oracle, even not generating mutants dead. Therefore, the generated mutants should be scrutinized to make the actual operators.

As future work, we will carry out further experiments with real-world programs, seeking to affirm the results obtained with this work. In addition, we will design mutation operators to other oracle types.

ACKNOWLEDGMENT

Ana is supported by Fapesp (Grant Number 2014/09629-1).

REFERENCES

[1] R. A. Oliveira, U. Kanewala, and P. A. Nardi, "Automated test oracles: State of the art, taxonomies, and trends," Advances In Computers, Vol 95, vol. 95, 2014, pp. 113–199.

[2] M. Pezz and C. Zhang, "Automated test oracles: A survey," Advances in Computers, vol. 95, 2014, pp. 1–48.

[3] K. Shrestha and M. Rutherford, "An Empirical Evaluation of Assertions as Oracles," in Proceedings of the 4th ICST, March 2011, pp. 110–119.

[4] E. Beck and K. Gamma, "JUnit: A cook's tour," Java Report, vol. 4, no. 5, May 1999, pp. 27–38.

[5] E. J. Weyuker, "On testing non-testable programs," The Computer Journal, vol. 25, no. 4, 1982, pp. 465–470.

[6] L. Baresi and M. Young, "Test oracles," Technical Report CISTR-01, vol. 2, 2001, p. 9.

[7] D. S. Rosenblum, "Towards a method of programming with assertions," in Proceedings of the 14th ICSE. ACM, 1992, pp. 92–104.

[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," IEEE Computer Society Press, vol. 11, no. 4, Apr. 1978, pp. 34–41.

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

[10] R. Abraham and M. Erwig, "Mutation operators for spreadsheets," IEEE Transactions on Software Engineering, vol. 35, no. 1, January 2009, pp. 94–108.

[11] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "MuJava: A Mutation System for Java," in Proceedings of the 28th ICSE. New York, NY, USA: ACM, 2006, pp. 827–830.

[12] A. Maciel. MuJava 4 JUnit. [Online]. Available: https://goo.gl/ZGXqI5 (2016)

 ISBN: 978-1-61208-553-1