Detecting Equivalent Mutants by Means of Constraint Systems

Simona Nica, Mihai Nica and Franz Wotawa

Technical University of Graz, Institute for Software Technology
Inffeldgasse 16B/II, A-8010 Graz, Austria
{snica, mnica, wotawa}@ist.tugraz.at

***Abstract*—Mutation testing has been used along the research community as an efficient method to evaluate the process of software testing, i.e., the quality of the test suite. One major drawback is represented by the equivalent mutant problem. Through this current research we aim to come with a reliable solution to this problem and improve the available test suite pool. We do this by combining the mutation testing procedure together with a constraint satisfaction paradigm and the concept of distinguishing test cases. Mutation testing has been seen, in most of the cases, as a measure for evaluating the quality of a user's test suite. But, also mutation testing can be of great help in the test case generation process. By means of a constraint system we generate test scenarios able to distinguish between two different versions of a program. We start from the hypothesis that when our constraint system is not able to find any solution it might be the case that two equivalent mutants were encountered. The first empirical results, i.e. an increased mutation score, encourage us to further apply the strategy on medium size applications.**

Keywords-Mutation Testing; Equivalent Mutants; Mutation Score; Constraint Satisfaction Problem; Distinguishing Test Case.

I. INTRODUCTION

Mutation testing has been intensively used in a large number of experiments as an efficient way to detect the quality of a program's test suite [1]. It is a fault based technique that makes use of a well determined set of faults for measuring the efficiency of the test suite. In mutation testing the original program is slightly changed using mutation operators and the resulting mutant is executed using the test suite. If there is at least one failing test run, the mutant is said to be detected or killed. In mutation testing mutants that are not killed are alive. A test suite is said to be more effective if it has the capability to detect more mutants. The efficiency of a test suite in mutation testing is measured using the *mutation score*. The mutation score is equivalent to the number of mutants detected divided by the overall number of of non equivalent mutants (a mutant is said to be equivalent if there is not a test case which can distinguish the output of the mutant from the output of the original program). In the ideal case the mutation score is 1 and all mutants are successfully detected. Mutation analysis is a good metric for measuring the coverage levels achieved [2]. In the work presented in this paper, we are using mutations not only for measuring the efficiency but also for improving the quality of the test suite. The idea is to generate new test cases in case of mutants that cannot be detected. The proposed technique is based on the constraint representation of programs [3], [4] and on the concept of distinguishing test cases [5]. We convert both the program and its alive-mutants to a constraint satisfaction problem (CSP) [6] and ask the constraint solver to search for an input such that the two programs differ by at least one output value (computation of a distinguishing test case). When receiving such an input we are able to discriminate the mutant and the original program using the generated test case. An input that allows to discriminate two programs is called a *distinguishing test case*.

However, sometimes it might happen that a mutation over the original program will not change the semantics of the program, making thus hard to detect the change by a test case. This is one important issue which must be considered when generating the mutants. In the literature this is denoted as the equivalent mutant problem. Although several techniques are available in order to solve this problem [7], [8], we do not have a general solution. Therefore, in the first phase, we propose an algorithm to detect and reduce the equivalent mutants and then apply the distinguishing test cases algorithm in order to improve the test suite of a given application, i.e. improve the mutation score.

The goal is to clarify the research question whether it is always possible to increase the mutation score of a test suite from $x\%$, e.g., 70%, to 100%, based on the method of computing distinguishing test cases from alive-mutants and eliminating the equivalent ones. Our hypothesis in this respect is that a mutation score close to 100% can be achieved when using our proposed technique. In some initial experiments we observed increases of the mutation score even from 42% to 100%. However, in these experiments we only used small-size programs for testing the algorithm. Hence, the initial experiments confirmed our hypothesis and further more sophisticated experiments have to be carried

The research herein is partially conducted within the competence network Softnet Austria (www.soft-net.at) and funded by the Austrian Federal Ministry of Economics (bm:wa), the province of Styria, the Steirische Wirtschaftsförderungsgesellschaft mbH. (SFG), and the city of Vienna in terms of the center for innovation and technology (ZIT) and the Austrian Science Fund (FWF).

Authors are listed in alphabetical order.

 ISBN: 978-1-61208-168-7

out.

In what follows, we will give the basic definitions and then describe the proposed algorithm.

II. Basic Definitions

In order to have an accurate understanding of the algorithms described later in the paper, we present the basic definitions which we will use throughout our paper. We will explain what a mutation is, what we understand by equivalent mutant and what a constraint system is.

***Definition** 1:* **[Test Case]** A test case for a program Π is a set (I, O) where I is the input variable environment specifying the values of all input variables used in Π, and O the output variable environment, which does not necessarily specifies the values for all output variables.

A test case is a *failing test case* if and only if the output environment computed from the program Π when executed on input I is not consistent with the expected environment O. Otherwise, we say that the test case is a *passing test case*. If a test case is a failing (passing) test case, we also say that the program fails (passes) executing the test case.

***Definition** 2:* **[Test Suite]** A test suite TS for a program Π is a set of test cases of Π.

***Definition** 3:* **[Constraint Satisfaction Problem (CSP)]** A constraint satisfaction problem is a tuple (V, D, CO) where V is a set of variables defined over a set of domains D connected to each other by a set of arithmetic and boolean relations, called constraints CO. A solution for a CSP represents a valid instantiation of the variables V with values from D such that none of the constraints from CO is violated.

The variables from the CSP system do not necessarily need to be the variables used in the program.

***Definition** 4:* **[Mutant]** Given a program Π and a statement $S_\Pi \in \Pi$. Further let S'_Π be a statement that results from S_Π when applying changes like modifying the operator or a variable. We call the program Π', which we obtain when replacing S_Π with S'_Π , the mutant of program Π with respect to statement S_Π.

***Definition** 5:* **[Equivalent Mutant]** Given a program $\Pi \in \mathcal{L}$ and one of its mutant Π', we say that Π' is an equivalent mutant if the mutation that differentiates Π from Π' does not change the semantic of Π.

For a better understanding, we illustrate our definition with the example from Figures 1 and 2. Over the original version of the program from 1 we apply the relational operator replacement $\geq$.

***Definition** 6:* **[Distinguishing Test Case]** Given a program Π and one of its mutant Π', a distinguishing test case for program Π and its mutant Π' is a tuple $(I, \emptyset)$ such that for the input value I the output value of program Π differs from the output value of program Π'.

```
int a, b;
int compute;
if (a == b)
  compute = a;
else
  compute = (a + b)/2;
System.out.println(compute);
```

Figure 1. Original Program

```
int a, b;
int compute;
if (a >= b)
compute = a;
 else
  compute = (a + b)/2;
System.out.println(compute);
```

Figure 2. Equivalent Mutant for ROR

III. Research Strategy

Mutation testing is used mainly to determine the effectiveness of the given test suite by making use of the mutation score metric [9]. The idea of using mutation testing also for test case generation is not new. In [10], the authors use model based mutation testing in order to obtain distinguishing test cases from contract mutations. In [5], there are used distinguishing test cases obtained from mutants to reduce the number of diagnoses in case of fault localization. Mutation can also be used to indicate possible fixes of faulty programs as suggested in [11]. Moreover, the use of constraints for test case generation is also not new. In [12], the authors propose a method that makes use of the constraint systems to generate test cases. What distinguishes our work from the previous one is the combination of program mutation and constraint solving techniques in order to improve the mutation score of the test suites and, moreover, to help detecting the equivalent mutants [13].

Moreover, we try to determine an efficient method for eliminating the equivalent mutants. In what follows, we will describe first the algorithm we use to detect and remove the equivalent mutants from the set of generated mutants, and then the algorithm for improving the mutation score of test suite.

In our research, we make use of an extended version of the MuJava tool [14], [15] for computing the mutants and the MINION constraint solver [16] for obtaining the distinguishing test cases.

First, we define an algorithm which will translate the original program into a constraint system. We will call

 ISBN: 978-1-61208-168-7

this algorithm along the research experiment. It receives as input the original program and it gives as output the constraint system.

Algorithm Transform_To_CSP (Π)

1) Eliminate all loops from the original program by replacing them with a bounded number of nested conditional statements,
2) Convert Π to its equivalent SSA (static single assignment form) representation,
3) Convert SSA_{Π} into its corresponding constraint representation system.

For the elimination phase, the algorithm receives as input the original program Π and the set of generated mutants M_{Π} and offers at the output the new set of mutants.

Algorithm Eliminate_Equivalent_Mutants (Π, M_{Π})

1) Call ***Transform_To_CSP*** (Π) and obtain the constraint system CON_{Π}
2) For each Π_i from M_{Π}
   a) Call ***Transform_To_CSP*** (Π_i) and get the mutant constraint system CON_{Π_i}
   b) Create the constraint system CS, corresponding to $CON_{\Pi} \wedge CON_{\Pi_i}$, in order to apply the distinguishing test case restriction to the entire constraint system;
   c) Add the *same inputs - different outputs* constraints, i.e., $I(CON_{\Pi_i}) = I(CON_{\Pi})$ and $O(CON_{\Pi_i}) \neq O(CON_{\Pi})$ to the set of constraints CS.
   d) Solve the constraint system CS.
   e) if no solution is found, then do:
      i) Equivalent mutant detected
      ii) Remove mutant Π_i from M_{Π}

The above algorithm will always end either when one or several solutions are found or when the constraint system is not able to detect any solution. The condition *same input - different outputs* is first used in our research in order to help us detect an equivalent mutant. The experiments conducted have demonstrated that when the constraint solver is not able to offer at least one solution, the two programs taken into consideration are semantically *equivalent*.

Now, we present the method for improving the test suite of a given program. The algorithm takes as input the program Π and the test suite TS, and delivers as output a test suite that must have a higher mutation score than the original one.

Algorithm Generate_Test_Cases (Π, TS)

1) For the program Π generate the finite set of mutants M_{Π}.
2) M_{Π} = ***Eliminate_Equivalent_Mutants*** (Π, M_{Π})
3) Run the original test suite TS against the set of mutants M_{Π} and, compute the mutation score $\mu = \frac{Mutants_{Killed}}{Mutants_{Total}}$ where $Mutants_{Killed}$ is the number of killed mutants, and $Mutants_{Total}$ represents the total number of mutants.
4) If the mutation score μ is larger than a predefined value, return TS as result. In this case no improvement is necessary.
5) Otherwise, for each Π_i from M_{Π}
   a) Call ***Transform_To_CSP*** (Π_i) in order to convert the original program and the alive mutants into their CSP representation (for more information concerning program conversion to its constraint representation we refer the interested reader to [4])
6) Let CON_{Π} be the constraint representation of the original, bug-free, program.
7) For every constraint representation Π_{DS_i} of the available set of mutants Π_{DS}, $i = 1, ..., |\Pi_{DS}|$ do:
   a) Let CS be the set of constraints comprising the constraints from CON_{Π} and Π_{DS_i}.
   b) Add the *same inputs - different outputs* constraints, i.e., $I(\Pi_{DS_i}) = I(CON_{\Pi})$ and $O(\Pi_{DS_i}) \neq O(CON_{\Pi})$ to the set of constraints CS.
   c) Solve the constraint system CS.
   d) if a solution is found, then do:
      i) Let $T^{'}$ denote the valid test case that kills mutant Π_{DS_i}.
      ii) Add $T^{'}$ to the test suite TS;
      iii) Run $T^{'}$ against the set of mutants Π_{DS} of program Π and eliminate $\Pi^{'}_{DS_i}$ from Π_{DS} if $\Pi^{'}_{DS_i}$ fails on $T^{'}$.
8) Compute the mutation score μ and return TS as result.

Our research experiment was run over a small set of simple Java programs (no more than 200 lines of code), e.g, bubble sort, arithmetic operations, and some of the classes belonging to HTML Parser project [17] - a Java library used to parse HTML. Only small deviations, i.e., mutants that are close to the original program, were taken into consideration. Up do now we did not benefit from a significant test pool, but we were able to obtain a higher mutation score with a small number of generated distinguishing test cases and a smaller number of mutants. In order to demonstrate the practicability of our approach, we intend to substantially extend the empirical results based on larger programs with a variety of test suites.

In Table I, we summarize the first empirical results of our approach. By **LOC** we denote the lines of code, by $\mathbf{Line_{Cov}}$ we show the line coverage. For each class we record the initial mutation score, $\mathbf{MS_{Init}}$, resulted from the normal mutation testing procedure, and then, after applying our algorithm, we compute the new mutation score

Class	LOC	$\mathbf{Line_{Cov}}$	$\mathbf{MS_{Init}}$	$\mathbf{MS_{DTC}}$
tagTests.AppletTagTest	96	52.00%	27.41%	65.00%
tagTests.BaseHrefTagTest	13	23.00%	18.10%	54.13%
tagTests.BodyTagTest	7	86.00%	79.00%	84.30%
tagTests.CompositeTagTest	156	27.00%	17.56%	51.12%
tagTests.FormTagTest	46	11.00%	6.18%	19.23%
tagTests.LinkTagTest	58	43.00%	31.33%	65.34%
DivATC	21	100.00%	67.66%	100.00%
SumATC	18	100.00%	41.87%	100.00%
BubbleSort	43	99.97%	56.40%	79.10%

Table I
MUTATION SCORE WITH DISTINGUISHING TEST CASES

$\mathbf{MS_{DTC}}$, not taking into account the equivalent mutants.

The strategy is prone to some limitations, connected to the mutation testing tool and the constraint solver we use. The MINION solver does not support object-oriented constructs. Concerning the mutations we produce, we are not able to mutate constant values, nor to add or remove statements.

IV. CONCLUSION

In this paper, we aim at improving the quality (given as the mutation score) of a program's test suite. We achieve this by generating distinguishing test cases for extending the available test suite, and also by reducing the number of equivalent mutants. A distinguishing test cases is a test case that allows for distinguishing a program from its mutant using the same input. When adding this test case to the test suite, the mutation score of the new test suite has to increase, assuming a mutant that is not equivalent to the original program.

Up to now, the obtained empirical results support the claim that our approach improves test suites. However, we further strengthen the empirical results and aim to test our algorithm on medium scale applications.

REFERENCES

[1] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," in *IEEE Transactions of Software Engineering*, vol. PP, no. 99, Paris, France, 2010, p. 1.

[2] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria," in *IEEE Transactions on Software Engineering*, September 2006, pp. 608–624.

[3] R. Ceballos, M. Nica, J. Weber, and F. Wotawa, "On the complexity of program debugging using constraints for modeling the program's syntax and semantics," in *Proc. Conference of the Spanish Association for Artificial Intelligence (CAEPIA)*, Seville, Spain, 2009, pp. 22–31.

[4] M. Nica, J. Weber, and F. Wotawa, "How to debug sequential code by means of constraint representation," in *International Workshop on Principles of Diagnosis (DX-08)*, Leura, Australia.

[5] F. Wotawa, M. Nica, and B. K. Aichernig, "Generating Distinguishing Tests using the MINION Constraint Solver," in *CSTVA 2010: Proceedings of the 2nd Workshop on Constraints for Testing, Verification and Analysis*, Paris, France, 2010, pp. 325–330.

[6] R. Dechter, *Constraint Processing*. The Morgan Kaufmann Series in Artificial Intelligence, 2003.

[7] A. J. Offutt and W. M. Craft, "Using compiler optimization techniques to detect equivalent mutants," in *Software Testing, Verification, and Reliability*, 1994, pp. 131–154.

[8] D. Schuler and A. Zeller, "(Un-)Covering Equivalent Mutants," in *ICST '10: Third International Conference on Software Testing, Verification and Validation*. Paris, France: IEEE Computer Society, April 2010, pp. 45–54.

[9] J. H. Andrews, L. Briand, and Y. Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?" in *Proceedings of IEEE International Conference on Software Engineering*, St. Louis, MO, USA, May 2005, pp. 402–411.

[10] W. Krenn and B. K. Aichernig, "Test Case Generation by Contract Mutation in Spec #," in *Electronic Notes in Theoretical Computer Science*, 2009, pp. 71–86.

[11] V. Debroy and W. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *Third International Conference on Software Testing, Verification and Validation (ICST 2010)*, Paris, France, 2010, pp. 65–74.

[12] A. Gotlieb, B. Botella, and M. Rueher, "Automatic Test Data Generation using Constraint Solving Techniques," in *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software testing and analysis*, Clearwater Beach, Florida, United States, 1998, pp. 53–62.

[13] B. J. M. Grün, D. Schuler, and A. Zeller, "The Impact of Equivalent Mutants," in *IEEE International Conference on Software Testing, Verification, and Validation Workshops*, Denver, USA, 2009, pp. 192–199.

[14] Y. Ma, J. Offutt, and Y. Kwon, "Mujava : An automated class mutation system," in *Software Testing, Verification and Reliability* , 2005, pp. 97–133.

[15] S. Nica and B. Peischl, "Challenges in Applying Mutation Analysis on EJB-based Business Applications," in *Proceedings of Metrikon 2009*, Kaiserslautern, Germany, November 2009.

[16] I. Gent, C. Jefferson, and I. Miguel, "Minion: A fast, scalable, constraint solver," in *17th European Conference on Artificial Intelligence ECAI-06*, Trento, Italy, 2006, pp. 98–102.

[17] HTML Parser, "http://htmlparser.sourceforge.net/," 2011.

 ISBN: 978-1-61208-168-7