Benchmarking Testing Strategies with Tools from Mutation Analysis

Ralph Guderlei* René Just Christoph Schneckenburger
Franz Schweiggert
Institute of Applied Information Processing
Ulm University, D-89069 Ulm, Germany
{ralph.guderlei, rene.just, christoph.schneckenburger, franz.schweiggert}@uni-ulm.de

Abstract

The assessment of a testing strategy and the comparison of different testing strategies is a crucial part in current research on software testing. Often, manual error seeding is used to generate faulty programs. As a consequence, the results obtained from the examination of these programs are often not reproducible and likely to be biased.

In this paper, a flexible approach to the benchmarking of testing strategies is presented. The approach utilizes well-known results from mutation analysis to construct an objective effectiveness measure for test oracles. This measure allows to draw conclusions not only on the effectiveness of a single testing strategy but also to compare different testing strategies by their effectiveness measure.

1 Introduction

Manual extensive testing is often not feasible. Therefore, the automation of the software testing process is necessary to reduce the testing effort. The automated testing requires solutions for two problems: the automatic generation of test inputs and the automatic validation of the output of the implementation under test (IUT). The latter problem is often referred to as the oracle problem (c.f. [18]). The combination of a method to generate the test input and an oracle will be referred to as a *testing strategy*.

A class of oracles, the so-called *partial oracles* (c.f. [2]), seem to be a promising class of techniques to automate the verification of the output of the IUT. Partial oracles are characterized by the property, that if an IUT is judged to be faulty by the oracle, then the IUT contains a fault. That means that the oracle produces no false-positive results. On the other hand, if an IUT is judged to be correct, it may contain a fault. The fact that partial oracles may leave some faults undetected requires a strict assessment of the proposed oracles.

The test input has also an impact on the ability of the testing strategy to detect faults in the IUT. Several methods have been proposed for the assessment of test inputs. In the present paper, methods from mutation analysis will be used to measure the adequacy of the test input.

Often, selected example programs are used for the assessment of testing methods. The assessment procedure usually starts with the manual insertion of errors to the programs. Then, the testing method is measured by its ability to detect the inserted faults. The results of this kind of assessment are hard to reproduce and can hardly be used for the comparison to other testing methods.

Therefore, a benchmarking method based on known results from mutation analysis is presented in this paper. After a short introduction to mutation analysis in Section 2, the proposed approach is depicted and an example is given in Section 3. Thereafter, the presented approach is discussed in Section 4. Finally, a conclusion is given in Section 5.

2 Preliminaries

Mutation analysis is a fault-based method to examine the effectiveness of a test set. It has been explored since 1977 and was originally introduced in [4, 6]. In order to assess the test case adequacy, faults are seeded systematically into the implementation which is to be investigated. Thus, faulty versions of the original implementation, so-called *mutants*, are generated. In contrary to the classical error seeding (c.f. [13]), where the seeding is led by the intuition of experienced engineers, the way of seeding faults in mutation analysis is formally described by so-called *mutation operators*. Examples for mutation operators are the replacement of variables by constants or swapping the comparison operators (i.e. a<b will be replaced by e.g. a>b or a==b) (see e.g. [9, 10] for definitions of mutation operators). Due to the methodical seeding of faults the way of obtaining the mutants is reproduceable and the applicability is indepen-

*work supported by the DFG research training group 1100 "Modellierung, Analyse und Simulation in der Wirtschaftsmathematik"

dent of the semantics of the concerning implementation, i.e. the implementation can be chosen arbitrarily and the corresponding semantics is exclusively covered by the test cases. This fact can be regarded as fundamental in order to yield reliable and comparable benchmarking results of those test cases.

Mutation analysis is based on two hypotheses, namely the so called *competent programmer hypothesis* and the *coupling effect*. The latter assumes that complex faults can be regarded as an accumulation of small mutations. Due to this premise it can be said that every test case which detects a simple mutation will reveal a complex fault. On the other hand the competent programmer hypothesis implies that an application written by an experienced programmer contains just small faults, i.e. these faults are similar to simple mutations. Therefore, every mutant contains just one mutation to satisfy the indicated hypotheses and to avoid an exploding number of mutants as a result of combining multiple mutation operators.

Before the mutants are generated all test cases are applied to the original implementation in order to ensure that the implementation as well as the test cases are correct (correctness means for every input value and test case respectively the output of the implementation meets the expectation). Thereafter, mutants are generated by applying all mutant operators to the original implementation.

Every test case is then applied to the original implementation as well as every mutant and the outputs are compared. If the outputs differ, the mutant is said to be *killed*. Owing to Rice's theorem [17] which states that equivalence of nontrivial programs is not decidable in general, it can be necessary to inspect the not-killed mutants for equivalence manually (A mutant can produce the same output as the original implementation either if the test cases don't cover the mutation or if the mutation has no effect, e.g. a post-increment of a variable at the end of its life cycle).

As a result of disassociating the equivalent mutants from the whole bundle the number of killed mutants can be related to the number of mutants not being equivalent to the original implementation. The subsequent quotient $M_s(T)$is referred to as *mutation score*.

$$M_s(T) = \frac{\#\{\text{killed mutants}\}}{\#\{\text{not-equivalent mutants}\}}$$

Thus, the mutation score is suitable for measuring the adequacy of the test input. Further extensive research results and the current state of affairs in the field of mutation analysis can be found in [15].

3 A Benchmark for Testing Strategies

Recall that a testing strategy consists of two parts, a method to select test input data and a method to verify the actual output of the IUT (an oracle). Each part of the testing strategy will be considered in a separate quality measure. The benchmark of the testing strategy T is then defined as the product

$$E(T) = E_{\text{in}}(T) \cdot E_{\text{out}}(T)$$

of the quality measures for the generated test input and the oracle, E_{in} and E_{out}, respectively. These quality measures are based on results from mutation analysis. The measure E_{in} is equal to the mutation score as presented in Section 2.

In order to benchmark a testing strategy T, the following steps have to be proceeded:

1. **Choose a benchmark implementation**:
   The domain and the structure of the implementation should match the testing strategy to be assessed and the implementation itself should be representative for the testing problem to be solved by the testing method. The implementation should be complex enough so that a large number of mutants can be generated. In the following, the chosen implementation will be referred to as *original implementation*.

2. **Apply T to the benchmark implementation**:
   The testing strategy should not detect a bug in the original implementation. Otherwise, either the original implementation or the testing strategy has to be corrected.

3. **Generate mutants**:
   This step can be automated using various tools, e.g. for programs written in Java, Fortran, C#, or PHP. One possibility is to consider programs written in Java and to use the tool MuJava (c.f. [11]) to create the mutants.

4. **Determine the number of non-equivalent mutants** N: Usually, this step has to be proceeded manually, as there is no tool for that. But an algorithm for the automatic detection of equivalent mutants is presented in [16].

5. **Examination of T**:
   For each mutant, do:

   (a) Generate test input according to the input generation method taken from T

   (b) Execute the mutant and compare the output of the mutant to the output of the original implementation. If the outputs differ increase m by one.

   (c) Apply the oracle of T to the mutant. If the oracle judges the mutant to be faulty, increase n by one.

6. **Compute $E(T)$**:

$$E(T) = \frac{m}{N} \cdot \frac{n}{m} = \frac{n}{N}$$

The comparison of the mutants outputs to the outputs of the original implementation can be interpreted as a perfect oracle (see e.g. [3]) as the original implementation is a defect free version of the mutant. The perfect oracle is the best available oracle. Therefore, if the same inputs are used for the perfect oracle and the examined testing strategy T, the number of mutants killed by T cannot exceed the number of mutants killed by the comparison with the original implementation. If the oracle of T is a partial oracle, it holds that $0 \leq E_{\text{out}}(T) \leq 1$ and thus $0 \leq E(T) \leq 1$. It is necessary to benchmark only partial oracles, because if the oracle can have false-positive decisions, the measure E_{out} makes no sense.

The separation of the overall benchmark in two parts allows to separate the impact of the two parts of the testing strategy on the overall measure. A low value in one of the two measures indicates which part of the testing strategy has to be improved.

In principal, it is not necessary to use the mutation score as E_{in}. Any other adequacy criterion, e.g. a coverage measure, can be used instead. By doing that, the manual detection of equivalent mutants can be avoided, as the comparison with the original implementation as a benchmark for the oracle does not require to identify equivalent mutants. If the input adequacy criterion can be automatically obtained, the benchmarking process itself can be automated.

Furthermore, the presented approach can be used to compare different testing strategies T_1 and T_2. If it holds $E(T_1) < E(T_2)$, then it can be concluded that T_2 is more effective than T_1 and in that sense T_2 is better than T_1.

In our past work, we have used the proposed benchmark to examine various testing strategies. In [12], the effectiveness of Metamorphic Testing (see e.g. [5]) was examined using different implementations for the computation of matrix determinants. Roughly said, Metamorphic Testing checks necessary (post-)conditions on tuples of program outputs. The preconditions (for tuples of inputs) and post-conditions (for tuples of outputs) are specified by so-called Metamorphic relations. Different metamorphic relations were assessed and general hints on the choice of good relations are derived from the empirical results. Each Metamorphic Relation implicitly defines a testing strategy.

In the empirical study, 306 mutants were created, 37 of the mutants are equivalent to the original implementation ($N = 269$). For all of the relations, it turned out that $E_{\text{in}} = 0.978$ ($m = 263$). Therefore, the different relations differ only by E_{out} ranging from $E_{\text{out}}(R2) = 0.44$ to $E_{\text{out}}(R12) = 1$. Fig. 3 gives an overview on the results. On the x-axis, the size of the input matrices is shown. The y-axis depicts the oracle effectiveness E_{out}.

An approach for the fully automated testing of imaging applications by means of random and Metamorphic testing

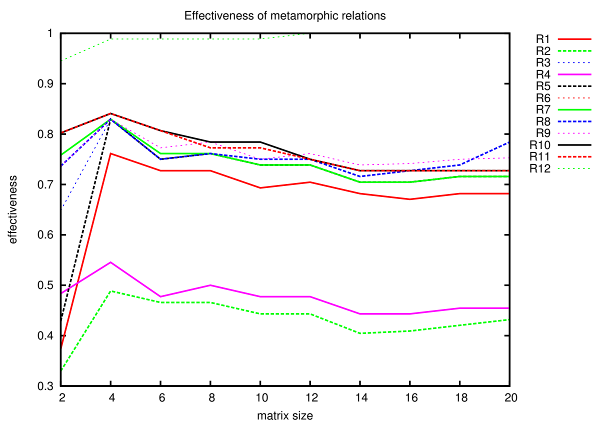

Figure 1. Example: effectiveness of different metamorphic relations

was presented and examined in [7]. Here, the influence of different techniques for the generation of input data was taken into consideration and quantified using the proposed approach. The effectiveness of statistical methods to test randomized software was examined in [8]. In these papers, the proposed methods were also compare to other testing strategies.

4 Discussion

Some restrictions must be considered when using the proposed approach. First, the weaknesses of mutation analysis also apply to our approach. There is only empirical evidence that the general assumptions of mutation analysis, the competent programmer hypothesis and the coupling effect actually hold. Additionally, it is not clear, that the results derived from experiments based on mutation analysis can be transferred to real world faults. An empirical study (c.f. [1]) supports this hypothesis, but that has to be confirmed by further research.

A further technical limitation is the tool support. All of the available tools have certain limitations, e.g. MuJava cannot work on source code containing Java 1.5 language features. Other tools implement only some mutation operators. Remind that our approach is not efficiently applicable without appropriate tool support.

On the other hand, the presented approach is very generic. The implementation which is used to generate the mutants is not restricted to a specific application domain. The proposed approach can be applied to any testing strategy which consists of a partial oracle. Therefore, the benchmark implementation and the testing strategy be chosen to fit exactly to the research project.

Given the necessary tools and testing strategies, the approach can be fully automated. Therefore, it is possible to examine either a large number of testing strategies or to apply these testing strategies on a large number of different programs.

To make the approach more efficient, the mutation operators can also be selected. In [14], an empirical study determining the most relevant mutation operators is presented. Manual error seeding often leads to faults which support the claims of the researcher. In contrary, the proposed effectiveness measure based on the formal specification of the mutation operators. Therefore, it leads to (at least more) objective results of the benchmarks.

The two parts of the benchmark can be used separately. For example, if two testing strategies share a common method for the generation of the input, then only the effectiveness measure of the oracles have to be considered.

5 Conclusion

In the present paper, a very flexible method to measure the effectiveness of a testing strategy is presented. By choosing the implementation to generate the mutants, the approach can be tailored to the researchers needs. The automation of the benchmarking process also allows to consider a large number of different programs in empirical studies. Therefore, the presented approach can be (and has been) applied to a broad range of different testing problems.

References

[1] J. Andrews, L. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th International Conference on Software Engineering (ICSE 2005)*, pages 402–411. ACM, 2005.

[2] A. Bertolino. Software Testing Research: Achievements, Challenges, Dreams. In *Proceedings of the International Conference on Software Engineering (ICSE 2007)*, pages 85–103. IEEE Computer Society Washington, DC, USA, 2007.

[3] R. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison-Wesley Professional, 1999.

[4] T. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, 1980.

[5] T. Y. Chen, T. Tse, and Z. Zhou. Fault based testing in the absence of an oracle. In *Proceedings of the 25th IEEE Annual International Computer Software and Applications Conference (COMPSAC 2001)*, pages 172–178. IEEE Computer Society, 2001.

[6] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11:34–41, 1978.

[7] R. Guderlei and J. Mayer. Towards automatic testing of imaging software by means of random and metamorphic testing. *International Journal on Software Engineering and Knowledge Engineering (accepted; Special Issue on Quality Software, Invited extended version)*, 18(3):??, 2008.

[8] R. Guderlei, J. Mayer, C. Schneckenburger, and F. Fleischer. Testing randomized software by means of statistical hypothesis tests. In *Proceedings of SOQUA '07: Fourth international workshop on Software quality assurance*, pages 46–54, New York, NY, USA, 2007. ACM.

[9] K. N. King and A. J. Offutt. A fortran language system for mutation-based software testing. *Software Practice and Experience*, 21(7):685–718, 1991.

[10] 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 2002)*, pages 352–366, 2002.

[11] Y.-S. Ma, J. Offutt, and Y. R. Kwon. Mujava: an automated class mutation system. *Softw. Test., Verif. Reliab.*, 15(2):97–133, 2005.

[12] J. Mayer and R. Guderlei. An Empirical Study on the Selection of Good Metamorphic Relations. In *Proceedings of the 30th Annual International Computer Software and Applications Conference, 2006 (COMPSAC 06).*, volume 1, 2006.

[13] H. Mills. On the Statistical Validation of Computer Programs. Technical report, IBM FSD Report, 1970.

[14] A. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5(2):99–118, 1996.

[15] J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In *Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries, San Jose, CA*, pages 45–55, 2000.

[16] A. Ofutt and J. Pan. Detecting equivalent mutants and the feasible path problem. *The Journal of Software Testing, Verification, and Reliability*, 7(3):165–192, 1997.

[17] H. Rice. Classes of Recursively Enumerable Sets and Their Decision Problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[18] E. Weyuker. On Testing Non-Testable Programs. *The Computer Journal*, 25(4):465–470, 1982.