Breeding High-Impact Mutations

Birgit Schwarz, David Schuler, Andreas Zeller
Saarland University
Saarbrücken, Germany
birgit.schwarz@stud.uni-saarland.de · {*ds, zeller*}*@cs.uni-saarland.de*

***Abstract*—Mutation testing was developed to measure the adequacy of a test suite by seeding artificial bugs (mutations) into a program, and checking whether the test suite detects them. An undetected mutation either indicates a insufficiency in the test suite and provides means for improvement, or it is an *equivalent mutation* that cannot be detected because it does not change the program's semantics. Impact metrics—that quantify the difference between a run of the original and the mutated version of a program—are one way to detect non-equivalent mutants. In this paper we present a *genetic algorithm* that aims to produce a set of mutations that have a *high impact*, are *not detected* by the test suite, and at the same time are *well spread* all over the code. We believe that such a set is useful for improving a test suite, as a high impact of a mutation implies it caused a grave damage, which is not detected by the test suite, and that the mutation is likely to be non-equivalent.**

First results are promising: The number of undetected mutants in a set of evolved mutants increases from 20 to over 70 percent, the average impact of these undetected mutants grows at the same time by a factor of 5.

***Keywords*-mutation testing, genetic algorithm**

I. Introduction

In software development huge costs could be avoided by improving the testing infrastructure that allows detecting software defects earlier and more efficiently [5]. Before being able to improve the testing infrastructure, one has to know its deficiencies. *Mutation testing* aims at detecting such deficiencies. To this end, artificial defects, *mutations*, are inserted into the program by using a set of *mutation operators*. A mutation that is not detected by the test suite gives a hint on how the test suite can be improved. However, there are also *equivalent mutants*, which are mutations that do not change the program's semantics, and thus, cannot be detected. These equivalent mutants impose a major drawback as they dilute the results of mutation testing.

In our previous work, we developed impact metrics that help to detect non-equivalent mutants [8, 10]. We believe that a set of undetected mutants that contains as few equivalent mutants as possible and is well spread over the code is most beneficial for improving a test suite, as it highlights deficiencies of the test suite throughout the program.

To this end, we apply a genetic algorithm that aims to produce such a set of mutants, which fulfills the following three requirements. The mutations should (1) have a high impact, (2) not be detected by the test suite, (3) and be well spread over the code.

In this paper we make the following contributions:

- We present a genetic algorithm that produces mutations with high impact, that are not detected, and well spread over the code.
- We provide an implementation of our approach and evaluate it on a medium sized project.

The rest of the paper is structured as follows: First we comment on the background of our work (Section II). Then we explain the details of the genetic algorithm (Section III), and shortly comment on the implementation (Section IV). In Section V we present an evaluation of our approach on a medium sized Java project. After this we comment on the threats to validity (Section V-D) and the related work (Section VI). We close with conclusions and consequences (Section VII).

II. Background

A. Impact of Mutations

In our previous work, we introduced JAVALANCHE, a mutation testing framework developed with a focus on automation and efficiency [8, 9]. For JAVALANCHE, we introduced several impact measures to detect non-equivalent mutants, such as *invariant impact* [8], *impact on return values*, and *coverage impact* [10]. The impact measures compare properties of an original (not mutated) run with properties of a mutated run. The coverage impact for example is defined as the number of methods that have a difference in their line coverage—that is at least one line in the method is executed with a different frequency in the mutated run than in the original run. A high impact of a mutation implies that it caused severe damage across the program execution. Tests that fail to detect such mutations might have a bad oracle quality, because they provide input to trigger this behavior but do not check the results well enough.

Previous experiments [10] have shown that among all the measures the coverage impact performed best—non-equivalent mutations were detected with a precision of 75% and a recall of 56%. Therefore, our approach tries to produce a set of mutations with a *maximized coverage impact* as these mutations are more likely to be non-equivalent, and thus, more useful for improving a test suite.

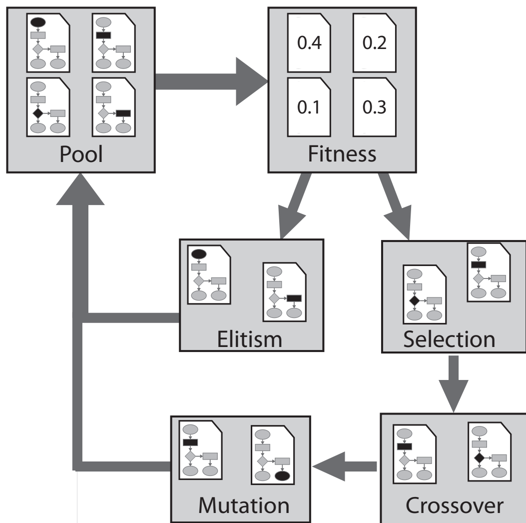


Figure 1. A simplified overview of the genetic algorithm

III. Genetic Algorithm

Genetic algorithms are a particular class of evolutionary algorithms that use techniques inspired by evolutionary biology such as inheritance, elitism, mutation, selection, and crossover. They are based on a probabilistic, meta-heuristic approach to search. In our approach, we use a genetic algorithm in order to produce mutations with a higher average impact which are nonetheless not detected by a test suite.

To this end, a pool of mutated versions of the original program is created (Section III-A), from which a start population is chosen. Then for every mutant its fitness is computed (Section III-B). The fittest mutants are chosen for the elitism set (Section III-D), which gives the best individuals unchanged to the next generation, while the other part of the mutants is found via heuristic-based selection. Some of the selected mutants then perform a crossover (Section III-E) where they exchange certain properties. All the selected and crossed individuals can undergo mutation (Section III-F), where some of their properties are changed randomly. The mutants of the elitism set and the mutated mutants are combined into one set that forms the origin of the next generation. The final output of the algorithm is a set of mostly undetected mutations, with a high unique impact (see Section III-B). Figure 1 gives an overview of the algorithm, and the following sections describe the steps in more detail.

A. Setup

Each mutant forms an individual of the genetic algorithm, consisting of three *genes*: The *location* of the mutation (meaning the class where it is applied), the *mutation operator* itself and an optional *integer parameter* (to determine either the arithmetic operator or the constant to be used in the mutation).

Before the algorithm itself starts, a pool of mutants of the original program is generated (cf. Figure 2). To achieve a sufficient number of mutants, two of the original JAVALANCHE mutation operators, the "Replace numerical constant" and the "Replace arithmetic operator" mutation operators were expanded by introducing a parameter that can take different values. From this pool, the start population of the algorithm is drawn randomly.

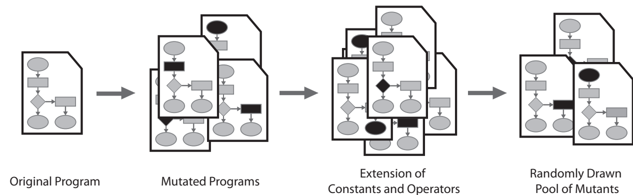


Figure 2. Initialization of the mutant pool

B. Fitness Function

The fitness function is constructed with the aim of reaching a maximum impact of those mutants that remain undetected. At the same time it should focus on an equal distribution of the mutations over the whole program to avoid accumulation of mutations. Using the coverage impact alone is not sufficient, as there might be parts in the program where a high impact can be reached for many mutations, whereas in other parts only a few mutations cause merely slight changes at all.

Therefore, we define the *unique impact*, which offers at the same time a way of taking into account the influence of other mutations of the same generation, so that different mutations that are causing (partly) the same changes in the program execution have to share their impact. Mutations that are detected shall not be allowed to reproduce, and so, disregarding their unique impact, they are assigned zero fitness.

The unique impact for each mutation of a generation can be computed as follows: Collect the code coverage for all individuals of the generation. Now, for every method that was changed by at least one mutation, count the number of mutations that change it. For a method m this value is given by the function $c(m)$. The unique impact i of a mutation M is now:

$$i(M) = \sum\{\tfrac{1}{c(m)} \mid \text{m is changed by M}\}.$$

The complete fitness function reads:

$$f(M) = \begin{cases} i(M) \text{ if M is not detected} \\ 0 \;\; \text{if M is detected} \end{cases}$$

Figure 3. An example of mutants with impact on different methods

Figure 3 shows a small example of three mutants, where each mutant changes some of the methods A, B, C, D and E. When we compute the value m_i for each method, we get: $c(A) = 2$, $c(B) = 1$, $c(C) = 1$, $c(D) = 2$, $c(E) = 3$. The unique fitness for mutant 1 is $\frac{1}{2} + 1 + \frac{1}{2} + \frac{1}{3} = \frac{7}{3} \approx 2.33$, for mutant 2: $\frac{1}{2} + \frac{1}{3} = \frac{5}{6} \approx 0.83$ and for mutant 3: $\frac{1}{2} + 1 + \frac{1}{3} = \frac{11}{6} \approx 1.83$.

Obviously, mutant 1 has the highest unique fitness, as it has impact on more methods than the other mutants and is the only one to change method B, whereas mutant 2 is the least fittest as it only affects two methods, which are both already affected by other mutants.

C. Evolving the Next Generation

A single step in the algorithm proceeds as follows: First, every individual of the generation is evaluated using the fitness function as given above. Then, consecutively, the mutants that build the next generation are selected and evolved in several steps, at which we are going to look more closely in the following sections.

D. Elitism and Selection

The elitism rate determines the amount of mutations of a population which is given unchanged to the next generation in percent. The mutations with the highest quality according to the fitness function are passed on. Afterwards, the individuals that are going to be the "parents" of the rest of the next generation, are chosen. They are selected by using a so called *roulette wheel selection* [4].

E. Crossover

The crossover allows the mutations to exchange some of their properties. Each individual that was selected by the roulette wheel for the new population, will participate in the crossover with a given probability. These selected individuals are then arranged in pairs. For each pair of coupled individuals, the two "parent" individuals are split at a random position, and replaced by their "children", a recombination of the split individuals as can be seen in Figure 4.

Not in all of the cases in which a crossover takes place, a mutant that is applicable to the code is created. Those invalid mutants have of course to be discarded; in such a case, no crossover takes place and the original mutants are fed once more into the algorithm.

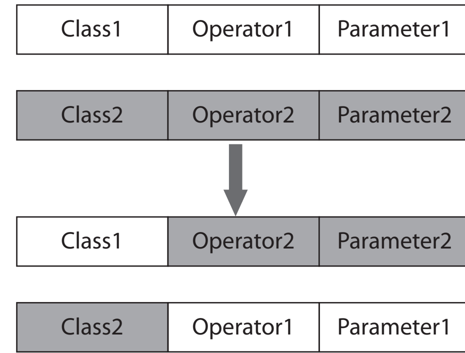


Figure 4. An example of two mutants performing a crossover

F. Mutation

The last step is the mutation phase. Its main function is to provide each generation with some new genetic material that has not been used before, so that the evolution of the population is not forced into one direction only.

A fixed probability is used to select the individuals that undergo a genetic mutation from the selected and then crossed-over set. Only those individuals that were not chosen to be handed over unchanged are considered. When a mutant is selected for genetic mutation, either its line number or parameter is mutated. Line numbers are replaced with a feasible line number from the near environment (i.e. a mutation of the same type within a range of $\pm$ 5 lines), and parameters are replaced with a random value.

During the selection, crossover and mutation phase, some mutants may be drawn or produced multiple times. This, however, stands against our aim of an equal distribution of mutations regarding their location over the program code, and therefore all mutations that occur more than once in the newly created generation are removed (i.e. only one copy of the mutation is kept) and replaced with new mutants from the initially generated pool. This has the side-effect of bringing more new genetic material to the algorithm and so widening its search area.

G. Termination and Output

The algorithm stops after an initially fixed number of steps. An alternative would have been to depend the termination of the algorithm on the quality of a generation, but as it is not sure if a population ever reaches this limit we dropped this possibility. The output of the algorithm is a ranking of the mutants of the last generation produced, giving the undetected mutants with the highest impact first to the user.

IV. Implementation

The evolutionary algorithm was directly included as a package into the JAVALANCHE source code. The main part consists of two steps: Step 1 creates the start population (pool) of the algorithm; Step 2 evaluates each generation

and creates the next one. To create the initial pool, mutants are drawn randomly from the JAVALANCHE mutant database and then extended by giving an additional parameter to them. For reasons of efficiency, in each step of the algorithm, only the non-elitism mutants are run to compute their traces. This is sufficient for computing the impact of each mutant as the traces of the mutants that remain unchanged due to elitism stay the same as in the step before. The tracing itself is done with the built-in coverage tool of JAVALANCHE.

V. EVALUATION

For the evaluation of our approach, we were interested in the following questions:

- Does our genetic algorithm help to produce a set of mutations that have a high unique impact and are not detected?
- How does the choice of parameters affect the results?
- How do the results compare to the results of regular mutation testing?

To answer these questions, we applied our genetic algorithm approach to JAXEN, an open source XPath Engine. JAXEN is a medium sized project of about 12.000 lines of source code that comes with a JUnit tests suite which consists of 690 test cases.

In order to answer our first question, we look at the average unique impact, the average impact and the number of undetected mutations for applying our approach using the default parameters of our algorithm (see Section V-A). The effects of varying some of the parameters are investigated in Section V-B. Finally in Section V-C, we compare our approach against the output of an original run of JAVALANCHE that measures the coverage impact for all mutations.

A. Evolution of a Population

This section explores the evolution of a population, looking at how the impact, fitness and number of undetected mutants, developed while running the algorithm. For the experiments we used the default values of the algorithm which are: An elitism rate of 0.7, a crossover rate of 0.5, a mutation rate of 0.2, and a population size of 100 (which we believe is a convenient output size for the user).

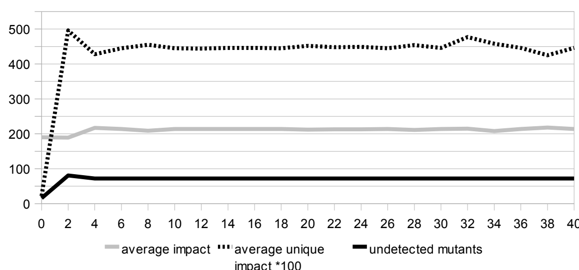


Figure 5. Run of the genetic algorithm with an elitism rate of 0.7, a mutation rate of 0.2, crossover rate of 1 and population size of 100

The results for this experiment are shown in Figure 5. The x-Axis give the number of steps in the genetic algorithm and the y-axis gives a scale for the different values. The average impact of a population, shown as a grey line, does not increase much while running the algorithm, on the average from 190 to 210, which is at the most 10 to 20 percent. This is due to the fact that detected mutants, which are still contained in the population, have an on average higher impact than undetected mutants. Nevertheless, the quality of the undetected mutants increases a lot: The average impact of the undetected mutants grows from 34.8 to 179, the highest-ranked mutants in the start populations have an impact of approximately 55, after 30 steps the highest-ranked mutants the highest impact reached for an undetected mutant is 650. The number of undetected mutants evolved from a pool of 100 mutants increases from 16 in the original pool to 72, that is three quarters of the whole pool, in about 4 steps. In general we can observe that in the first few steps the algorithm performs very well, but after that no significant improvements are made.

The genetic algorithm produces a set of mutations that consist of mostly undetected mutations that have a high impact and are well spread throughout the program.

B. Algorithm Parameters

This section explores the effects the single parameters have on the results of running the genetic algorithm.

1) Elitism Rate: The elitism rate determines the amount of mutations of a population which is given unchanged to the next generation in percent. These mutations are chosen by their quality, such that the most valuable mutants, i.e. those with the highest fitness, are passed on.

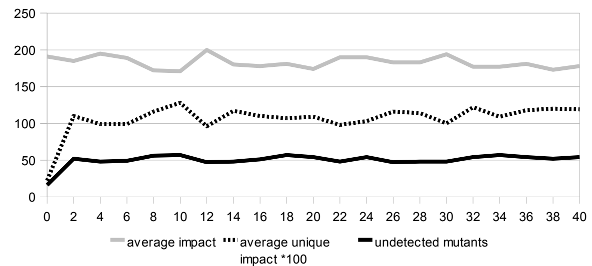


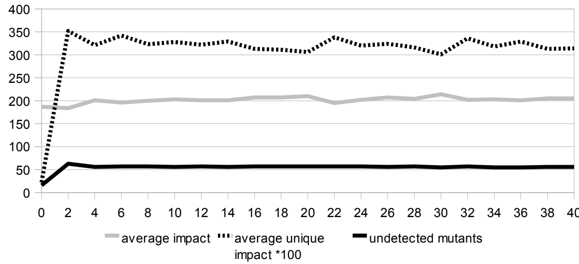
Figure 6. Elitism 0.2

For our experiments we tried values from 0.2 (Figure 6) to 0.7 (Figure 5) with clear results. When we compare the results we can see that with the elitism rate also the average impact, the average unique impact, and the number of undetected mutants increases. Thus, it can be followed that a higher elitism rate leads to better mutants in a generation, regarding both impact and undetectability.

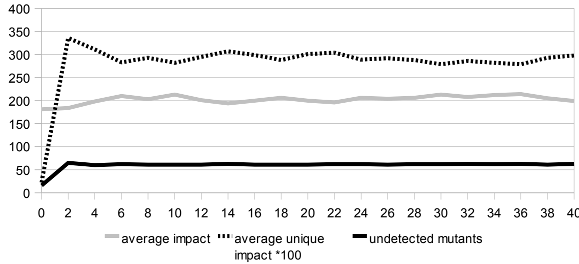
2) Crossover Rate: When performing a crossover, two mutants can exchange certain properties in order to create new genetic material. For our experiments we applied

Table I
RESULTS FOR A RANDOMLY DRAWN SET OF MUTANTS FROM JAVALANCHE AND A POPULATION BEFORE AND AFTER 30 STEPS OF EVOLUTION, BOTH OF SIZE 100

	JAVALANCHE	Start Pool	Genetic Algorithm
Average Impact	308	180	210
Average Impact of undetected Mutants	4.78	34.8	179
Impact of Best undetected Mutant	158	158	650
Percentage of undetected Mutants	9	16	72



(a) Crossover rate 1



(b) Crossover rate 0.5

Figure 7. Comparison of the crossover rates 1 and 0.5

crossover to the non-elitism mutants with a probability of either 100 or 50 percent.

As Figure 7 shows, there is no significant difference for the different crossover rates. Crossing mutants less often might preserve, similar to elitism, already good mutants instead of manipulating them. Nevertheless, crossover is necessary to find new, not yet used mutations for the next generation.

3) Mutation Rate: The main function of mutation is to provide each generation with some new genetic material that has not been used before, so that the evolution of the population is not forced into one direction only. Similar to the change of the crossover rate, changing the mutation rate does not change the results visibly. A higher mutation rate nevertheless decreases the number of double mutations that have to be replaced after each step in the algorithm and therefore is still useful.

C. Comparison with JAVALANCHE

This section compares the final results of the algorithm presented in the section before with the output of JAVALANCHE.

Table I gives the result for all mutants produced by JAVALANCHE (column 2), a randomly chosen set of mutations from of mutant start pool drawn for running the genetic algorithm (column 3), and the results after 30 steps of the algorithm (column 4).

The average impact of all JAVALANCHE mutations is 308 which is higher than the average impact of the start population (180) and of the population that has been evolved for 30 steps (210). This can be explained by the fact that on average, undetected mutants have a lower impact than detected ones. However, this is not true for the average impact of the undetected mutants: The undetected mutants generated by the genetic algorithm have an average impact of 179, whereas for JAVALANCHE this value is 4.78. Already the use of a diversified set of mutation operators as in the case of the pool has significant effects: The average impact of the mutations from the pool lay at about 34.8.

A population after an evolution of 30 steps contains up to 72 % undetected mutants, that is nearly three quarters of the population. On the contrary, the output set of JAVALANCHE contains only 10 % of undetected mutants.

When comparing the quality of the best mutants of a randomly drawn output set of JAVALANCHE and a population evolved for 30 steps, *the differences are huge*: The by far best mutation of the first set has an impact of 158, whereas the result set of the algorithm contains two mutants with an impact of 650 each.

The genetic algorithm produces a set with more undetected mutants which have a higher impact than drawing a random set from the regular output of JAVALANCHE.

D. Threats to Validity

There are several possible threats to the validity of this work. Those are:

- *Threats to external validity* concern the generalization of the results. As the method was only examined on one project, JAXEN, generalization of the results can not be claimed.
- *Threats to internal validity* may be caused by the relatively small number of mutants and mutation operators available. The algorithm was run with a total number of 6550 mutants made available by JAVALANCHE and 4 different mutation operators. This limitation might hinder a broader evolution of the mutants.
- *Threats to construct validity* concern the appropriateness of the measures used. The definition of the fitness

function was based on the assumption that the most valuable and desirable properties of a mutant are its undetectability and a high impact. This may be a false assumption, and might create a bias towards a specific type of mutations.

VI. Related Work

The idea of using genetic algorithms to evolve mutations was also introduced in the following works: Adamopoulos et al. [1] introduced the idea of co-evolving a set of mutants and a set of test cases. Their aim was to show how selective mutation testing could be achieved without selecting mutation operators beforehand, but instead using genetic algorithms to evolve mutants and test cases to fit each other in a way that avoids generation of equivalent mutants. However, they only presented results of simulations of genetic algorithm runs, which, however, seemed promising. In our approach, also a genetic algorithm is used to evolve generations of mutants, but instead of avoiding equivalent mutants, the algorithm aims at producing undetected mutants with an impact as high as possible.

In their work on Higher Order Mutation Testing (HOM Testing), which combines two or more first order mutants in order to produce more subtle fault combinations, Jia and Harman [3] use a genetic algorithm to find subsuming HOMs—that are HOM for which tests that detect this HOM also detect all the first order mutations that it consists of.

Other works aim at making mutation testing more efficient or avoiding the generation of equivalent mutants. Weak Mutation Testing as proposed by Howden [2] is an approach to increase the efficiency of mutation testing by looking at differences in the results of components rather than the output of the program. Offutt and Pan [6, 7] show that detecting equivalent mutants is an instance of the feasible path problem and presented an algorithm based on mathematical constraints for automatically detecting equivalent mutants.

The impact metrics and the JAVALANCHE framework that our approach is based on was presented in earlier papers [9, 8, 10]

VII. Conclusions and Consequences

In this paper, we presented a genetic algorithm that aims to produce a set of mutations that is undetected, has a high impact and is well spread over the code. In our evaluation, we showed that applying the genetic algorithm improves from 20 to over 70 percent of undetected mutants in the output set, at which the average impact of the undetected mutants increases by a factor of 5. However, our evaluation also showed that these improvements are made in the first steps of the algorithm and in later steps the results do not improve anymore, and different crossover and mutation rates do not change the results visibly. This indicates a potential for improving the genetic algorithm. In contrast to treating single mutations as individuals of the genetic algorithm, we plan to use sets of mutations for evolution in the algorithm (similar to the approach of Adamopoulos et al. [1]). The algorithm runs on several sets of mutations with the aim of finding an optimal set of mutations that are the most useful ones for the programmer. Similar to the present approach, the fitness of the sets is based on the sum of the impact of the single mutations in the set. Crossover allows the exchange of mutations between two sets, while mutation removes mutants from the set or adds new ones, and also allows manipulation of single mutations in a set.

Besides improvements to the genetic algorithm, we also plan to apply our approach to multiple other projects, in order to get more representative results whether genetic algorithms can help in improving the results from mutation testing. Furthermore, we plan to investigate the runtime improvements compared to applying all mutants exhaustively more detailed, which is missing in this paper.

Acknowledgments. Gordon Fraser as well as the anonymous reviewers provided helpful feedback on earlier revisions of this paper.

References

[1] K. Adamopoulos, M. Harman, and R. M. Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Genetic and Evolutionary Computation—GECCO 2004*, volume 3103 of *Lecture Notes in Computer Science*, pages 1338–1349, 2004.

[2] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, 1982.

[3] Y. Jia and M. Harman. Higher order mutation testing. *Information and Software Technology*, 51:1379–1393, 2009.

[4] Z. Michalewicz. *Genetic algorithms + data structures = evolution programs (2nd, extended ed.).* Springer-Verlag New York, Inc., 1994.

[5] National Institute of Standards and Technology (NIST). Software errors cost u.s. economy $59.5 billion annually, 2002.

[6] A. J. Offutt and J. Pan. Detecting equivalent mutants and the feasible path problem. In *COMPASS '96: Proceedings 11th Conference on Computer Assurance*, pages 224–236, 1996.

[7] A. J. Offutt and J. Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification, and Reliability*, 7(3):165–192, 1997.

[8] D. Schuler, V. Dallmeier, and A. Zeller. Efficient mutation testing by checking invariant violations. In *ISSTA '09: Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, pages 69–80, 2009.

[9] D. Schuler and A. Zeller. Javalanche: Efficient mutation testing for java. In *ESEC/FSE '09: Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 297–298, August 2009.

[10] D. Schuler and A. Zeller. (Un-)covering equivalent mutants. In *ICST '10: Third International Conference on Software Testing, Verification and Validation*, pages 45–54, 2010.