Strong Mutation-Based Test Data Generation using Hill Climbing

Francisco Carlos M. Souza
Computer Systems Department
University of São Paulo
São Carlos, SP, Brazil
fcarlos@icmc.usp.br

Mike Papadakis
Interdisciplinary Centre for Security, Reliability and Trust
University of Luxembourg
Luxembourg
michail.papadakis@uni.lu

Yves Le Traon
Interdisciplinary Centre for Security, Reliability and Trust
University of Luxembourg
Luxembourg
yves.letraon@uni.lu

Márcio E. Delamaro
Computer Systems Department
University of São Paulo
São Carlos, SP, Brazil
delamaro@icmc.usp.br

ABSTRACT

Mutation Testing is an effective test criterion for finding faults and assessing the quality of a test suite. Every test criterion requires the generation of test cases, which turns to be a manual and difficult task. In literature, search-based techniques are effective in generating structural-based test data. This fact motivates their use for mutation testing. Thus, if automatic test data generation can achieve an acceptable level of mutation score, it has the potential to greatly reduce the involved manual effort. This paper proposes an automated test generation approach, using hill climbing, for strong mutation. It incremental aims at strongly killing mutants, by focusing on mutants' propagation, i.e., how to kill mutants that are weakly killed but not strongly. Furthermore, the paper reports empirical results regarding the cost and effectiveness of the proposed approach on a set of 18 C programs. Overall, for the majority of the studied programs, the proposed approach achieved a higher strong mutation score than random testing, by 19,02% on average, and the previously proposed test generation techniques that ignore mutants' propagation, by 7,2% on average. Our results also demonstrate the improved efficiency of the proposed scheme over the previous methods.

1. INTRODUCTION

Mutation Testing (MT) is a fault-based criterion originally proposed by Hamlet [10] and DeMillo et al. [6]. It works by injecting simple faults into the program under test (PUT) to obtain faulty versions of the program called mutants. Test data are then generated to reveal these faults, i.e., distinguishing the outputs of the mutated programs from the original one. If a mutant produces different outputs from the original program for a given test, then, the mutant is named "dead", otherwise, it is called "alive". In practice, some mutants produce the same output as the original program for all possible tests. This mutant is termed as equivalent, i.e., test data to detect this mutant are non-existent. The test quality is measured by the mutation score. It is defined as a ratio of the number of killed mutants to the total number of nonequivalent mutants, which indicates how adequate a test data set can be.

Other important aspects of mutation testing are the concepts of strong and weak or firm mutation that refer to the way that mutants are considered to be dead. Strong mutation (traditional mutation testing) aims at creating test data that cause the program state to propagate to the output, where failures are observable [5]. Thus, a test data is said to *strongly* kill a mutant when it causes the given mutant to produce an output that differs from the one generated by the original program [12].

In weak or firm mutation, the focus is shifted from the outputs of the internal part of the programs, i.e., in their components. The idea is that mutants are considered to be dead when they have differences in the internal program states. In weak mutation, instead of checking mutants after the execution of the entire program, mutants are checked immediately after the execution of the mutated code part. In firm mutation, they are checked at an arbitrary intermediate point of the execution, i.e., not directly after the execution of the mutated statement.

Among several testing activities, test data generation is one of the most critical and labor-intensive. Most often in the industry, test data generation is performed manually. However, this process is time-consuming, expensive and susceptible to errors. For several years, great effort has been devoted to studying this issue, but, the existing techniques are not yet quite satisfactory.

In general, test data generation is an undecidable problem, meaning that it cannot be completely solved. Nevertheless,

SBST '16 May 16–17, 2016, Austin, TX, USA

DOI: 10.1145/1235

this does not mean there is no algorithm that can find a plausible but partial solution to satisfy a specific test goal. In order to come up with an efficient heuristic, the main challenge in mutation testing consists of identifying an adequate test data set that maximizes the number of nonequivalent mutants killed. In collaterally obtaining a high mutation score, the manual effort of analyzing equivalent mutants can be reduced.

Several publications have appeared in recent years documenting different techniques for mutation testing. Although some of these studies address mutation testing, little attention has been given to test data generation in the same context. According to the study performed in 2012 by Souza et al. [25] only 19 publications use techniques for test data generation to kill mutants. Among these publications, only three of them are for C programs and only two studies [9], [13] employ search-based techniques.

This paper introduces an automated approach for generating test data by combining weak and strong mutation approaches via a search-based technique. Our approach extends the fitness function scheme suggested by Papadakis and Malevris [23] to strongly kill mutants for java programs. In our approach, we used the AUgmented Search-based Test-INg (AUSTIN) tool [16], which is the current state-of-the-art search-based test generation tool for C. It combines test data generation based on structural criteria with a search-based technique.

To deal with mutants, we extended AUSTIN to integrate with Proteum [4], a mutation testing tool for C. To perform the search, we use hill climbing, in particular, the Alternating Variable Method (AVM) [16], as implemented by AUSTIN. AVM was found to be quite effective and efficient in generating test cases for branch coverage [14], and hence, we believe that it is appropriate for mutation testing as well.

The original fitness function of AUSTIN is composed of two metrics: (i) approach level and (ii) branch distance. In addition to these metrics, we implemented and evaluated our fitness function involving three parts, the Reach Distance (RD), the Mutation Distance (MD) and Impact Distance (ID). The first part of the fitness, RD, guides the search towards the mutation point. The second one, MD, towards infecting the program state at the mutation point, i.e., making a difference in the execution between the two program versions, and the third one, ID, to propagate, i.e., manifest, the corrupted state in the program output.

The proposed approach uses a novel fitness function that incrementally guides the process to generate test cases that reach, infect and kill the considered mutants. The difference from the previous work is that it incrementally aims at strongly killing mutants, by focusing on mutants' propagation, rather than mutant infection. Thus, the proposed fitness function measures how close are the candidate test cases in making a divergence at every executed statement. Previous work, i.e., [8], [13], [7], measures the number of predicates that diverge and not the closeness of making the predicates diverge.

The evaluation of our approach was performed with two independent experiments in order to verify the effectiveness and computational cost of the approach for generating test data using the proposed function fitness. We also evaluated the contribution of every part of the proposed fitness function, i.e., RD, MD and ID parts, with respect to its effectiveness in strongly killing mutants.

Overall, the contributions of the present work can be summarized into the following points: *(i)* a fitness function to support search-based testing for strong mutation, using a novel ID scheme; *(ii)* an extension of an existing automated test data generation tool that uses branch coverage and search-based technique to produce mutation-based test data to kill weak and strong mutants; and *(iii)* an empirical assessment of the effectiveness and cost of the fitness function scheme used by our approach. We also compare and demonstrate that we outperform random testing.

The remainder of this paper is organized as follows: Section 2 details the automated approach proposed. Section 3 reports research design of experiments conducted. Section 4 presents the results obtained and discusses the benefits, relevance and limitations of the proposed approach and threats to validity. Section 5 shows related work for the present one. Finally, in Section 6 the conclusions and future directions are discussed.

2. APPROACH DESCRIPTION

The proposed approach attempts to automate the test generation process by searching for test data that can kill mutants. We target C programs and strong mutation. The idea of the approach is to produce test data sets using a fitness functions scheme based on weak and firm mutation for strongly killing mutants. An overview of the approach is presented in Figure 1.

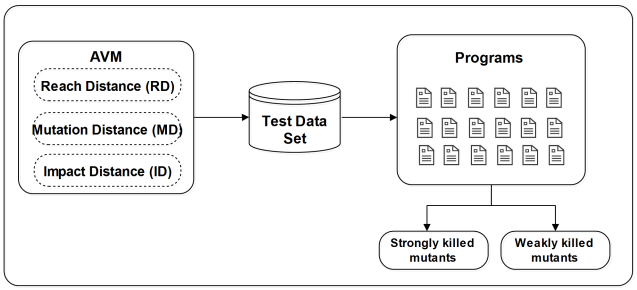

Figure 1: Approach for Mutation-Based Test Data Generation

Our approach uses the following metrics in the fitness function scheme: *(i) Reach Distance (RD)*, *(ii) Mutation Distance (MD)*, and the *(iii) Impact Distance (ID)*. These metrics are used to measure the adequacy of test data and direct the search process to an optimum in the search space.

The main contribution of the present approach is the handling of the mutants' propagation condition. Generally to kill a mutant, test data must: a) reach the mutated statement, b) infect the program state and c) propagate the infection to the program output. Our fitness function aims at guiding the search towards the abovementioned three conditions. Each one of them is handled by different parts of the fitness function: a) by RD, b) by MD and c) by ID. To kill a mutant it needs to incrementally do RD, MD and ID.

In this paper, the weakly killing mutants activity is handled as a covering branches problem. Assuming each mutant as a branch, it is possible to generate test data to cover a specific mutated statement and employ an existing structural testing tool for carrying out weak mutation. In the following subsections, details of the approach are given.

2.1 Generating test data for mutation testing

Test data generation in mutation testing consists of identifying test inputs that maximize the number of mutants killed. Killing all nonequivalents mutants means that a high-quality test set has been obtained and an alive mutant indicates the inefficiency of a test set. The fundamental problem is to identify a small test data set (x_1, x_2, x_3,..., x_k), so that, when it is executed over the set of mutants (M_1, M_2, M_3, ..., M_n) it is obtained the highest number of mutants killed [1].

Nevertheless, finding an adequate test data set is a complex activity, firstly due to the input domain size of a program (often, it is huge or even infinite) and after, the required effort for a mutant be killed. There are mutants Hard to Kill (HTK) and other Easy to Kill (ETK). For the first, only a few test cases in the input domain cause the mutant to fail. For the second, a large number of test cases exist that kill the mutant, so they are easier to find.

According to May [17] there are three possible scenarios for the relationship between the test sets that kill easy and hard mutants, Figure 2. It is important to highlight that this categorization is based on the difficulty of killing a mutant, differently of the weak and strong mutation concepts aforementioned.

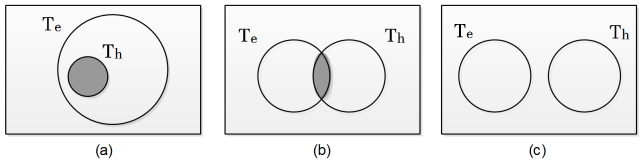

Figure 2: The relationship between the test sets that kill easily and hardly a mutant.

1. The test set Th is a subset of the test set Te (Figure 2 (a)). Where, Th and Te are test sets that kill hardly and easily mutants, respectively.
2. The test sets Th and Te are intersection (Figure 2 (b)).
3. The test sets are mutually exclusive (Figure 2 (c)).

Usually, test data are generated at random until a particular test objective has been achieved. However, in practice, in most of the cases, a random search is poor, time-consuming and it has no guarantee of finding a test data set to reach a certain part of the program or killing some specific mutants [11]. The use of the search-based and metaheuristic techniques guided by a fitness function can provide the sought high-quality test set, due to its emergent features to explore different points in the search space and improve a solution from the other one.

In this paper, we use the AUSTIN tool for test data generation. It aims only at covering program statements or branches by using a Hill Climbing (HC) technique known as the Alternating Variable Method (AVM). AVM was initially proposed by Korel [15] and shown to be effective for the structural testing of programs. AUSTIN combines AVM with a constraint-based approach that generates symbolic inputs, i.e., pointer inputs.

AUSTIN handles primitive data types and pointer variables. Pointer inputs are initially set to NULL and a symbolic process is performed. AUSTIN uses the CIL (C Intermediate Language) [20] to guarantee that the remaining constraints over memory locations are of the form $x = y$ and $x! = y$, where x or y can be the constant null or a symbolic variable indicating a pointer input. Initially, a symbolic path condition is built, describing the path taken by the concrete execution. The path condition may contain constraints over arithmetic and pointer type inputs. Furthermore, it describes the flow of execution taken by a concrete input vector, which took an infeasible path regarding the target branch. From this point, AUSTIN produces an equivalence graph of symbolic variables (the target of a pointer), which is adopted to solve pointer constraints. A graph-based process is performed to verify that the constraints over the pointer variables are feasible, and finally, the concrete pointer inputs are generated from the equivalence graph [16]. Pointer constructs, not related with inputs, are handled as integers. This has a minor influence on the effectiveness of the approach since they only depend on the followed paths.

We use the implementation of AUSTIN as the basis to generate test data and compute our fitness function. Since it provides a structure and metrics based on branch coverage and we target strong mutation, we extended it to operate with Proteum. We also extend it to use a new fitness function that direct the search towards maximizing the mutants' impact. This is described in Section 2.3.

2.2 Search-based Software Testing

Search-based Software Testing (SBST) is the process of automating a testing activity utilizing different techniques, such as local and global searches, metaheuristics and evolutionary algorithms. They offer good solutions for complex problems at a reasonable computational cost compared, for example, with random techniques. Test data generation is a classic problem for SBST since it cautiously identifies a subset of input data from a set that has all possible inputs to a given program.

Many researchers proposed methods and tools for SBST by focusing on different techniques such as mutation testing, structural testing, and functional testing. Additionally, great efforts have been devoted to the test data generation activity using search-based techniques. An empirical study performed by Harman and McMinn [14] indicates that an HC algorithm can be more effective than other search-based techniques for this context and, the most widely used in this context AVM has been employed on branch coverage criteria.

The AVM is a method based on a neighborhood search that can be quite powerful when combined with an adequate fitness function. The search is composed by two steps known as *exploratory moves* and *pattern moves*. The first starts by selecting random values for the input variables of the program. Then, the first variable is taken and the other variables' values are kept until the solution is found. After that, a search is performed for each input variable in turn until finding a solution and no further fitness improvement. Exploratory moves consist in adjustments in the input variables' values through small increments and decrements (usually, 1 for integer and 0.1 for real variables).

When the exploratory moves are able to be identified, a direction for which there are improved solutions, the pattern moves begin. This step is performed through large moves, i.e., increases in the adjustment values to accelerate the improvement of objective value and to achieve higher search space coverage.

2.3 Fitness Function

The key ingredient of Search-based approaches are the fitness function which guides the test generation process towards the promising areas of the search space. Thus, more effective functions lead to significantly better results [14], [18]. The fitness function must be defined according to problem features, i.e., the objective is represented by a particular feature capable of evaluating candidate solutions, in terms of their goodness and suitability for solving the problem at hand. Here, it should be noted that in order to apply effectively a search-based approach it is required to have a good fitness function, but it is not always an easy task [3].

In this paper, we utilized a fitness function scheme that combines four metrics. The first two were proposed by Wegener [26] called Reach Distance (RD) used for branch coverage criteria in structural testing and they include approach level and branch distance. The third one introduced by Papadakis [23] is called Mutation Distance (MD) and the fourth one is named Impact Distance (ID) which forms one of the main contributions of this paper.

In structural testing, one of the most effective fitness function schemes is the one based on the use of the approach level and branch distance. We use these features to compose our fitness function.

The *approach level* measures how close test data are in covering the targeted statement. It is computed using the number of the target mutant's control dependent nodes that were not executed by the test data [26]. On the other hand, the *branch distance* estimates how close an alternative branch is to be taken, i.e., switching a true branch to a false one or the opposite, and it is computed using the values of the variables or constants in the condition statements at which the execution flow was diverted from what was the target branch. This measure is computed following the expressions presented in the Table 1.

Table 1: Branch fitness

Expression	True Branch	False Branch
a == b	abs (a - b)	a == b?k : 0
a != b	a != b? 0 : k	abs (a != b?a - b : 0)
a < b	abs (a < b?0: a - b + k)	abs (a < b?a - b + k : 0)
a <= b	abs (a <= b?0 : a - b)	abs (a <= b?a - b : 0)
a > b	abs (a > b?0 : a - b + k)	abs (a > b?a - b + k : 0)
a >= b	abs (a >= b?0 : a - b)	abs (a >= b?a - b : 0)
a || b	min[fit(a), fit(b)]	fit(a) + fit(b)
a && b	fit(a) + fit(b)	min[fit(a), fit(b)]

Mutation Distance (MD) measures the branch distances on mutants. In other words, it measures how close the test data are in exposing a difference of the mutant statement. To compute this measure, first, it is necessary to quantify the distance that makes a change between the mutant and the original program predicates. This distance is computed according to the expression (1). This fitness function is a generalization of the study proposed by Bottaci [2], in which a genetic algorithm fitness function was described using the conditions proposed by Demillo [5]. For the cases of compound statements, it considers making differences between the whole program statements and not only at the mutated condition.

$$(Opred == T \&\& Mpred == F) || \\ (Opred == F \&\& Mpred == T) \quad (1)$$

where O and M, represent the original and the mutant predicates fitness calculations.

According to the branch distance fitness calculations shown in table 1, the fitness related to the expression (1) is called *predicate mutation distance (pmd)* and is defined according to the expression (2). When is not possible to satisfy the expression (1), the pmd determines how far it is to make the predicate behave distinctly at the mutation point.

$$pmd = min[Tfit(O) + Ffit(M), Tfit(M) + Ffit(O)] \quad (2)$$

where $Tfit(O)$ and $Ffit(O)$ are the mutant distance for the original predicate true and original predicate false and $Tfit(M)$ and $Ffit(M)$ are the mutation distance for the mutant predicate true and mutant predicate false. If pmd is 0, the difference between the original and the mutated statement can be seen. However, if pmd is not 0, the test data was not suitable to propagate this difference to the outcome of the mutant statement.

Impact Distance (ID) aims at guiding the search towards identifying test data capable of exposing different program outputs, between the original and mutant programs. This function attempts to approximate the mutant sufficiency condition [5] by measuring the impact of the mutant on the program execution.

We measure the ID using the trace (statementNum) on both the original and mutant programs and the impact, which is computed (per statement) as follows: *i)* if the traces differ then the distance is 1 and *ii)* if the traces are the same it considers the branch distances computed by expression (3), where a and b are the branch distances from original and mutant programs.

In essence, the ID measures how close the tests are in making an impact at every predicate that is traversed. This is achieved by measuring the branch distances all the way from the mutation point to the output statement. Previous researchers, i.e., Fraser and Zeller [8], Fraser and Arcuri [7] and Harman et al. [13], only measure the number of predicates that diverge ignoring the branch distances on the predicates that do not have an impact. Hence, the search has no guidance towards increasing the impact. An analogous situation is when trying to cover branches without using branch distances, i.e., counting only the number of branches covered, as done in the early days of SSBST, i.e., [19], which is clearly less effective.

$$\textbf{Impact} = abs(a - b)/(abs(a - b) + 1) \quad (3)$$

Overall, the proposed approach uses the RD fitness function, leading the search to reach the mutated statement (reachability condition); MD, to induce a different behaviour at the mutation point compared to the original program in the same point (necessity condition); and ID to guide the search towards the program elements that can be impacted

and ultimately expose the mutant's program (sufficiency condition), according to the expression (4).

$$\begin{aligned} \textbf{fitness function scheme} &= RD + MD + ID \\ \textbf{RD} &= 2 * approachlevel + normalized(branchdistance) \\ \textbf{MD} &= normalized(PMD) \\ \textbf{ID} &= statementNum - \sum Impact \end{aligned} \quad (4)$$

3. EXPERIMENTAL STUDY

We conducted an experiments to analyze and evaluate the effectiveness and cost of the proposed approach for test data generation on set of C programs. It is noted that while the fitness scheme of (RD & MD & ID) has been used before with genetic algorithms, here we only consider the AVM method (since it was found to be quite effective and efficient [14]). Under the AVM approach, the fitness scheme proposed by all previous approaches, i.e., [7, 13] is equivalent to (RD & MD). This because, AVM does hill climbing and hence stops when the fitness cannot be improved.

We also evaluate the contribution of all the three parts of the fitness function scheme (RD & MD & ID) in generating mutation-based test data.

3.1 Goals and Hypotheses

The experiments described in this section empirically investigates the effectiveness and efficiency of the proposed fitness function. We are interested in measuring the contribution made by each one of the parts composing the proposed fitness function and the overall effectiveness of the proposed scheme. Thus, we seek to investigate the following Research Questions (RQs):

RQ_1: How effective is the proposed fitness scheme (RD & MD & ID) for generating and improving test data to kill mutants compared to random testing?.

In order to answer RQ_1, we compare the fitness scheme with a random test for generating of the test data. Herein, the test data generation is twofold. Consider that m contains all alive mutants and m' is the set of the remaining mutants, i.e., a set of mutants not dead from m. First, we generate and improve test sets using the fitness function scheme to achieve strong mutation adequacy. In this case, we generate the test data using RD to kill m; MD to improve the test data to kill m' and ID to improve the test data generated from MD in order to kill the remainder of m'. Second, we used a random testing employing the mutation score as the fitness function to assess the test data adequacy for this method. Finally, the strong mutation score and time (in seconds) of both cases were compared with each other. We repeat this experiment 10 times and compute the average of strong mutation adequacy. This question allows evaluating the degree of the test data adequacy using the proposed approach. Note that, the fitness scheme should be capable of achieving a higher adequacy than random testing. We have defined the following hypotheses for this research question:

H_0: There is no difference on strong mutation score between the proposed fitness scheme (μF) and random test data generation (μR), thus (H_0: $\mu F = \mu R$).

H_1: The proposed fitness scheme will achieves a better strong mutation score than random test data generation (H_1: $\mu F > \mu R$) or (H_2: $\mu F < \mu R$) for the opposite.

RQ_2: How effective is the proposed fitness scheme (RD & MD & ID) compared to the RD fitness function used in previous studies?.

In order to answer RQ_2, we compare the fitness scheme with RD. In this case, only the RD fitness function is used and the number of mutants weakly and strongly killed are computed. We also performed this experiment 10 times and compute the average of weak and strong mutation score. We had the following hypotheses for this research question:

H_0: There is no difference on strong mutation score between the proposed fitness scheme (μF) and Reach Distance (μRD), thus (H_0: $\mu F = \mu RD$).

H_1: The proposed fitness scheme will achieves a better score strong mutation score than Reach Distance (H_1: $\mu F > \mu R$) or (H_2: $\mu F < \mu R$) for the opposite.

RQ_3: How effective is the adopted fitness scheme (RD & MD & ID) compared to the (RD & MD) fitness function?.

In order to answer RQ_3, we compare the fitness scheme performed similarly in the RQ_1 with RD & MD. In using the RD & MD fitness function, first we used RD to generate a test set to try kill the mutants m. Then, we have used the MD to produce a new test set through the improvement of previous test data generated by RD, in order to kill the mutants m'. Herein, we also performed this experiment 10 times and compute the average of weak and strong mutation score. For this question, we have defined the following hypotheses:

H_0: There is no difference on strong mutation score between the proposed fitness scheme (μF) and Reach and Mutant Distance ($\mu RD\&MD$), thus (H_0: $\mu F = \mu RD\&MD$).

H_1: The proposed fitness scheme will achieves a better score strong mutation score than Reach and Mutant Distance (H_1: $\mu F > \mu RD\&MD$) or (H_2: $\mu F < \mu RD\&MD$) for the opposite.

RQ_4: How efficient is the proposed approach for test data generation to mutation testing?.

The efficiency of the approach was measured using the time for test data generation and the number of fitness evaluations. The time includes all times to generate test data and their improvement. The total time*(T)* was computed by expression (5).

$$\mathbf{T} = time(RD) + \sum time(MD) + \sum time(ID) \quad (5)$$

where *time(RD)* is the time of test data generation for original program. *time(MD)* is the sum of time test data improvement using MD for each alive mutant selected. Finally, *time(ID)* is the sum of time test data improvement using ID for each alive mutant selected. The time was recorded through the Linux time utility on a laptop with Intel Core i7 1.8 GHz CPU, 8GB memory in the Ubuntu 14.04 operating system. We define the following hypotheses for this research question:

H_0: The strong mutation score of Reach and Mutant Distance ($\mu RD\&MD$) and Reach Distance (μRD) are higher than the proposed fitness scheme (μF) in the same time, thus (H_0: $\mu RD > \mu F$ and $\mu RD\&MD > \mu F$).

H_1: The proposed fitness scheme (μF) will achieves higher strong mutation score than Reach Distance (μRD) and Reach and Mutant Distance ($\mu RD\&MD$) in the same time (H_1: $\mu F > \mu RD$ and $\mu F > \mu RD\&MD$).

3.2 Procedure of Experiment

To answer the RQs, we carried out the experiments, as follows: *(i)* generating and improving test data to kill the mutants using the proposed fitness scheme and to measure mutation score, time, and cost; and *(ii)* comparison of the fitness scheme with the fitness function previously proposed and random testing. These experiments were performed in four steps:

1. We chose 18 C programs P = (p_1, p_2,...,p_{18}) of different domains and varying sizes as experimental subjects. These programs varied in size from 1 to 7 functions, and from 9 to 49 lines of code, totalling 28 functions and 126 lines of code.

2. We used the PROTEUM tool [4] to compute the strong mutation score and to generate all the mutants M = (m_1, m_2,...,m_n), where m_i is the set of mutants for each program p_i. These mutants were produced by 12 mutation operators presented in Table 2.

Table 2: Relational and Logical Operators

Operators	Description
ORRN	Relational Operator by Relational Operator
ORAN	Relational Operator by Arithmetic Operator
ORSN	Relational Operator by Shift Operator
ORBN	Relational Operator by Bitwise Operator
ORLN	Relational Operator by Logical Operator
OLSN	Logical Operator by Shift Operator
OLRN	Logical Operator by Relational Operator
OLLN	Logical Operator by Logical Operator
OLAN	Logical Operator by Arithmetic Operator
OLBN	Logical Operator by Bitwise Operator
OLNG	Logical Negation
OCNG	Logical Context Negation

3. We used the AUSTIN tool adapted to mutation testing to generate the test data T = (t_1, t_2,...,t_n), which is guided by the proposed fitness scheme. The test data are generated iteratively until a test data gets an appropriate fitness value for a given function. For each program p_i the test generation process considered up to 1000 fitness evaluations per branch to select a test data. This step is divided into three sub-steps:

3.1. The test data are generated for the original program by AVM using the RD fitness function to achieve branch coverage, i.e., every branch in the original program is evaluated to true and false if it is possible. After that, test data are applied in the mutants: if an input executes the true branch for original program and false for the mutant or the opposite for a statement, the mutant has been weakly killed and then strong mutation scores are computed and the dead mutants are removed;

3.2. A mutant is selected from those that are live and an improvement in the test data is performed by AVM using the MD fitness function. If the test data achieve a fitness value equal to zero, it means that the mutant has been weakly killed, then the improved test data is executed in the remainder of the live mutants. Otherwise, if the mutant is alive, another mutant is selected. This step is performed until the largest number of mutants has been weakly killed. After this, the weak and strong mutations scored are computed and dead mutants are removed;

3.3. A mutant is selected from those that are alive and an improvement in the test data is performed by AVM using the ID fitness function. The test data with the smallest fitness value is used and applied in the alive mutants. Then, the strong mutation score is computed.

4. The total of mutation score is computed by the sum obtained in the previous sub-steps.

4. RESULTS AND ANALYSIS

In this section we answer the RQs presented in Section 3 from the analysis of results concerning the effectiveness and efficiency of the proposed approach.

Figure 3 shows the mutation score for fitness function scheme and random test against the subject programs. As the results show, the proposed approach achieved an improvement in the mutation score of 19,02% more than random testing (RQ_1). The results analyzed through the Wilcoxon test indicated that there was a significant difference in the strong mutation score between them, since z=-2.766, p<0.05 (the null hypothesis was rejected). Therefore, the results indicate that the proposed fitness scheme is more effective than random testing in the most subjects.

Additionally, even if the cost for random test data generations is lower than an intelligent approach, if the searching and improvement of an additional test data increase the mutation score significantly, thereby the cost of generating and executing can be justifiable. Compared with random generation, it can be noted (see Figure 3) that test data generation using AVM and fitness scheme is more efficient in the most of the subjects regarding strong mutation scores. Thus, the proposed approach can be seen as an option to be used in this context, since weak and firm mutation based test data generation is a cost-effective alternative for producing test data to strong mutation. Moreover, the search-based approach with a worthy fitness function provides a high potential for reaching these difficult parts of the PUT.

In Figure 4 the lines represent the average of strong mutation score obtained for each fitness function combination. We observed that the fitness scheme improves strong mutation score in all programs compared with RD fitness function (RQ_2). More importantly, the mutation score increases from the addition of a new fitness function for generating and improving of the test data. We notice in the results from of Wilcoxon test that there was a significant difference in the strong mutation score, since z=-3.724, p<0.05 (the null hypothesis was rejected). Therefore, the results indicate that the proposed fitness scheme is more effective than RD fitness function.

Table 3 shows the number of mutants (second column), number of equivalent mutants (third column), mutation score of random test (fourth column), weakly and strongly killed mutants by test data generation using RD and RD & MD fitness function (fifth and sixth column), the number of strongly killed mutants by RD & MD & ID fitness scheme in the seventh column and its strong mutation score in the last one. Weak mutation score for RD & MD is able to weakly kill around 76% of the mutants and for the strong score, around 66% (RQ_3). It is natural to expect that the weak score is greater than or equal to the strong score since a test data that strongly kills a mutant must be able to weakly kill it though not necessarily the opposite. The results obtained and analyzed using the Wilcoxon test shown that there was a significant difference in the strong mutation score, since z=-3.267, p<0.05 (the null hypothesis was rejected). Therefore, the results indicate that the proposed fitness scheme is more effective than RD & MD fitness function.

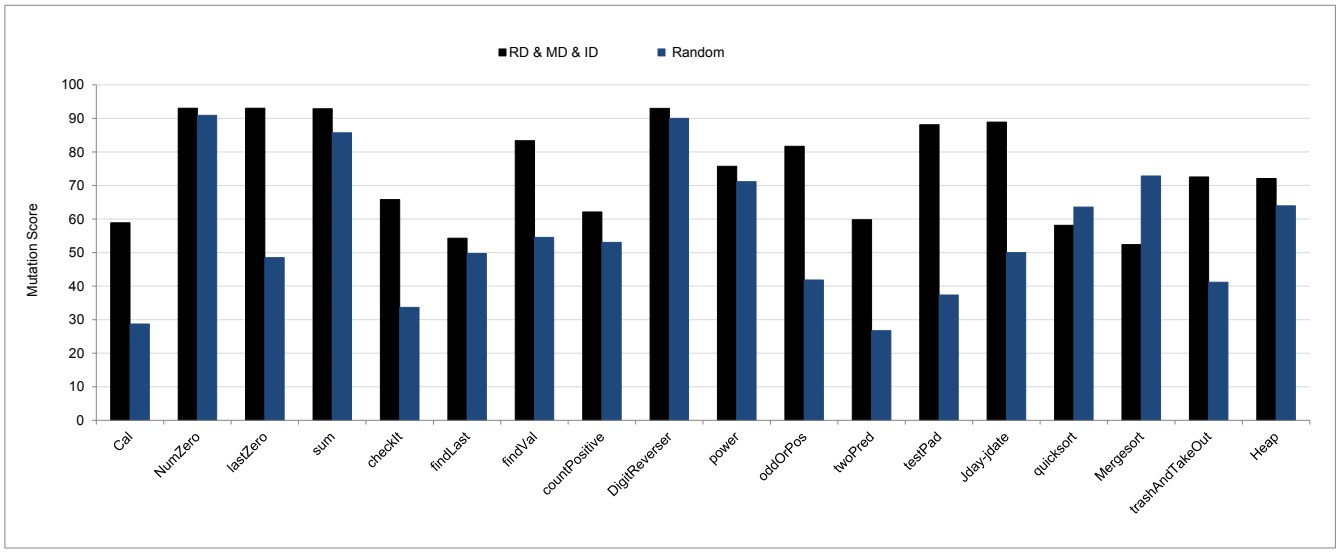

Figure 3: Fitness function scheme (RD & MD & ID) vs Random testing

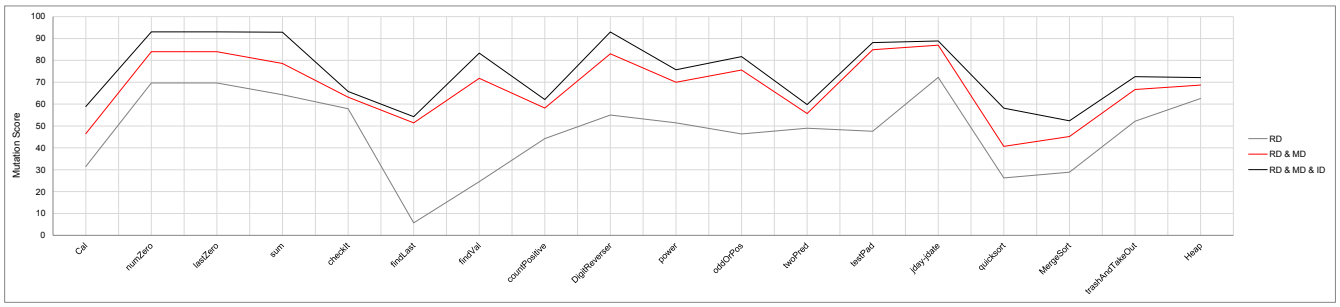

Figure 4: Strong Mutation Score of the fitness function scheme (RD & MD & ID), RD & MD and RD

Table 4 reports the results from a second experiment based on time and number of fitness evaluations required for generating and improving test data. The column two indicates the time in seconds, columns labeled RD & MD & ID, RD & MD and RD report the strong mutation score and the last column reports the number of fitness evaluations required. To answer RQ_4, this second experiment, was performed to measure whether the score obtained through the fitness function scheme at a particular time also could be achieved by RD & MD and only RD in the same time. The idea is to show that the fitness functions are complementary and they cannot achieve the best results individually.

We notice in the results from of Wilcoxon test that the null hypothesis was rejected i.e, RD and RD & MD are not able to achieve the same scores when executed with the time obtained by fitness function scheme (see Table 4), since z=-3.723 and z=-3.725 for $p<0.05$, respectively. Thus, the results indicate that the proposed fitness function scheme is more effective than the others.

It is noted that by the addition of a new fitness function the search is led to paths not reached by the initial test data. Furthermore, considering that target branch in the mutant that has been reached, this branch possibly will be traversed in the original program. Therefore, is possible to determine if the mutant has been killed or if the test data has failed.

In terms of cost, we cannot compare it to others approaches because there are only two approaches to C and with different experiment subjects, due to the cost depends mainly on the size of the program under test.

Also, we can consider that the number of fitness evaluations obtained is admissible compared with others approaches, because the fitness function scheme leads to a considerably higher number of killed mutants compared to the number of fitness evaluations. In general, the approach spends more time and fitness evaluations to improve a test data than to generate it because the difficulty of a new test data to kill a specific mutant, i.e., the more difficult the mutant is, the more fitness evaluation the approach is likely to spend.

Figure 5 indicates that more mutants can be weakly and strongly killable when the fitness function scheme is utilized. From this plot, we can see that the test data produced by different fitness functions that lead the search towards reachability, necessity and sufficiency conditions are more promising for killing mutants.

4.1 Threats to validity

Naturally, there are common threats to validity in experimental software engineering. This section presents an overview of the threats to validity and how they have been addressed. In our approach, we used 18 programs as ex-

Table 3: Mutants killed by fitness scheme (RD & MD & ID) and fitness functions

Programs	Mutants	Equiv.	Random	RD		RD & MD		RD & MD & ID	MS
				Weak	Strong	Weak	Strong	Strong	
Cal	127	19	28.7	49	34	79.6	50.2	64.2	58.9
numZero	36	3	90.9	26	23	33	27.7	30.7	93
lastZero	36	3	48.4	26	23	33	27.7	30.7	93
sum	18	4	85.7	12	9	17	11	13	92.8
checkIt	41	3	33.6	22	22	25	24	25	65.8
findLast	36	1	49.7	13	2	30.4	18	19	54.3
findVal	36	3	54.5	15.4	8.1	31.8	23.7	27.5	83.3
countPositive	36	3	53	17.6	14.6	23.4	19.2	20.5	62.1
DigitReverser	36	16	90	12	11	20.6	16.6	18.6	93
power	36	1	71.1	18.5	18	30	24.5	26.5	75.7
oddOrPos	110	28	41.8	50.4	38	84.4	62	67	81.7
twoPred	51	2	26.7	33	24	48.1	27.3	29.3	59.8
testPad	110	7	37.3	54	49	96	87.4	89.9	88.1
jday-jdate	36	0	50	26	26	32.5	30.9	32	88.9
quicksort	127	9	63.5	38	31	73	48	68.6	58.1
MergeSort	145	10	72.8	39	39	70.4	61	70.7	52.3
trashAndTakeOut	54	3	41.1	29.6	26.6	53.6	34	37	72.5
Heap	166	19	63.9	156	92	166	101	106	72.1

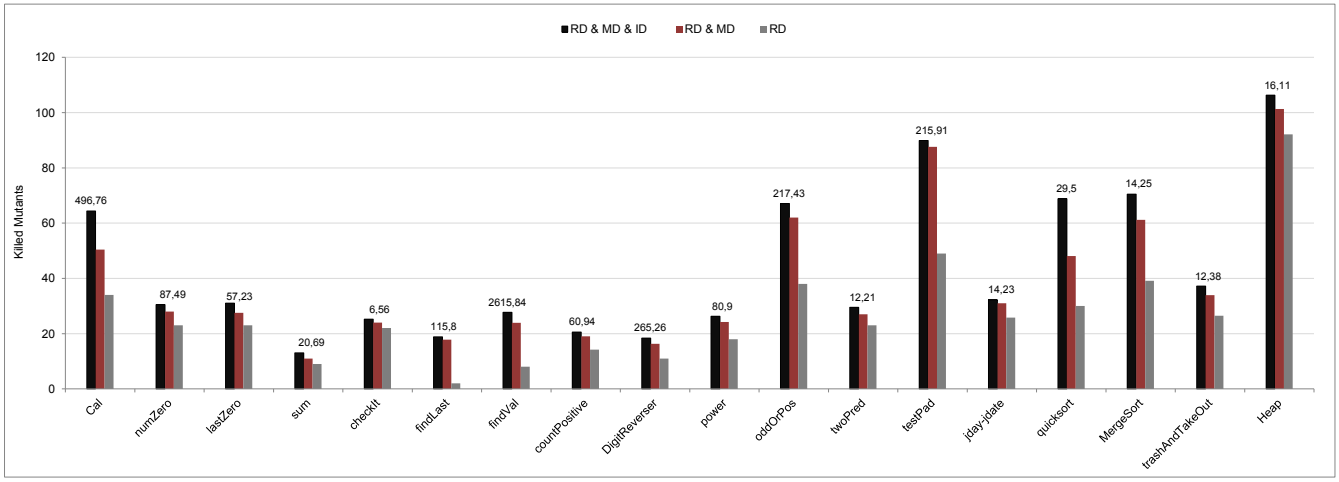

Figure 5: Comparison between fitness function scheme (RD & MD & ID), RD & MD and RD functions in the same time

perimental subjects and we cannot guarantee that they are representative. We try to solve this problem by selecting programs from different domains and with various sizes.

Another issue is the number of fitness evaluations because a low number can influence the quality of test data. In general, search-based techniques with low fitness evaluations have a premature convergence risk or impediments for the search to make substantial progress. It means that in some cases the adequate test set cannot even be found. On the other hand, a high number of fitness evaluations leads the approach to high time and computational costs.

In the last threat of this paper, the test data generated through AVM may have been found by luck when we utilized the fitness function scheme and its combinations since search-based techniques have no assurance of finding test data that kills the same quantity of mutants in different executions. We mitigated this issue by performing of a second experiment that aimed at generating test data to make sure that the incomplete fitness function RD and RD & MD have not obtained the same score than the complete fitness function scheme in a specific time.

5. RELATED WORK

Few studies have investigated techniques for test data generation attempting to reduce cost in mutation testing. Despite around 25 years of research from the first study, was identified only 19 studies in this context [25], that shows there is still a research gap in this area.

The first study addresses the Constraint Based Testing (CBT) proposed by Demillo and Offut [5], which introduced conditions that must be satisfied to kill a given mutant. From this study, these conditions referred to as reachability, necessity and sufficiency have became fundamental for mutation-based test data-generation approaches. The first condition, reachability, states that a test must achieve the mutated statement for a mutant to be dead. The second

Table 4: Time and Fitness evaluation

Programs	Time	RD & MD & ID	RD & MD	RD	Fitness Evaluation
Cal	496.7	64.4	50.4	34	550
numZero	87.5	30.5	28	23	1948.5
lastZero	57.2	31	27.5	23	1816.2
sum	20.7	13	11	9	4.5
checkIt	6.5	25.2	24	22	422.5
findLast	115.8	18.8	17.8	2	1565.6
findVal	2615.8	27.7	23.9	8	823
countPositive	60.9	20.6	19	14.2	664.7
DigitReverser	265.2	18.4	16.3	11	16.3
power	80.9	26.3	24.2	18	579.4
oddOrPos	217.4	67.1	62	38	57.4
twoPred	12.2	29.5	27	23	44.8
testPad	215.9	89.9	87.6	49	96.8
jday-jdate	14.2	32.2	31	25.8	75.3
quicksort	29.5	68.8	48.1	30	23.2
MergeSort	14.2	70.5	61.2	39.1	59.7
trashAndTakeOut	12.3	37.2	33.9	26.5	13.4
Heap	16.1	106.3	101.3	92.1	26.1

condition, necessity, requires that the execution state of the mutant program differs from of the original program after execution of the mutated statement. The third condition, sufficiency, requires that the incorrect state must be propagated to an output of the program. In addition, some studies such as [1], [24], [8], [13], [23], [7] proposed approaches using search-based techniques employing these conditions.

Ayari et al. [1] developed a approach using meta-heuristic, more precisely ACO, for automatic test input data generation that kills mutants in the context of mutation testing. This approach is guided by a fitness function that guides test to reach mutants with the hope that this will infect and kill them as well. Zhan and Clark [27] applied the same principles to kill mutants of Simulink Models. In [24], [22], a testability transformation that transforms the weakly killing mutant problem into a covering branches ones is used and demonstrates that existing tools can easily be adapted to kill mutants. Thus, search aims at reaching and infecting the mutants, handling reachability and necessity conditions. Along the same lines, in [7, 8], an evolutionary approach that automatically generates unit tests for object-oriented classes based on mutation analysis is proposed. As already explained in section 2.3, these aim at handling all three mutatnt killing conditions.

Papadakis and Malevris [21] proposed an approach that generates test inputs to kill weak and strong mutants using Dynamic Symbolic Execution (DSE). This tool transforms the original program under test into a meta-program, containing all the weak-mutant-killing constraints, and then the test inputs to cover all the branches the meta-program are generated through DSE. Thus, the DSE produces test data to kill the mutants automatically based on weak-mutant-killing constraints. To strongly kill mutants, the approach search the path space from the mutation point to the program output. More recently, Harman et al. [13] present a hybrid approach that combines DSE with HC for strongly killing both first order and higher order mutants. This approach uses DSE to generate weakly killing constraints and test data that satisfy them. Furthermore, HC is used to search test inputs that propagate infection to the output from the infection point. As stated before, contrary to our approach, Harman et al. counts the number of statements that have been impacted and hence, it fails to provide guidance towards increasing the mutants' impact.

This paper was based on the work of Papadakis and Malevris [23], which proposed a search-based technique for Java. The approach introduced here, uses a novel fitness function scheme to strongly kill mutants of C programs, by aiming at mutants propagation, ID. The new ID fitness scheme, contrary to [23], guides the search by measuring the closeness of making an impact at every traversed predicate, from the mutation point to the program output. Therefore, the proposed approach extends a previous study carried out by Papadakis [23] for a new language, new fitness scheme, ID in particular, and by examining the exact contribution of every part of the fitness, i.e., RD, MD and ID.

6. CONCLUSIONS

This paper presents an automated approach for generating test data for mutation testing. Our approach employs a Hill Climbing technique known as Alternative Variable Method guided by a fitness scheme to generate a test data set based on weak and firm mutation and in order to strongly kill mutants. Inspired by previous studies, we defined a fitness function scheme that generates and improves the test data generation process by measuring how close the candidate test data are to kill a targeted mutant.

Mutation testing is a powerful and effective technique that detects faults and measures the adequacy of the test suite. However, mutation testing is also computationally expensive, mainly because it has a high computational cost for generating test data and executing them against a large number of mutants. In this context, an adequate test data generation approach must be able to produce a high-quality test data set to kill all nonequivalent mutants and provide a cost reduction to mutation testing. Therefore, to be efficient, it needs to have a trade-off between the time of generation and mutation score.

Based on our experimental results, we find that the proposed approach produces effective test data able to strongly

kill the majority of mutants on C programs in a small amount of time. Thus, our results indicate that the proposed approach is promising, it increases the mutation score and this in only a small number of iterations. Also, it is noted that the mutated programs contain equivalent mutants, i.e, those mutants semantically identical, and they cannot be detected by a test data. Nevertheless, it is believed that they can assist the generation of high-quality test data, as well as those mutants difficult to kill.

Conclusively, a search-based technique is an excellent alternative to finding test data. However, it requires an appropriate fitness function to guide the search for different points in the search space as has been focused in this study. An appropriate fitness function can lead to the greatest opportunities to create an adequate test data set ensuring a good mutation score and cost reduction for mutation testing.

Future work is directed towards the following topics: *(i)* improvement of the fitness functions, ID part in particular, to generate test data towards exposing a mutant in an observable output; *(ii)* new large-scale experiments that can further establish our finding and demonstrate the importance of mutant propagation; and *(iii)* extend the approach to use additional types of mutants.

7. REFERENCES

[1] K. Ayari, S. Bouktif, and G. Antoniol. Automatic mutation test input data generation via ant colony. In *Proceedings of the 9th GECCO*, pages 1074–1081. ACM, 2007.

[2] L. Bottaci. A genetic algorithm fitness function for mutation testing. In *Proceedings of the ICSE*, pages 3–7, 2001.

[3] K. P. Dahal, S. Remde, P. Cowling, and N. Colledge. Improving metaheuristic performance by evolving a variable fitness function. In *Proceedings of the 8th EvoCOP*, pages 170–181. Springer, 2008.

[4] M. E. Delamaro, J. C. Maldonado, and A. M. R. Vincenzi. Mutation testing for the new century. chapter Proteum/IM 2.0: An integrated mutation testing environment, pages 91–101. Kluwer Academic Publishers, 2001.

[5] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17:900–910, 1991.

[6] R. A. DeMillo, F. G. Sayward, and R. J. Lipton. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[7] G. Fraser and A. Arcuri. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering*, 20(3):783–812, 2014.

[8] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th IISSTA*, pages 147–158, 2010.

[9] H. Haga and A. Suehiro. Automatic test case generation based on genetic algorithm and mutation analysis. In *Proceedings of the ICCSCE*, pages 119–123, 2012.

[10] R. G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng.*, 3(4):279–290, 1977.

[11] M. Harman. A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In *Proceedings of the ISSTA*, pages 73–83, 2007.

[12] M. Harman and Y. Jia. Analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.*, 37:649–678, 2009.

[13] M. Harman, Y. Jia, and W. B. Langdon. Strong higher order mutation-based test data generation. In *Proceedings of the 19th SIGSOFT*, pages 212–222. ACM, 2011.

[14] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Trans. Softw. Eng.*, 36(2):226–247, 2010.

[15] B. Korel. Automated software test data generation. *IEEE Trans. Softw. Eng.*, 16(8):870–879, 1990.

[16] K. Lakhotia, M. Harman, and H. Gross. Austin: A tool for search based software testing for the c language and its evaluation on deployed automotive systems. In *Proceedings of the 2th SSBSE*, pages 101–110, 2010.

[17] P. S. May. *Test data generation: two evolutionary approaches to mutation testing*. PhD thesis, University of Kent, 2007.

[18] P. McMinn. Search-based software testing: Past, present and future. In *International Workshop on Search-Based Software Testing (SBST)*, pages 153–163. IEEE, 2011.

[19] C. C. Michael, G. McGraw, and M. Schatz. Generating software test data by evolution. *IEEE Trans. Software Eng.*, 27(12):1085–1110, 2001.

[20] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. *Lecture Notes in Computer Science*, 2304:213–228, 2002.

[21] M. Papadakis and N. Malevris. Automatic mutation test case generation via dynamic symbolic execution. In *Proceedings of the 21st International Symposium on Software Reliability Engineering (ISSRE'10)*, pages 121–130, 2010.

[22] M. Papadakis and N. Malevris. Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. *Software Quality Control*, 19(4):691–723, 2011.

[23] M. Papadakis and N. Malevris. Searching and generating test inputs for mutation testing. *Springer Plus*, 2(1):1–12, 2013.

[24] M. Papadakis, N. Malevris, and M. Kallia. Towards automating the generation of mutation tests. In *Proceedings of the 5th Workshop on Automation of Software Test*, pages 111–118, 2010.

[25] F. C. Souza, M. Papadakis, V. H. S. Durelli, and M. E. Delamaro. Test data generation techniques for mutation testing: A systematic mapping. In *Proceedings of the 11th ESELAW*, pages 1–14, 2014.

[26] J. Wegener, A. Baresel, and S. Harmen. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.

[27] Y. Zhan and J. A. Clark. Search-based mutation testing for *Simulink* models. In *Genetic and Evolutionary Computation Conference, GECCO 2005, Proceedings, Washington DC, USA, June 25-29, 2005*, pages 1061–1068, 2005.