Automatic mutation test data generation via search-based technique

1st Francisco Carlos M. Souza
Universidade Tecnologica Federal do Paraná
Dois Vizinhos, Paraná, Brazil
franciscosouza@utfpr.edu.br

2nd Mike Papadakis
University of Luxembourg
Luxembourg
michail.papadakis@uni.lu

3rd Marcio E. Delamaro
Universidade de São Paulo
São Carlos, São Paulo, Brazil
delamaro@icmc.usp.br

***Abstract*—Mutation Testing is a powerful test criterion for detecting faults and measuring the effectiveness of test data sets. However, it is computationally expensive. The high cost comes mainly from the effort needed in generating adequate tests (tests that kill all the mutants) and by the existence of equivalent mutants. To overcome these problems, several techniques have been suggested. The main and most successful ones for this context are the search-based techniques. The present study proposes an automated test generation approach using hill climbing and a fitness scheme to produce test data that (R)each, (I)nfect, and (P)ropagate mutants. The main difference of this work with the previous papers is that it mainly focuses on the mutant propagation condition. Overall, the paper makes the following contributions: First, it defines and assesses two new impact measures (propagation condition) to compose a fitness scheme. Second, it reports on an empirical study that evaluates the proposed approach and compares with a genetic algorithm and random testing.**

***Index Terms*—Mutation testing, Search-based software testing, Test data generation**

I. Introduction

Mutation Testing is a fault-based criterion originally proposed for detecting faults and measuring the effectiveness of tests. Simple faults are injected into the program under test (PUT) and form program faulty versions, which are called mutants. The faults are introduced based on a set of mutation operators, which are syntactic rules that alter the source code of the PUT. After a mutant set is produced, the objective is to generate a test data set able 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 case, then, the mutant is named "dead", meaning that the fault has detected, otherwise, it is called "alive" if a test data is unable to detect the fault injected into the mutant.

There is a specific type of mutants that produce the same output as the original program for all possible tests. Such a mutant is termed equivalent, i.e., test data to detect this mutant are non-existent. Thus, without inspecting the internal composition of the source code, it is impossible to discover if the test data is unable to kill the mutant or if the mutant is equivalent to the original program.

Mutation testing techniques can be classified into three categories depending on the way that mutants are judged as killed. Thus, *i)* Strong Mutation: refers to the original case of mutation testing that mutants are considered as killed when they produce different program outputs from the original programs; *ii)* Weak Mutation: Mutants are killed if immediately after the execution of the mutated statements there is a state difference between the original and mutant programs; *iii)* Firm Mutation: there is difference between the states of the original and mutant programs at a later point after the execution of the mutation point.

Mutation testing is widely considered as effective powerful and costly. Cost comes from the quantity of mutants produced, the number of test data to kill the mutants, and the difficulty of identifying equivalents mutants. Among these issues, test data generation is one of the most critical, labor-intensive and susceptible to errors task, mainly if it is done manually. For these reasons automation in the test data generation is an important step to reduce mutation testing expenses and additionally increase the confidence in the testing. 动机

One approach to software test automation that has gaining great importance in the recent years is the Search-Based Software Testing (SBST), which have been used to resolve many problems and in particular on techniques for generating test data. SBST consists of formulating software testing as a search-based optimization problem and thus, using search-based optimization algorithms guided by a fitness functions to generate test data [1]. 适应性函数

In this context, the paper presents an automated approach for generating test data by combining weak, firm and strong mutation approaches via a search-based technique. Motivated by the previous studies that aim at strongly killing mutants [2], our approach extends the fitness function scheme by aiming at the mutant propagation. In our approach, we used the AUgmented Search-based TestINg (AUSTIN) tool [3], which is the current state-of-the-art search-based test generation tool for C. AUSTIN automatically generates test data for structural criteria based on 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) [3], as implemented by AUSTIN. AVM was found to be quite effective and efficient in generating test cases for branch coverage [1], 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. We extend this scheme for mutants involving three parts: Reach Distance (RD), 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 the corrupted state in the program output.

Additionally, we propose and investigate two impact measures, we assessed and compared them with the impact measure proposed in our previous work [5]. The aim of an impact measure is to guide the search for a test data by maximizing the number of changes in program behavior between the mutant and the original version. A high impact implies that a mutant is more likely to be killable than those with no impact. Thus, different impact measures can generally produce different outcomes, since a test data is able to expose changes in specific parts of the programs.

The proposed approach uses a fitness function that incrementally guides the process by generating 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 data in making a divergence at every executed statement. Previous work, such as [6]–[8] measures the number of predicates that diverge from the target and not the closeness of making the predicates diverge.

The evaluation of our approach was performed with an experiment, we verify the effectiveness and computational cost of the proposed approach for generating test data using two different impact measures. Overall, the contributions of the present work can be summarized into the following points:

- (i) a fitness scheme to support search-based testing based on weak and firm mutation for strongly killing mutants;
- (ii) an approaches to handling the mutants' propagation condition;

传播

- (iii) an empirical assessment of the effectiveness and cost of the fitness function scheme used by our approach.
- (iv) we also compare and demonstrate that our approach outperforms genetic algorithm and random generation.

The remainder of this paper is organized as follows: Section II details the proposed approach. Section III presents and details the experimental setup. Section IV discusses the results of the conducted experiment. Finally, Sections V and VI respectively present the related work and the conclusions of the paper.

II. Approach Description

Our test data generation approach aims to automate the test generation process directly from the source code of programs written in C. This paper addresses this issue by using a fitness function scheme based on RIP model and a search-based technique for generating test data extending the method presented in our previous paper [5]. An overview of the approach is presented in Figure 1.

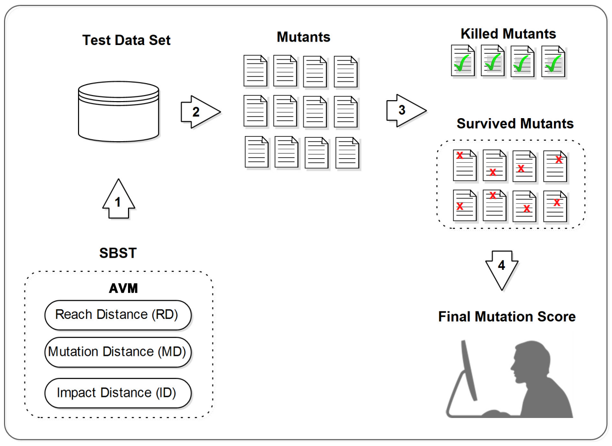

Fig. 1. Approach for Test Data Generation - The process is composed of the following steps: (a) generating test data; (b) executing the test suite against mutants; (c) removing the killed mutants; (d) mutation score evaluation; (e) improving the generated test data; and f) final mutation score. This process is repeated until reaching a predefined score threshold. 门槛

For test data generation, we use a Hill Climbing (known as Alternating Variable Method) combined with a fitness function scheme composed by the following metrics: *(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 guide the search process to an optimum in the search space. metrics的主要作用

适应度函数方案

Commonly to kill a mutant, test data must: a) **R**each the mutated statement, b) **I**nfect the program state and c) **P**ropagate the infection to the program output. Thus, our fitness function is used to adjust the search direction to find a test data set which can achieve the maximum possible mutation score. Each one of the above mentioned three conditions is handled by different parts of the fitness function: *(i)* by RD, *(ii)* by MD and *(iii)* by ID. To kill a mutant it needs to incrementally do RD, MD and ID.

In order to ensure that the search algorithm finds a solution satisfying the requirements in the fitness function, we mainly consider the branch coverage as basis for our search objective. 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. Hence, the approach proposed starts by producing a set of tests that try kill a large proportion of mutants then continues generating incrementally to each of the remaining live ones.

覆盖较多的变异语句

A. *Generating test data for mutation testing*

Generating of test data for mutation testing can be a very costly activity. Firstly due to the complexity and input domain size of a program and after in regarding the high number of mutants generated.

To generate mutation adequate test set, a tester must design tests data able to reach necessarily the source code mutated, infect the rest of the code to incorrectly affect the state of

利用变异体没有被行到的控制依赖节点的数目来量化

the program and propagate this infection to the programs' output, these are referred to as reachability, necessity and sufficiency conditions. Generally, there are three scenarios for the relationship between the test data and the mutants:

1) **Easy-to-kill mutants:** a large number of test data exist that kill the mutant;
2) **Hard-to-kill mutants:** a small number of test data in the input domain cause the mutant to killed;
3) **Equivalent mutants:** there is no test data to kill this type of mutant.

In this paper, to deal with these scenarios, we use a adapted AUSTIN tool for test data generation and Proteum tool to generate the set of mutants and compute mutation score. 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 the next Section.

B. Fitness Functions

A fitness function describes the goodness of a candidate solution and it plays an essential role in guiding search-based techniques to obtain the best solutions. In this study, we utilized a fitness function scheme that combines four metrics.

The first two were proposed by Wegener [9] called Reach Distance (RD) used for branch coverage criteria in structural testing and they include approach level and branch distance. 错了 The third one introduced by Papadakis [2] is called Mutation Distance (MD) and the fourth one is named Impact Distance (ID). For the Impact Distance we developed and analyzed in three distinct strategies, which forms the main contributions of this paper.

Generally, for the test data generation problem, the search space is very large even to a small program. The search space consists of all input values possible to execute the program under test. For instance, consider a program with two integer input variable (a and b), the search space is each possible combination of integers for a and b within the input domain. Note that for a program can exists different optimal solutions, i.e. there are various inputs that fulfils the same specific test criterion. The problem that finding a test data set into a large search space is not always an easy task without a good fitness function [10]. Depending on the program and the test objective, different strategies for fitness functions can lead to the search towards different solutions.

Firstly, ensuring that the Reachability condition is fulfilled by at least one test data for a given mutant, we use branch coverage as a measure, because this criterion is widely studied and utilized as the basic principle in the most of fitness functions for structural based test data generation, such as 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 [9].

Korel [11] introduced branch distance as measure that computes how close a predicate is to being evaluated as true, 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.

For achieving infection state, we utilized a fitness function called mutation distance which is a generalization of the study proposed by Bottaci [12], in which a genetic algorithm fitness function was described using the conditions presented by Demillo [13]. An infection condition defines whether a test data execution would lead to a different state on a mutant program in the mutation point.

If this condition is satisfied, then the mutant differs from the original program at the mutation point or at some point during its execution and the mutant can be considered to be weakly killed. Otherwise, if executing a test data on a mutant does not lead to an infected execution state, this test cannot possibly lead to detection of the mutant. For instance, to infect the expression $a > b$ and the mutated one $a >= b$ is necessary an equal test input for both variables to obtain a different outcome.

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, 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 expression (1).

$$\begin{aligned}(Opred == T \&\& Mpred == F)|| \\ (Opred == F \&\& Mpred == T)\end{aligned} \qquad (1)$$

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

According to the branch distance fitness calculations, the fitness related to the expression (2) is called *predicate mutation distance (pmd)*. When is not possible to satisfy the expression (1), then the pmd specifies the distance of the test data 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.

Consequently, we use the impact as the last part of our fitness function. *Impact Distance (ID)* aims at guiding the

程序不实用啊

search towards identifying test data capable of exposing the differences in behavior, between the original and mutant programs. This function attempts to approximate the mutant sufficiency condition [13] by measuring the impact of the mutant on the program execution.

After the mutation is executed, a test data must propagate any changes induced by the mutation point such that they can be observed. Precisely, for achieving impact state, a test data should fulfill two conditions: First, the mutation needs to infect the state (necessity condition) and the hardest one, the mutation needs to propagate the infection to an observable state (sufficiency condition). Previous studies have been shown that mutants with high impact are more likely to be killable than those with less impact regardless of the strategy to measure the impact.

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 [6], Fraser and Arcuri [8] and Harman et al. [7], 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.

Here, we propose two new measures for impact distances. The aim is to evaluate its efficiency with a previous measure presented in [5] and between them. In the following, details of the three impact measures are given.

Impact Distance 1 (ID_1): The first impact distance was proposed in our previous study and ID1 is computed by expression (4), where $statementNum$ is the number of statements traversed 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, the impact is computed using the expression 3, where a and b are the branch distances from original and mutant programs.

$$Impact = \frac{abs(a-b)}{(abs(a-b)+1)} \tag{3}$$

$$\textbf{ID1} = statementNum - \sum Impact \tag{4}$$

Impact Distance 2 (ID_2): The objective of the second impact distance is to maximize the number of differences between the original program and a mutant considering the divergence on the paths after the mutation point. Therefore, if a test data traverses a different path from a mutant, after the mutation point, it means that the mutant is likely to be killed. The fitness value is measured through expression 5, where $diff$ are the different statements on the paths of the programs.

$$ID_2 = \sum(diff) \tag{5}$$

Impact Distance 3 (ID_3): For computing the third strategy proposed, we used a weighted measure considering each statement in the mutants. In this impact distance the objective is maximize the weighted divergences in each program statement (i) using the function 6. Where, $NoImpKilled$ is the number of times that this statement was impacted and the mutant was killed, $NoImpLive$ is the number of times this statement was impacted, and the mutant was live and $totalKilled$ for the number of times that a test data cover the statement, and the mutant was killed (see expression 6).

$$score(i) = \frac{100*(NoImpKilled(i))}{\sqrt{totalKilled(i)(NoImpKilled(i)+NoImpLive(i)))}} \tag{6}$$

The impact distance for a mutant is computed through sum the scores of each statements using the expression 7.

$$ID_3 = \sum(score(i)) \tag{7}$$

Finally, the proposed approach is guided by a fitness scheme based on RIP model and consequently in the three necessary conditions for killing a mutant. The whole process is guided by RD fitness function, leading the search to reach the mutation point (reachability condition); MD, to induce a different behavior 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 8.

$$\begin{aligned}\textbf{fitness function scheme} &= RD + MD + ID\\ \textbf{RD} = 2 * approachlevel &+ normalized(branchdistance)\\ \textbf{MD} &= normalized(PMD)\\ \textbf{ID} &= ID1, ID2 or ID3\end{aligned} \tag{8}$$

III. Experimental Study

This study empirically evaluates the effectiveness of the proposed approach to automating the test generation process on a set of C programs. This section describes the undertaken experiment. It firstly presents the goals of the experiment in Section III-A. Then, Section III-B presents the chosen subjects of the experiment. Finally, Section III-C details the procedure followed during the study.

A. Experimental Design

The present study aims to investigate the fitness scheme (RD & MD & ID) with two new impact distances measures for test data generation. We are interested in analyzing the contribution performed by each impact distance measures and overall effectiveness for generating and improving test data. Thus, we seek to explore the following Research Questions (RQs):

测试数据经过遍历点之后，如果执行路径与源程序不同，这有可能将错误的状态传播出去

*a) RQ_1: **How effective is fitness scheme for generating and improving test data to kill mutants?***

To answer RQ_1, we analyze the fitness scheme (RD & MD & ID) according to the three impact distance measures (ID_1, ID_2 and ID_3) for generating of test data. For this experiment, we used a set of mutants produced from Logical and Relational (L-R) operators implemented in the PROTEUM.

*b) RQ_2: **How effective is the fitness scheme for generating and improving test data to kill mutants compared to the genetic algorithm and random testing?***

为什么要跟这两个算法比较

In order to answer RQ_2, we compare the fitness scheme with the best ID, obtained from the RQ_1, with the genetic algorithm and random testing. We used a genetic algorithm employing the mutation score as the fitness function to assess the test data adequacy generated.

*c) RQ_3: **How efficient is the proposed approach for test data generation to mutation testing?***

The efficiency refers to the time of test data generation. The time was computed for each fitness scheme and includes all times to generate test data and their improvement. The total time *(T)* was computed by expression (9).

没有和随机测试以及基因算法比较？

$$\mathbf{T} = time(RD) + \sum time(MD) + \sum time(ID_x) \qquad (9)$$

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_x)* is the sum of time test data improvement using each ID for the alive mutants selected.

*d) RQ_4: **How efficient is the proposed approach about the number of test data compared to the genetic algorithm and random testing?***

The goal of this question is to evaluate the fitness scheme by considering the amount of test data necessary to achieve the mutation score when compared with genetic algorithm and random testing. For this, the test data generated for each approach were stored in 10 different executions and then computed the average.

程序会不会太小了

B. Subjects of Experiment

The conducted study uses in total 19 subjects of different domains and varying sizes. These programs varied in lines of code from 7 to 49 lines of code, totalizing 314 lines of code. The Subject Program column represents the program names. The LOC column details the programs' lines of code. These programs were chosen because they were in C and they have been used in empirical studies involving mutation such as [5].

C. Procedure of Experiment

To answer the stated RQs, we report the results derived from the experiments. Two experiments were conducted: *(i)* generation and improvement of test data to kill the mutants using fitness scheme (RD & MD & ID) with three impact distance measures (ID_1, ID_2 and ID_3) and, computation of mutation score, and time; and *(ii)* comparison between the best fitness scheme, genetic algorithm and random generation;

1) **Selecting subjects:** we selected 19 C programs P={p_1, p_2,...,p_{19}} as experimental subjects.
2) **Generating mutants:** we generate the set of mutants M = (m_1, m_2,...,m_x) using the PROTEUM tool and we compute the mutation score. The m_i is the set of mutants per each program p_i firstly obtained by logical and relational(L-R) operators.[1]
3) **Generating test data:** we generate the test data T = (t_1, t_2,...,t_n) using the AUSTIN tool adapted to mutation testing. The test data generation is guided by the fitness scheme. This test data are generated iteratively until a test data gets an appropriate fitness value for a determined fitness scheme. The test generation process considered up to 1000 fitness evaluations per branch to select a test data for each program p_i.
4) **Computing MS:** the total of mutation score is computed by the sum obtained in the each part of fitness function scheme.
5) **Computing Time:** the total time (seconds) of test data generation using the proposed approach is computed by expression 9.
6) **Test data generation using a GA:** we generate the test data through genetic algorithm to kill m.
7) **Comparison between the fitness scheme, random generation and, genetic algorithm:** we compare the most adequate fitness scheme, random generation and GA, to assess the suitability of the test data generated.

IV. Results and Analysis

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

A. Approach effectiveness (RQ_1)

This subsection considers the effectiveness of the assessed proposed approach (Fitness Scheme-FS). Figure 2 presents the mutation score achieved through our fitness schemes using each impact distance proposed (hereby denoted as FS_1 for RD, MD, ID_1; FS_2 for RD, MD, ID_2; and FS_3 for RD, MD, ID_3). The x-axis represents different subject programs and y-axis represents the average score from 0 to 100. In general, the different impact measures are not more efficient than each other. FS_1, FS_2 and FS_3 shows relatively constant mutation score, even that in some programs, the test data set from FS3 have killed more mutants.

在实际中到底用哪一个？

We can notice that in the most programs the number of killed mutants not suffered many changes, the best fitness scheme (FS_3) was able to kill around 86.02% of the mutants.

For additional details regarding the adequacy of test data generated through the best fitness scheme (FS_3) we have used the previous test data set generated by this fitness scheme to verify if these test data set are capable achieve a high mutation score when executed on mutants generated using L-R and all 71 operators per program.

[1]in the present study we used mutants produced by 12 mutation operators

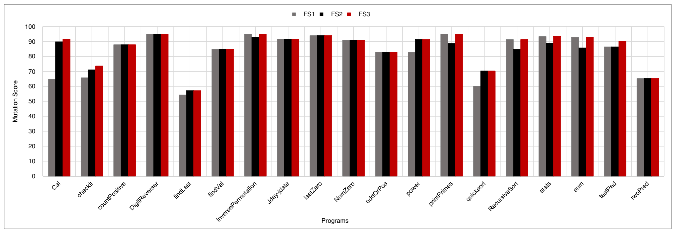

Fig. 2. Comparison of mutation score between each fitness scheme

怎么识别的等价变异体

为什么把这些变异算子
单独列出来

The obtained results are presented in Table I for the two set of operators (L-R and all 71 operators from PROTEUM) respectively. This table records the test data (TD column), the number of mutants (Num. of. Mut. column), the number of equivalent mutants (Equiv. column), the number of killed mutants (Killed column) and mutation score (% MS column) that were generated using L-R and all 71 operators per program.

Analyzing the results, we can observe that the test data set generated to kill the L-R mutants is adequate to kill the mutants generated with all operators since the mutation score reached by FS using all 71 operators was in average 89.06%. This is explained due to the test data set generated by the third part of the fitness scheme (ID) aid in the production of test data that also capable reach test requirements of other mutants in regarding the L-R mutants. Therefore, improving test data to maximize the impact on a mutant can produces a test that traverses different program parts and it can kill mutants not executed before.

B. Comparison between FS, GA and random testing (RQ_2)

The effectiveness of approaches was assessed through comparisons between FS, GA and random testing, in terms of mutation score. The results are summarized in Figures 3 and 4. Figure 3 presents details about the L-R mutants and test suite generated and Figure 4 refers to the results when all 71 mutants are used.

Figure 3 presents average mutation score achieved by FS, GA, and random generation. The bars represent the average of mutation score obtained for each approach and horizontal x-axis the subject programs. The results of Figure 3 suggest that the test suite generated by our approach is always more efficient than GA and random generation for L-R mutants. This is due fitness functions provide some knowledge about the problem to guide the search toward good candidate solutions. The result of this experiment also suggests that test suites from techniques that use only the mutation score as fitness function might not achieve some test requirements.

Furthermore, as can be seen at Figure 3, the result shows that FS achieve in average a mutation score was close to 100% using the L-R operators. The mutation score reached by FS was 29.17% greater than GA and 44.78% greater than random generation. We also notice that in 18 of 19 programs FS obtained better mutation scores than GA. However, one program, *twoPred*, achieved the same mutation score (65.31%) by FS and GA e another program (*findLast*) reached the same mutation score (48.57%) by GA and random.

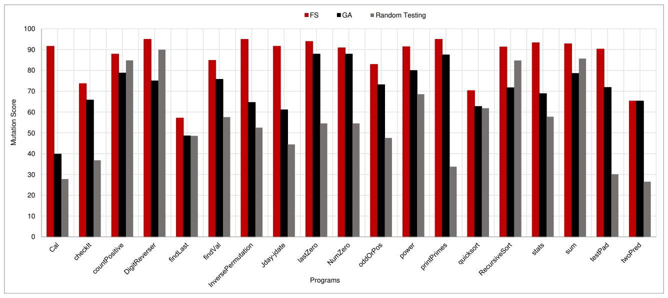

Fig. 3. Comparison of mutation score obtained by FS, GA and random testing for L-R mutants

Figure 4 shows average mutation score achieved by fitness scheme, GA and random testing using all operators. The x-axis represents the different programs. The y-axis represents average mutation score achieved by each program across 10 executions. In all programs the FS achieved greater score mutation (87.96%) than GA (70.82%) and random testing (53.13%). Three programs, *countPositive*, *DigitReverse*and *twoPred* reached the same mutation score (79.58%, 88.30% and 76.13% respectively) by GA and random testing approaches.

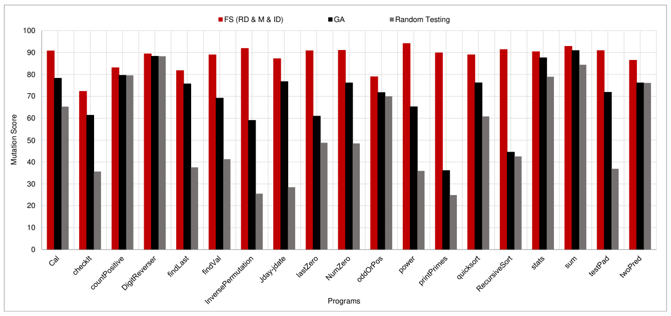

Fig. 4. Comparison of mutation score obtained by FS, GA and random testing for all mutants

C. Approach efficiency (RQ_3)

为什么没有GA以及RT的时间

To answer this RQ, we measure the time required to produce a test data suite using FS_1, FS_2 and, FS_3 for each subject program. The results showed that there is a notable difference between the times since we obtained on average 233.2, 353.13 and 616.18 for FS_1, FS_2 and, FS_3, respectively. This is due to the fact that FS_2 and FS_3 need satisfy more requirements for improving the test data than FS_1.

V. Related Work

In the last 25 years, great effort has been devoted to the study of mutation testing. However, few studies have investigated techniques to test data generation by trying to reduce costs in mutation testing. Despite the research's time,

TABLE I

ADEQUACY OF THE TEST DATA GENERATED FOR L-R MUTANTS AND APPLIED TO MUTANTS PRODUCED FROM 71 OPERATORS

Programs	TD	L-R Mutants				All Mutants			
		Num. of. Mut.	Equiv.	Killed	% MS	Num. of. Mut.	Equiv.	Killed	% MS
Cal	7	127	19	99	91.67	891	71	748	90.78
checkIt	6	41	3	28	73.68	104	3	73	72.28
countPositive	3	36	3	29	87.88	151	9	118	83.10
DigitReverser	3	36	16	19	95	496	43	424	93.60
findLast	6	36	1	20	57.14	198	17	148	81.77
findVal	7	36	3	28	84.85	190	18	153	88.95
InversePermutation	6	117	18	94	94.95	576	61	464	91.88
Jday-jdate	4	36	0	33	91.67	2821	81	2390	87.23
lastZero	5	36	3	31	93.94	173	9	149	90.85
NumZero	5	36	3	30	90.91	151	17	122	91.04
oddOrPos	3	110	28	68	82.93	361	71	229	78.97
power	5	36	1	32	91.43	268	12	241	94.14
printPrimes	7	90	10	76	95	715	64	642	98.62
quicksort	3	127	9	83	70.34	1026	82	881	93.33
RecursiveSort	4	48	2	42	91.30	555	45	466	91.37
stats	6	54	9	42	93.33	884	101	708	90.42
sum	4	18	4	13	92.86	165	11	143	92.86
testPad	11	110	7	93	90.29	629	57	520	90.91
twoPred	10	51	2	32	65.31	246	24	200	90.09

only 19 studies were identified in this context [14], which confirms that there is still a gap in research in this area.

Demillo and Offut [13] suggested the Constraint Based Testing (CBT) approach, 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 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 [2], [6]–[8], [15], [16] proposed approaches using search-based techniques employing these conditions.

Ayari et al. [15] proposed a approach using Ant Colony Optimization (ACO) meta-heuristic 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 [17] applied the same principles to kill mutants of Simulink Models.

In [16], [18], 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 [6], [8], an evolutionary approach that automatically generates unit tests for object-oriented classes based on mutation analysis is proposed. As already explained in section II-B, these aim at handling all three mutatnt killing conditions.

Another approach tackling the test data generation to mutation testing is proposed by Papadakis and Malevris [19]. This approach generates test inputs to kill weak and strong mutants using Dynamic Symbolic Execution (DSE). The main idea underlying the developed approach is to transform 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. [7] 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.

Furthermore, new directions of the mutation testing that have been investigated are the analyzing of impact mutation metrics can reducing the effects of equivalent mutants in its process.

Bernhard et al. [20] investigated the impact of a mutation on execution being that the more a mutation changes the execution, the greater the chance of not being equivalent. To assist this investigation the authors proposed a mutation testing framework called JAVALANCHE to assess the equivalence of mutations. To this end, JAVALANCHE focuses on small set of mutation operators that, from a mutant schemata, use coverage data to reduce the number of tests that need to be executed. The authors found that if a mutant alters control flow, it is more likely to be detectable by an actual test. When improving test suites, test managers therefore may focus on those surviving mutations that have the greatest impact on code coverage.

Schuler and Zeller [21] presents a study that extends the previous work on detect equivalent mutants [20] in several dimensions. This study adds impact on return values, three impact metrics (coverage, data and combined coverage and data impact) and three distance metrics (coverage, data and combined coverage and data distance) as an additional factor. These metrics was integrated into the JAVALANCHE framework. First, the framework runs the test suite on each generated mutant and ranks mutations by their impact on data and coverage. In order to measure the impact of mutations on the control flow, the program records the statement coverage for each test case and every mutation, that is, the number of times a statement is executed. From this, the result is a set of lines that were covered together with frequency counts for every program's method. Finally, the tester enhances the test suite to find out the top-ranked mutations. Furthermore, JAVALANCHE allows to notice and track the execution of mutations in order to define their impact. Similar to an avalanche, where one small event can have a huge impact, the framework aims at finding those mutations that have a big impact on the program run, that is, automatically classify mutations that are less likely to be equivalents.

Contrary, the present approach aims to automate the test generation process. We use AVM combined with a fitness scheme (RD & MD & ID). For the ID we developed and analyzed in three distinct strategies that also this can be the basis for an equivalent mutants analysis method. These metrics are used to measure the adequacy of test data and direct the search process to an optimum in the search space.

VI. Conclusions

The paper presents an automated approach for generating test data for mutation testing. Our approach is based on two observations. First, search techniques with an adequate fitness function are beneficial to generate test data even without any knowledge of the program under test. Second, a method that achieves high adequacy can reduce the efforts to identify equivalents mutants and overcome the weakness of the mutation testing.

The results show that search based techniques are excellent in finding test data. The RIP represents the three conditions (reachability, necessity, and sufficiency) for a test data to kill a mutant, i.e. if a test data satisfies these conditions it can kill a given mutant. Our experimental results indicate that when using the RIP conditions into our fitness function, it produces effective test data that are able to strongly kill the majority of mutants.

Conclusively, the major contributions made by the present paper can be described as: a fitness scheme that provides support search-based testing based on weak and firm mutation for strongly killing mutants using two new impact measures. 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.

Despite encouraging results, there are many challenges remaining in making test data generation and mutation testing tools robust to be used in industrial projects. Test data generation and equivalent mutants identification are considered undecidable problems, thus there are no a solution 100% adequate to all programs. In this sense, future work comprises conducting additional experiments to statistically re-validate the findings of the experiment. Further, a conduction with new large-scale experiments that can establish our finding and demonstrate the importance of RIP conditions and impact measures to generate test data and identify equivalent mutants.

References

[1] M. Harman and P. McMinn, "A theoretical and empirical study of search-based testing: Local, global, and hybrid search," *IEEE Trans. Softw. Eng.*, vol. 36, no. 2, pp. 226–247, 2010.

[2] M. Papadakis and N. Malevris, "Searching and generating test inputs for mutation testing," *Springer Plus*, vol. 2, no. 1, pp. 1–12, 2013.

[3] 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*, 2010, pp. 101–110.

[4] D. M. E. and J. C. Maldonado, "Proteum-a tool for the assessment of test adequacy for c programs," in *Proceedings of the Conference on Performability in Computing Systems (PCS'96)*, New Brunswick, New Jersey, July 1996, pp. 79–95.

[5] F. C. M. Souza, M. Papadakis, Y. Le Traon, and M. E. Delamaro, "Strong mutation-based test data generation using hill climbing," in *Proceedings of the 9th International Workshop on Search-Based Software Testing*, ser. SBST '16. ACM, 2016, pp. 45–54.

[6] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *Proceedings of the 19th IISSTA*, 2010, pp. 147–158.

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

[8] G. Fraser and A. Arcuri, "Achieving scalable mutation-based generation of whole test suites," *Empirical Software Engineering*, vol. 20, no. 3, pp. 783–812, 2014.

[9] J. Wegener, A. Baresel, and S. Harmen, "Evolutionary test environment for automatic structural testing," *Information and Software Technology*, vol. 43, no. 14, pp. 841–854, 2001.

[10] 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*. Springer, 2008, pp. 170–181.

[11] B. Korel, "Automated software test data generation," *IEEE Trans. Softw. Eng.*, vol. 16, no. 8, pp. 870–879, 1990.

[12] L. Bottaci, "A genetic algorithm fitness function for mutation testing," in *Proceedings of the ICSE*, 2001, pp. 3–7.

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

[14] 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*, 2014, pp. 1–14.

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

[16] 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*, 2010, pp. 111–118.

[17] 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*, 2005, pp. 1061–1068.

[18] M. Papadakis and N. Malevris, "Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing," *Software Quality Control*, vol. 19, no. 4, pp. 691–723, 2011.

[19] N. Malevris and M. Papadakis, "Automatic mutation test case generation via dynamic symbolic execution," in *2010 IEEE 21st International Symposium on Software Reliability Engineering(ISSRE)*, vol. 00, 11 2010, pp. 121–130.

[20] B. J. M. Grun, D. Schuler, and A. Zeller, "The impact of equivalent mutant," in *Proceedings of the 9th International Conference on Software Testing Verification and Validation Workshops*, ser. ICSTW '09. IEEE, 2009, pp. 192–199.

[21] D. Schuler and A. Zeller, "(un-)covering equivalent mutants," in *Proceedings of the 3rd International Conference on Software Testing Verification and Validation*, ser. ICST '10. IEEE, 2010, pp. 45–54.