Search-Based Mutant Selection for Efficient Test Suite Improvement: Evaluation and Results

Pedro Delgado-Pérez[a,*], Inmaculada Medina Bulo[a]

[a]*Escuela Superior de Ingeniería, Universidad de Cádiz, Spain*

Abstract

Context: Search-based techniques have been applied to almost all areas in software engineering, especially to software testing, seeking to solve hard optimization problems. However, the problem of selecting mutants to improve the test suite at a lower cost has not been explored to the same extent as other problems, such as mutant selection for test suite evaluation or test data generation. **Objective:** In this paper, we apply search-based mutant selection to enhance the quality of test suites efficiently. Namely, we use the technique known as Evolutionary Mutation Testing (EMT), which allows reducing the number of mutants while preserving the power to refine the test suite. Despite reported benefits of its application, the existing empirical results were derived from a limited number of case studies, a particular set of mutation operators and a vague measure, which currently makes it difficult to determine the real performance of this technique. **Method:** This paper addresses the shortcomings of previous studies, providing a new methodology to evaluate EMT on the basis of the actual improvement of the test suite achieved by using the evolutionary strategy. We make use of that methodology in new experiments with a carefully selected set of real-world C++ case studies. **Results:** EMT shows a good performance for most case studies and levels of demand of test suite improvement (around 45% less mutants than random selection in the best case). The results reveal that even a reduced subset of mutants selected with EMT can serve to increase confidence in the test suite, especially in programs with a large set of mutants. **Conclusions:** These results support the use of search-based techniques to solve the problem of mutant selection for a more efficient test suite refinement. Additionally, we identify some aspects that could foreseeably help enhance EMT.

Keywords: search-based software engineering, mutation testing, evolutionary algorithm, genetic algorithm

*Corresponding author: pedro.delgado@uca.es

1. Introduction

The increasing complexity of software systems has led to the emergence of a variety of well-known software engineering problems related to almost all activities of the software lifecycle. The gradual appearance of search and optimization techniques has allowed successfully finding near-optimal solutions to many of these hard-to-solve problems with affordable resources, with a remarkable contribution to the software testing phase [1]. Mutation testing, a powerful technique to assess and improve test suites, has not been indifferent to the opportunity offered by these optimization methods, as it can be seen in this review [2]. These search-based techniques are especially appealing to alleviate costly and tedious tasks, such as the execution of the whole set of mutants and the inspection of all equivalent mutants. Indeed, these are two key impediments that researchers are urged to investigate for a greater integration of mutation testing in the industry.

However, these innovative techniques have been mainly devised for the problem of test case generation [3, 4, 5]. Only in the last years, these techniques have been applied to the problem of mutant selection with the aim of enhancing the fault detection of a test suite. That is the goal of the technique known as *Evolutionary Mutation Testing* (EMT). This technique was conceived on the premise that mutants killed by few or none of the test cases in the current test suite can guide the generation of similar useful mutants. This can be achieved by means of an evolutionary algorithm that favors the generation of mutants from the same mutation operators that produced them, or in nearby areas to those mutation locations. In this way, we can increase the probability of finding a great percentage of surviving mutants –which can help the tester improve the test suite– without the need to generate all mutants, thereby reducing the cost of applying mutation testing.

Problem: The late attention to the application of optimization techniques to automate the selection of mutants for test suite improvement implies a lack of knowledge about the efficiency of these methods in different contexts (e.g., EMT has only been applied to WS-BPEL compositions since its inception). Furthermore, we have also detected the absence of a well-defined methodology to measure its performance appropriately. The ratio of *strong mutants* found (i.e., those mutants with great potential to induce the design of a new test case) was used as an indicator of the performance in the first experiments conducted. However, that measure only gives us a partial view of the performance: finding a percentage of strong mutants does not necessarily entail a proportional extension of the test suite.

Contribution: The objective of this research paper is to increase knowledge via experimentation about how much the test suite under evaluation can be refined through the mutants selected by EMT. The experiments conducted in this paper go a step beyond than previous experiments –where the ability of the technique to find strong mutants was evaluated– by assessing in depth a new methodology that allows us to estimate the extent to which EMT could help improve the test suite. This work connects and extends the results of two previous papers [6, 7], providing a comprehensive picture regarding the behavior of EMT, descriptive examples on how to implement the proposed methodology, statistical significance of the results and a discussion comparing the old and the new methodology and the results of EMT, random selection and selective mutation. The paper also includes a list of lessons learned that can be useful for researchers in this field in a foreseeable future. Additionally, we double the number of case studies to eight publicly available programs coded in C++. Overall, the results reported in this paper support the following hypothesis:

Hypothesis: *Evolutionary Mutation Testing is a useful mechanism to improve the test suite generating a subset of the full set of mutants.*

Namely, the contributions in this paper are as follows:

1. **We describe and evaluate in detail the proposed method to assess the effectiveness of EMT.** This methodology is based on the actual improvement of the test suite that the selected mutants may induce. A comparison between the results of the previous and this novel methodology reveals that the performance of EMT relates to the nature of the mutants selected and not only to finding a large number of strong mutants. Indeed, one of the main conclusions is that we do not need to generate a great percentage of mutants to reach noticeable improvements. As an example, to reach 80% of the size of a test suite that kills all non-equivalent mutants (i.e., a mutant-adequate test suite), the percentage of mutants generated on average ranges from 16% to 54% of the full set of mutants.
2. **The experiments show that EMT outperforms random selection of mutants and selective mutation**. Both, when measuring the percentage of strong mutants and test suite improvement, EMT gets better results overall. The differences between EMT and random selection of mutants are even more remarkable with the new evaluation method, statistically significant in many cases, and the evolutionary algorithm also shows more stability. In the best case, the difference between both techniques was about 45% of the mutants to find a mutant-adequate test suite. Combining random selection with selective mutation does not decrease the number of mutants generated in general, but the results suggest that removing low-productive operators could increase the performance of EMT.
3. **We provide a list of lessons learned based on the experience and anticipate possible refinements of the algorithm to improve the effectiveness of EMT**. These handy tips address different factors that affect the performance of EMT. We can highlight that EMT performs better in complex programs, and when the test suite is not at an initial stage or comprised of overlapped test cases. Additionally, EMT should work better when valuable mutants are concentrated on specific areas or produced by a subset of mutation operators. If this is not the case (valuable mutants are highly spread), it is likely that favoring the generation of mutants from all operators and in all areas of the code may help guide the

search. Additionally, the results suggest that the algorithm could benefit from the integration of methods that detect equivalent mutants.

The structure of the rest of the paper is as follows. Section 2 motivates the use of a search-based technique to search for useful mutants to improve the quality of the test suite. Section 3 details the evolutionary approach followed by EMT, while Section 4 explains the proposed method to assess its effectiveness. The research questions and the empirical evaluation are presented in Section 5. Several lessons learned and possible enhancements are shown in Section 6. The paper ends with the related work in Section 7 and the conclusions in Section 8.

2. Motivation

Mutation testing is a fault-based technique with increasing importance as a method for measuring and improving the effectiveness of test suites in finding plausible faults in the code. To achieve this:

1. Different transformation rules are applied to the code of the program under test, replacing, inserting or deleting simple code fragments, so-called *mutation operators*.
2. As a result, an assortment of faulty versions are generated (each one with a syntactic change), which are called *mutants*.
3. Then, the test suite is urged to reveal those faults by re-executing each mutant against the current test suite and determine if some of its test cases are able to *kill the mutant* (i.e., detect the mutation). Such detection means that the mutated program exhibits different functionality from the original program (based on the output of both programs).
4. Undetected mutants are known as *live mutants* and show that the test suite might be insufficient –therefore it would require to be reinforced.

Search-based software engineering seeks to reformulate known problems in the field of software engineering as optimization problems, and then use metaheuristics techniques to solve them, such as using genetic algorithms [8] or ant colony optimizations [9]. Search-based techniques are especially useful with hard optimization problems, where using exact algorithms is often unfeasible while metaheuristics strategies have shown the ability to reach near-optimal solutions with reasonable computational effort.

Despite awareness of the effectiveness of mutation testing as a method for increasing the confidence level in test quality, this technique suffers from the kind of problems that search-based strategies could help to ease: the high computational cost of executing all mutants and the huge amount of human effort required to identify *equivalent mutants* (i.e., mutants which do not differ from the original program from a functional point of view). Different surveys [10, 11] have collected several techniques suggested over the years to face both problems.

Evolutionary Mutation Testing (EMT) proposes to use a metaheuristic approach for the selection of mutants with the aim of improving a test suite. More specifically, EMT selects only a subset of the full set of mutants with a particular goal: maximizing the number of mutants contained in that subset with the potential to provide the tester with information to enhance the current test suite. This is especially helpful when a vast number of mutants can be derived from the code, and the program compilation and execution are costly.

Delgado-Pérez et al. [12] recently gave evidence that the effectiveness of selection techniques should be differently assessed depending on whether our goal is evaluating (TSE) or refining a test suite (TSR):

- A good selection technique for *TSE* should avoid the selection of redundant mutants, as they can be removed without affecting the mutation score (i.e., ratio of non-equivalent mutants killed by the test suite). This is the measure traditionally used to assess the fault detection capability of a test suite [13].
- A good selection technique for *TSR* should select mutants that can induce the creation of new test cases. Each added test case can affect the mutation score differently depending on its design (i.e, the number of mutants killed depends on the specificity of the new test case). Therefore, the mutation score can be misleading and the increase in the test suite size should be measured instead.

Notice that the selection of mutants through the application of EMT is done solely with the aim of improving the test suite, that is, TSR. As such, EMT should be assessed on the basis of the increase in the quality of the test suite.

Finally, we should note that the problem of mutant selection for test suite improvement is different from the problem of test case generation [14]; the design of new test cases is the following step that the tester should undertake by means of the selected mutants. In fact, these selected mutants could be used to feed other test case generation algorithms.

3. Evolutionary Mutation Testing

In the following subsections, we explain the main concepts related to this search-based strategy and how it works internally.

3.1. Genetic algorithm

EMT is based on the use of *evolutionary algorithms* [15]. These are population-based algorithms in which a set of individuals are recombined and mutated with the goal of generating better individuals for the problem addressed. The evolutionary approach fits well with the problem of the selection of a subset of mutants: we can first generate a relatively small subset of mutants or individuals (which depends on the size of the program under test) and then produce new individuals derived from those that seemingly have the greatest potential to contribute to the design of new test cases. Concretely, Domínguez-Jiménez et al. [16] proposed the use of a *genetic algorithm* (the most used method to address software engineering problems according to the review by Harman et al. [1]).

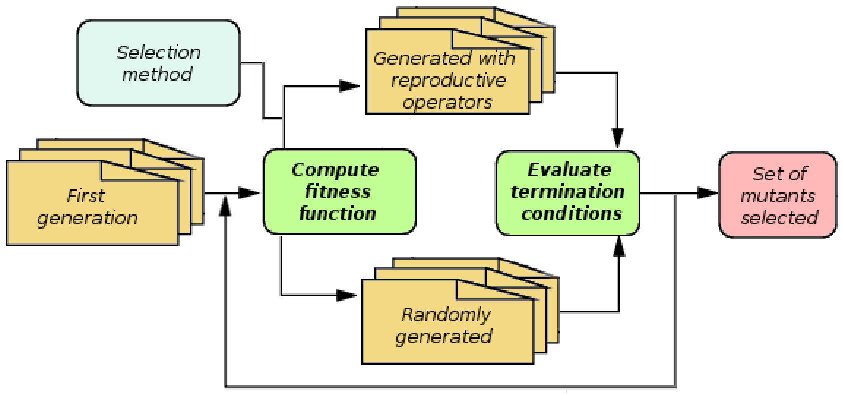

Figure 1: Diagram of the steps followed by EMT.

Figure 1 illustrates how this genetic algorithm manages the set of individuals and the elements involved in its application:

1. A *first generation* of mutants is randomly produced. The population size in a generation (PS) needs to be previously set.
2. The *fitness function* is responsible for determining the quality of each of the individuals.
3. The algorithm produces a new generation. To that end, a *selection method* uses the computed fitnesses to select some mutants in order to engender new mutants through predefined *reproductive operators*. At the same time, a percentage N of the mutants in each generation is randomly produced.
4. The algorithm stops according to one or more *termination conditions* set by the tester (for instance, when reaching a number of generations). If so, the algorithm returns the set of mutants generated so far. Otherwise, the algorithm keeps iterating and produces a new generation.

3.2. Fitness function

The fitness function in EMT gives preference to mutants that are not well covered by the test suite, as these are the mutants that can lead to the refinement of the test cases. Equation 1 shows how the fitness function measures the quality of each individual for this particular problem. Let I be the mutant to compute its fitness and S the current test suite. Let also be M the number of mutants and T the number of test cases in S. Finally, m_{ij} takes the value 1 when the mutant i is detected by test case j, and 0 otherwise. Then, according to Equation 1, the fitness of a mutant will be in the range $[0, M \times T]$[1], where this value will be close to $M \times T$ when (1) there are few test cases killing the mutant, and (2)

[1]The fitness can be normalized between $[0, 1]$. For the sake of clarity, we maintain the range $[0, M \times T]$ in this study.

there are few mutants being detected by those test cases at the same time. This second part of the equation helps differentiate mutants killed by specific test cases (and, therefore, interesting mutants) from those killed by general ones.

$$\text{Fitness}(I, S) = M \times T - \sum_{j=1}^{T} \left(m_{Ij} \times \sum_{i=1}^{M} m_{ij} \right) \tag{1}$$

Consequently:

- Live mutants, which are the most valuable mutants to improve the test suite, are assigned the highest value ($M \times T$) because $m_{Ij} = 0$ for all j. These surviving mutants were originally called *potentially-equivalent mutants* because, once these mutants are reviewed, either they can lead to the design of a new test case or turn out to be equivalent.
- Mutants killed by a single test case, which does not kill any other mutants, receive a fitness of $M \times T - 1$ because $m_{Ij} = 0$ for all j except for one (test case z), which kills no other mutants ($m_{Iz} = 1$ and $\sum_{i=1}^{M} m_{iz} = 1$). These mutants are known as difficult to kill mutants.
- The rest of the mutants are considered as weak in general and are assigned a fitness lower than $M \times T - 1$. The more test cases kill I and the more mutants those test cases kill in turn, the lower its fitness.

Potentially-equivalent mutants together with difficult to kill mutants were called *strong mutants* [16]. Nonetheless, different criteria could be used to define what is considered a strong mutant. Note that difficult to kill mutants are even more specific than stubborn mutants [17] since the latter does not require that the test case killing it does not kill other mutants.

The algorithm makes use of both current and historical information for a more precise measurement of the fitness of each mutant. This means that the fitness of a mutant is not only based on the execution results of the current generation but also on those of the mutants selected so far by the algorithm. To do so, EMT compiles in a *second population* the data of the test suite execution of all the mutants produced in previous generations. In this way, a mutant that seems well suited for reproduction with respect to the current generation may result in a low-quality mutant when analyzed with respect to the *current+second* population.

The fitness function is designed to penalize groups of mutants killed by a similar subset of tests. Thus, the more mutants in the same group are selected, the more the fitness of those mutants drops. This fact serves a twofold purpose:

- **Areas of code.** Mutations killed by the same test cases are normally placed in similar areas of the code. Therefore, when the fitness attached to these mutants is low, the algorithm will tend to focus on other areas.
- **Mutation operators.** Likewise, similar mutants are usually derived from the same operators. When those mutants present poor fitnesses, fewer mutants from those operators will be produced in successive generations.

Intuitively, the algorithm should work better when several strong mutants are found in some apparently not well-covered areas. The same happens when some mutation operators concentrate the generation of strong mutants. As it can be deduced, this fact not only depends on the nature of the mutants themselves, but it also depends on the initial test suite. That is, if the tester fails to address a feature of the language when designing test cases, that leaves room for EMT to find related mutants that help reveal those deficiencies.

3.3. Selection and reproductive operators

A genetic algorithm depends on two kinds of operators, selection and reproductive operators, described below:

Selection operator. Selection operators follow different criteria to select individuals from the population [15]. The genetic algorithm in *GAmera* [18], the first tool implementing EMT, applies the *roulette wheel method* [19]. This selection method is deemed suitable in the case of EMT because it is known to have a quick convergence, which allows for reducing the subset of mutants generated as much as possible.

Reproductive operators. The individual representation consists of two fields (which identify uniquely each mutant):

- **Operator**: A code assigned to the mutation operator that generated the mutant.
- **Location**: An integer that represents the order in the code of the mutants produced by each mutation operator.

As an example of this encoding format, the mutant `O3L5` is generated by the mutation operator (`O`) tagged with the code 3, and it is placed in the fifth location (`L`) –therefore, that operator generated four mutants before detecting this location. Note that, in the original definition of EMT [16], individuals were represented using a further field, the attribute, which identifies different mutations inserted into the same location. That field, however, can be mapped using the location field (each mutation is considered a new location). This is a preferable option when none or few mutation operators can produce more than a single mutant per location.

There are two reproductive operators that change these two fields to engender new mutants in the next generation:

- **Mutation operators**. There are two mutation operators that can be applied to a mutant `OxLy`. The first one modifies the value of the operator field, producing a mutant from a different operator but with the same number of location (`Ox'Ly`); the second one modifies the value of the location field instead of the operator (`OxLy'`). p_m represents the probability that a mutation operator is applied.

- **Crossover operator**. Two parents, `Ox1Ly1` and `Ox2Ly2`, share their data to breed two children that inherit information from both parents: `Ox1Ly2` and `Ox2Ly1`. p_c is the probability that a crossover operator is applied.

A graphical explanation of these two operators can be found in [7]. The idea behind these operators is to find similar individuals to those that were selected for reproduction. For instance, the mutation operator that changes the field location in the offspring seeks to produce mutants in a nearby area. We can illustrate this fact with the fragment of code shown in Figure 2, where two different mutation operators produce several mutants. Taking into account that the selection operator selects the individuals based on how high their fitnesses are, if mutant `O1L3` is selected for reproduction, it is reasonable to generate mutant `O1L2` as well: both mutants are in the same line of the code and, if that line is not properly exercised by the test suite, generating new mutants in near locations could lead to the detection of a missing test case.

```
/* First part of the code */
... ...

if( (x > 0 && (O1L2) y > 0) || (O1L3) (x > z) ){
    z++ (O2L1)
}

/* Intermediate code */
... ...

w = true;
while (x < y && (O1L8) w){

    if (x == z){
        w = false;
    }
    x++ (O2L2)
}

/* Rest of the code */
... ...
```

Figure 2: Fragment of code analyzed with respect to two mutation operators: O1 (replacement of the operators && and ||) and O2 (deletion of increment/decrement operators). Mutation locations are highlighted with a box, followed by the individual representation used by EMT (operator and number of location).

In order to generate a similar mutant to its parent, the value of the field is mutated according to this equation:

$$\beta = (\alpha \pm random(1, 10(1 - p_m))) \pmod{U} \tag{2}$$

Where:

- β is the final value of the field.
- α is the current value of the field.
- α is added or subtracted a random value in the range $[1, 10(1-p_m)]$. In this way, the upper limit of this range decreases as p_m increases, which reduces the impact of the mutation.
- U is the maximum value that the field can be assigned. The operation is carried out modulo U to avoid generating nonexistent mutants.

Thus, it is likely that the mutant `O1L3` breeds the mutant `O1L2` rather than other distant mutants like `O1L8`. Similarly, the change of the operator field aims at deriving mutants from operators that belong to the same syntactic category. As a remark, the method to generate new nearby mutants could be simplified by just increasing/decreasing 1 to the field selected for mutation. Analyzing the difference in results between both methods would merit further investigation.

Finally, the underlying idea of the application of the crossover operator is that two parents share their information to produce fitter individuals. For instance, imagine that the selection operator selects the mutants `O1L8` and `O2L1` for reproduction. By crossing these mutants, we favor the application of operators 1 and 2 (which seemingly produce high-quality mutants) as well as mutants located in two areas of the code not properly covered by the test suite. As such, the crossover operator should be especially effective when the different mutation operators produce a similar number of mutants across the code.

4. New Evaluation Methodology

4.1. Motivation

In order to justify the use of EMT, it is important to define a proper evaluation method to measure its performance. In previous works, the authors conducted some experiments to calculate **the percentage of the total number of strong mutants** found. To that end, they followed the next steps:

1. As a first step before EMT execution, run all mutants against the current test suite. By evaluating the results, we can know the total number of strong mutants in advance.
2. Execute EMT until the termination condition is reached.
3. Measure the percentage of the total number of strong mutants contained in the subset of mutants returned by EMT.

This evaluation method, however, presents two main issues. First, we should know that some of the strong mutants can turn out to be equivalent since the set of potentially-equivalent mutants contains the set of equivalent mutants (as it was commented in Section 3.2). Second, after inspecting a surviving mutant, we could design a test case that is able to kill several other surviving mutants. As a consequence, the number of strong mutants found is not an accurate measure of the actual test suite improvement, and a new evaluation method is required.

4.2. New methodology

The idea behind the proposed methodology is to resemble a real process of test suite refinement when the surviving mutants found by EMT are manually inspected. Note, however, that the new test cases are not added as a direct result of reviewing the selected mutants, but we base this simulation on previously generated information to be able to automate the process. In this way, we can estimate the **number of new test cases** that the subset of mutants generated with EMT could help design. In this case, we follow the next general steps:

1. As a first step before EMT execution, run all mutants against the current test suite.
2. Review the surviving mutants and create new test cases until all non-equivalent mutants are killed. The result is an improved version of the test suite, so-called *mutant-adequate test suite* (henceforth, simply adequate).
3. Execute EMT until the termination condition is reached.
4. Determine which test cases of the improved test suite could be induced by the mutants returned by EMT. As it will be seen later on, this step implies analyzing the execution results of the selected mutants against the adequate test suite in order to remove test cases that are not actually necessary (i.e., minimize the test suite). The result is a subset of the adequate test suite that kills all non-equivalent mutants selected by EMT.
5. Measure the percentage of the test cases in the adequate test suite contained in the set of test cases obtained in the previous step.

A valuable resource to achieve step 4 is to save the execution results as a matrix structure, so-called *execution matrix*. The execution matrix of size M (mutants) $\times$ T (test cases) records a value 1 in the intersection of a mutant and a test case when that test case is able to detect the mutant (otherwise, the value is 0). Among other advantages, the execution matrix is useful to know the state of a mutant:

- **Live mutant**: the row representing the mutant is filled with value 0.
- **Dead mutant**: the row contains, at least, a cell with value 1.

As a last remark, *invalid mutants* (i.e., mutants containing syntactically illegal mutations) are represented with a row filled with value 2. This representation of invalid mutants is useful for EMT as all these mutants can be directly identified and removed so that they are not assigned a fitness and they do not affect the fitness computation of the rest of the mutants.

In the following two subsections, we detail the two phases that are necessary to put this evaluation methodology into practice.

First phase

The first phase (steps 1 and 2) consists in designing an adequate test suite. This implies the execution of the whole set of mutants against the current test suite, the inspection of the mutants that survived the execution and the design of test cases driven to kill them.

In order to explain the proposed methodology, we will use a running example based on the execution matrix in Figure 3. This execution matrix, which we will call *original matrix*, shows the result of the execution of each of the test cases designed so far ($|T| = 5$) on each of the mutants generated in our example program ($|M| = 8$). By interpreting this matrix, we can know which mutants are alive, dead or invalid:

- **Live mutants**: $mutant_2$, $mutant_4$, $mutant_6$ and $mutant_7$.
- **Dead mutants**: $mutant_1$, $mutant_3$ and $mutant_8$.
- **Invalid mutants**: $mutant_5$.

$$\begin{array}{r c}
 & \begin{array}{ccccc} \text{test}_1 & \text{test}_2 & \text{test}_3 & \text{test}_4 & \text{test}_5 \end{array} \\
\begin{array}{r} \text{mutant}_1 \\ \text{mutant}_2 \\ \text{mutant}_3 \\ \text{mutant}_4 \\ \text{mutant}_5 \\ \text{mutant}_6 \\ \text{mutant}_7 \\ \text{mutant}_8 \end{array} &
\begin{pmatrix}
1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
2 & 2 & 2 & 2 & 2 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1
\end{pmatrix}
\end{array}$$

Figure 3: Original matrix (associated to the original test suite and the full set of mutants).

$$\begin{array}{r c}
 & \begin{array}{ccccccc} \text{test}_1 & \text{test}_2 & \text{test}_3 & \text{test}_4 & \text{test}_5 & \text{test}_6 & \text{test}_7 \end{array} \\
\begin{array}{r} \text{mutant}_1 \\ \text{mutant}_2 \\ \text{mutant}_3 \\ \text{mutant}_4 \\ \text{mutant}_5 \\ \text{mutant}_6 \\ \text{mutant}_7 \\ \text{mutant}_8 \end{array} &
\begin{pmatrix}
1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \mathbf{1} & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & \mathbf{1} \\
2 & 2 & 2 & 2 & 2 & 2 & 2 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & \mathbf{1} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0
\end{pmatrix}
\end{array}$$

Figure 4: Adequate matrix (associated to the adequate test suite and the full set of mutants). The new values "1" (mutants killed by new/modified test cases) are highlighted.

This is the information available to the tester after the execution of the *original test suite*. Based on those results, the tester will face the design of new test cases or the modification of existing ones to kill surviving mutants. Considering the previous example, imagine that the tester:

$$\begin{array}{c} \\ \text{mutant}_1 \\ \text{mutant}_3 \\ \text{mutant}_4 \\ \text{mutant}_7 \end{array} \begin{array}{c} \begin{array}{ccccccc} \text{test}_1 & \text{test}_2 & \text{test}_3 & \text{test}_4 & \text{test}_5 & \text{test}_6 & \text{test}_7 \end{array} \\ \left(\begin{array}{ccccccc} 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{array} \right) \end{array}$$

Figure 5: Selected matrix (associated to the adequate test suite and the subset of mutants selected by EMT).

- Finds that the mutants $mutant_2$, $mutant_4$ and $mutant_7$ are killable; $mutant_6$ is however equivalent.
- Adds two new test cases, $test_6$ and $test_7$, to kill $mutant_2$ and $mutant_4$ respectively.
- Modifies $test_4$ (by adding a missing assertion) to kill $mutant_7$.

Therefore, the tester obtains an adequate test suite with size $|T| = 7$. When running this improved test suite on the same mutants, a new execution matrix is obtained, which we will refer as *mutant-adequate matrix*. This matrix is shown in Figure 4. The adequate test suite will be used as a ground truth in the second phase to evaluate how close we are to killing all non-equivalent mutants with the aid of the mutants selected by EMT.

Second phase

In the second phase (steps 3, 4 and 5), it is the turn to run EMT and then evaluate how much the test suite could be improved with the mutants selected by the genetic algorithm so far. To do this, we analyze the mutants generated by EMT in order to know which of the test cases in the adequate test suite are necessary to kill that subset of mutants (those which are not equivalent). Finally, we compare the size of that subset of test cases with the size of the adequate test suite.

Going back to our example, we can imagine that EMT is run and $mutant_1$, $mutant_3$ $mutant_4$ and $mutant_7$ are selected. We can estimate how much the test suite could be improved thanks to these mutants by extracting their execution results from the adequate matrix. The resulting matrix, which we will call as *selected matrix* from now on, is shown in Figure 5.

At this point, we need to know which of those seven test cases are really necessary to kill those four mutants. A good option to do this is using the concept of *test suite minimization*: that avoids the negative impact that redundant test cases could have on the results, and also removes test cases that kill no mutants. We can minimize a test suite regarding a set of mutants to obtain a new test suite (*minimal test suite*) that detects the same mutants as the original test suite but with as few test cases as possible. According to this concept, we can obtain a *minimal adequate test suite* through the analysis of the adequate

$$\begin{array}{c} \\ \text{mutant}_1 \\ \text{mutant}_2 \\ \text{mutant}_3 \\ \text{mutant}_4 \\ \text{mutant}_7 \\ \text{mutant}_8 \end{array} \begin{array}{c} \begin{array}{ccccc} \text{test}_3 & \text{test}_4 & \text{test}_5 & \text{test}_6 & \text{test}_7 \end{array} \\ \begin{pmatrix} \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \mathbf{1} & 0 \\ \mathbf{1} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \mathbf{1} \\ 0 & \mathbf{1} & 0 & 0 & 0 \\ 0 & 0 & \mathbf{1} & 0 & 0 \end{pmatrix} \end{array}$$

(a)

$$\begin{array}{c} \\ \text{mutant}_1 \\ \text{mutant}_3 \\ \text{mutant}_4 \\ \text{mutant}_7 \end{array} \begin{array}{c} \begin{array}{ccc} \text{test}_3 & \text{test}_4 & \text{test}_7 \end{array} \\ \begin{pmatrix} \mathbf{1} & 0 & 0 \\ \mathbf{1} & 0 & 0 \\ 0 & 0 & \mathbf{1} \\ 0 & \mathbf{1} & 0 \end{pmatrix} \end{array}$$

(b)

Figure 6: a) Matrix associated to the minimal adequate test suite; b) Matrix associated to the minimal selected test suite.

matrix. Similarly, we can obtain a *minimal selected test suite* by analyzing the selected matrix as follows:

- **Minimal adequate test suite:** It contains the test cases $test_3$, $test_4$, $test_5$, $test_6$ and $test_7$; $test_1$ and $test_2$ can be discarded given that the mutants that they kill ($mutant_1$ and $mutant_3$ respectively, as shown in Figure 4) are also killed by $test_3$. The resulting matrix can be seen in Figure 6 (a).

- **Minimal selected test suite:** It contains $test_3$, $test_4$ and $test_7$: according to Figure 5, these test cases suffice to kill $mutant_1$, $mutant_3$ $mutant_4$ and $mutant_7$. The resulting matrix can be seen in Figure 6 (b).

As a result, we can say that the four mutants returned by EMT ($mutant_1$, $mutant_3$ $mutant_4$ and $mutant_7$) could guide the design of up to 3 ($test_3$, $test_4$ and $test_7$) of 5 test cases ($test_3$, $test_4$, $test_5$, $test_6$ and $test_7$) that are necessary to kill all non-equivalent mutants. Thus, we have reached the estimation of the improvement that we could achieve with the mutants selected by EMT.

4.3. Termination condition

In practice, the genetic algorithm can be configured to stop the execution after a fixed number of generations or when a certain percentage of the total number of mutants has been produced. For the experiments of this paper, however, we can define new stopping conditions that help to compare the performance of the different techniques studied later on in Section 5.

As such, we have implemented a new termination condition that fits better with the evaluation method explained in Section 4.2. Because of the availability of an adequate test suite for the program under test, this condition stops the algorithm when reaching a given percentage P of the size of the minimal adequate test suite. In this way, the algorithm keeps producing new generations of mutants until satisfying the condition in Equation 3.

$$|Minimal\ selected\ test\ suite| \geq |Minimal\ adequate\ test\ suite| \times P \tag{3}$$

Using the example in Section 4.2 again, we could have established $P = 70\%$. This means that we want the algorithm to generate as many mutants as necessary to obtain a test suite with a size greater or equal than 70% of the size of the minimal adequate test suite. Recall that $|Minimal\ selected\ test\ suite| = 3$ and $|Minimal\ adequate\ test\ suite| = 5$, so the condition $3 \geq 5 \times 70\%$ is not met. Therefore, at least a new generation of mutants is needed. On the contrary, the algorithm would have stopped with $P = 60\%$ ($3 \geq 5 \times 60\%$).

This termination condition is helpful because, in the end, we can compare the number of mutants that the different techniques need to generate until reaching the same percentage of the size of the test suite.

5. Evaluation of EMT

5.1. Research questions

In this section, we aim at answering the following research questions:

RQ1: *Is EMT able to generate a lower percentage of mutants to find the same percentage of strong mutants than Random Selection (RS)?*

To answer RQ1. With this question, we seek to assess the ability of EMT to find strong mutants when compared to the random selection of mutants (that is, one mutant is randomly created in each generation). Both algorithms, EMT and RS, were executed under the following conditions:

1. We established 5 different termination conditions: finding 30%, 45%, 60%, 75% and 90% of the total number of strong mutants.
2. We counted the number of mutants generated until reaching each termination condition in 30 independent runs.

By using different termination conditions, we can compare these techniques with different levels of demand regarding the number of strong mutants to find.

RQ2: *What percentage of mutants does EMT generate to reach different percentages of the size of the minimal adequate test suite?*

To answer RQ2. This question intends to analyze how EMT behaves as the level of demand for test suite improvement increases, and how the percentage of mutants generated relates to the percentage of strong mutants found. EMT was executed under the following conditions:

1. We established 4 different termination conditions: $P = 70\%$, $P = 80\%$, $P = 90\%$ and $P = 100\%$[2].
2. We counted the number of mutants generated until reaching each termination condition in 30 independent runs.

Finally, we measured the percentage of strong mutants generated until stopping the genetic algorithm. These percentages allow us to know the number of strong mutants that is actually needed to enhance the test suite, and how it grows in relation to the percentage of test suite improvement.

RQ3: *Is EMT able to induce a greater refinement of the test suite than RS?*

To answer RQ3. Both algorithms, EMT and RS, were executed under the same conditions imposed to answer RQ2. Different statistics were measured to compare the performance of both techniques. Finally, we run statistical tests to know about the significance of the results.

5.2. Experiment configuration

The experiments were conducted using the tool *GiGAn*, which connects the genetic algorithm implemented in *GAmera* [18] with the mutation tool *MuCPP* [20] for C++. Namely, we analyzed eight case studies with respect to the set of class mutation operators implemented in *MuCPP*. Further details about *GiGAn* can be found in a previous paper [6]. The parameter values used for the genetic algorithm (see Sections 3.1 and 3.3) were those recommended in the literature [16]:

- The population size PS is 5% from the full set of mutants.
- In that population, 90% of the mutants are generated with reproductive operators; the remaining 10% of the mutants are generated randomly (N).
- Mutation and crossover operators are applied with probability $p_m = 30\%$ and $p_c = 70\%$ respectively.

[2]These conditions are more demanding than those imposed to answer RQ1 with the goal that the new and modified test cases appear in the minimal selected test suite.

Recall that the first generation is generated completely at random.

These programs were extracted from known open-source projects. Namely, *Matrix TCL Pro* (TCL), *Dolphin* (DPH), *MuParser* (MUP), *KMyMoney* (KMY), *Kate* (KAT), *TinyXML2* (TXM), *MySQLServer* (SQL) and *QtDom* (DOM). In most cases, we carefully studied the source code of those programs and narrowed it down to those classes (1) with a manageable number of mutants and (2) apparently well covered by the respective test suites; each of the test suites had to undergo a process of refinement oriented to kill surviving mutants, so those two conditions were necessary to ease such laborious and time-consuming task.

Additionally, these case studies were selected because they present quite different properties regarding number of mutants, percentage of strong mutants, size of the test suite and lines of code, among others. All these parameters can be seen separately in Table 1 (test suite) and Table 2 (size of the code and mutants generated). In general, in those programs with a high percentage of strong mutants, such as *KMY* (60.2%), *KAT* (80.1%) and, especially, *SQL* (84.1%), it was necessary a greater refinement of the test suite than in the rest of programs.

Table 1: Test suites sizes

	TCL	**DPH**	**MUP**	**KMY**	**KAT**	**TXM**	**SQL**	**DOM**
|*Original*|	17	61	13	241	158	57	271	46
|*New tests*|	7	9	9	7	15	5	23	10
|*Modified tests*|	3	5	2	10	1	3	17	4
|*Adequate*|	24	70	22	248	173	62	294	56
|*Minimal*|	15	22	17	34	23	17	43	25

Table 2: Case studies statistics

	TCL	**DPH**	**MUP**	**KMY**	**KAT**	**TXM**	**SQL**	**DOM**
Lines of code	3,228	3,667	2,723	13,709	4,261	2,620	2,280	2,117
Total mutants	137	219	226	322	385	614	683	1,146
Operators applied	6	14	10	15	17	14	13	15
Valid mutants	135	208	207	251	261	433	530	681
Strong mutants	45	103	133	151	209	159	446	348
% Strong mutants	33.3%	49.5%	64.2%	60.2%	80.1%	36.7%	84.1%	51.1%

To summarize, the independent variables in these experiments are the configuration of the algorithm, the set of mutation operators, case studies with their respective test suites and different ground truths (depending on the methodology, percentage of strong mutants or generated mutant-adequate test suites). Our cost-effectiveness measure, the dependent variable, is the percentage of mutants generated to reach the aforementioned termination conditions (depending on the methodology, different percentages of all strong mutants found or different percentages of the size with respect to the minimal adequate test suite).

5.3. Results

The three following subsections show the results of the conducted experiments according to our three research questions:

Strong mutants

Table 3 collects statistics (average, minimum and maximum result, and standard deviation (SD)) about the percentage of mutants generated in the 30 executions using EMT and RS in each case study. This table breaks the results down into the five termination conditions, from 30% to 90%, which allows observing how the percentage of mutants generated evolves in relation to the percentage of strong mutants to find. Notice that the lower the percentage of mutants generated, the better the performance of the technique.

Analyzing the average results, we can see that EMT outperforms RS in all case studies and termination conditions, except for *MUP* and *KMY* (where EMT only produces a lower percentage of mutants with 30% as stopping condition). These results hold for the rest of statistics, except for a few cases. EMT also shows a lower standard deviation than RS in general.

The best results are obtained in *TXM* and *DOM*, where the gap between techniques is about 10% in some cases (see figures marked in bold in Table 3). On the contrary, the difference is not meaningful in *SQL*. We can also highlight that, in general, the distance between EMT and RS narrows with the more demanding condition (90%). For instance, this difference is 5.6% in the case of *TXM* in contrast with the three previous stopping conditions (8-10%).

Test suite improvement with EMT

Table 4 shows the statistical results of the 30 executions of EMT in a similar way as Table 3. In this case, the results represent the percentage of mutants generated before reaching four different levels of improvement of the test suite (from 70% to 100% of the minimal adequate test suite in each case study). Recall that the more mutants are generated, the less efficient is the algorithm. Also, note that we only focus on the results of EMT execution in this subsection.

Remarkably, the genetic algorithm only needs to generate 34.7% and 49.3% of the mutants on average in *DOM* and *TXM* respectively to complete the minimal adequate test suite. Similarly, EMT does not require to generate more than 44% of the mutants associated with each of the programs to reach 70% of their respective minimal adequate test suites. *DOM*, with 11.35% of mutants generated, and *DPH*, 43.85%, show the best and worst result in this aspect respectively. Therefore, it is not necessary to generate a great number of mutants to reach considerable improvements in the initial test suite, contrary to expectations on analyzing Table 3. In broad terms, these percentages decrease for the more complex programs (in terms of the number of mutants) and contrarily increases when the termination condition becomes more demanding, especially when $P = 100\%$.

Figure 7 depicts the relation between the new methodology and the percentage of strong mutants generated before reaching the four different percentages of

Table 3: Statistics about the percentage of mutants generated by EMT and RS to reach 30%, 45%, 60%, 75% and 90% of the set of strong mutants. Results derived from 30 executions.

Program	**30%**		**45%**		**60%**		**75%**		**90%**	
	EMT	RS	EMT	RS	EMT	RS	EMT	RS	EMT	RS
TCL										
Avg.	23.45	28.61	37.59	43.13	53.55	57.90	67.59	72.53	84.33	88.25
Min.	13.13	16.78	25.54	31.38	40.14	44.52	51.09	60.58	70.07	79.56
Max.	37.22	40.87	51.09	54.74	65.69	70.07	79.56	79.56	92.70	94.16
SD	5.44	6.03	6.67	5.17	6.10	5.63	6.98	3.89	5.21	3.57
DPH										
Avg.	28.35	29.89	41.94	45.76	55.19	59.98	69.87	75.11	85.35	89.07
Min.	24.65	24.65	38.35	38.35	50.68	52.96	62.55	67.57	78.99	82.64
Max.	33.33	35.15	47.03	53.42	59.81	67.12	76.71	81.27	90.41	93.15
SD	2.11	3.07	2.28	3.98	2.10	4.30	3.57	3.57	2.67	2.86
MUP										
Avg.	30.07	28.98	46.22	43.86	61.59	58.87	76.41	74.10	90.73	89.45
Min.	26.10	23.89	40.70	37.16	55.30	52.21	71.68	69.02	86.72	84.95
Max.	35.84	33.18	50.88	48.67	66.81	63.27	80.53	77.43	93.36	92.03
SD	2.79	2.22	2.60	2.90	2.49	2.71	2.17	2.27	1.62	1.78
KMY										
Avg.	29.03	29.93	44.91	44.14	60.92	59.26	76.37	74.60	90.27	88.88
Min.	24.53	24.22	40.37	36.33	55.59	51.55	70.49	69.56	87.57	83.85
Max.	34.16	35.09	49.37	50.00	65.52	64.28	81.36	79.81	92.54	92.23
SD	2.23	2.86	2.49	3.24	2.16	3.15	2.29	2.63	1.39	1.92
KAT										
Avg.	26.46	29.42	40.92	44.67	54.53	59.64	68.58	74.32	85.29	89.62
Min.	23.63	25.19	36.88	40.00	48.31	54.80	64.67	69.61	81.55	87.01
Max.	30.12	32.20	45.19	49.87	58.44	64.93	73.50	79.22	88.83	91.94
SD	1.53	2.11	2.03	2.31	2.22	2.62	2.23	2.21	1.86	1.35
TXM										
Avg.	24.09	29.09	36.62	44.50	**49.74**	**59.23**	**64.91**	**74.93**	84.32	89.98
Min.	20.52	22.63	31.92	38.27	44.78	54.23	60.58	69.70	77.85	85.01
Max.	27.85	35.01	41.04	51.14	56.35	65.63	71.49	80.61	89.73	93.64
SD	1.61	3.14	2.34	3.40	3.05	3.03	2.59	2.78	3.34	1.78
SQL										
Avg.	29.50	30.36	45.30	45.57	60.26	60.35	74.59	75.24	89.36	89.97
Min.	27.96	27.37	43.92	43.04	57.83	57.68	72.91	72.18	87.84	88.14
Max.	31.47	32.35	47.58	47.73	61.78	63.25	77.01	78.18	90.62	91.21
SD	0.87	1.20	0.89	1.23	0.95	1.35	1.02	1.57	0.63	0.86
DOM										
Avg.	21.20	29.49	**34.86**	**44.16**	52.21	59.38	69.96	74.43	87.84	89.76
Min.	19.02	25.65	32.28	39.09	46.59	54.88	66.05	71.64	83.33	86.21
Max.	23.03	34.29	37.26	49.47	57.06	63.96	73.38	78.88	90.13	93.71
SD	1.01	2.14	1.26	2.28	2.39	2.51	1.98	2.00	1.60	1.58

test suite improvement. In general, the results in Figure 7 (strong mutants) and Table 4 (mutants generated) are quite similar. For instance, EMT generated 40.9% of the mutants in *KMY* for $P = 70\%$, and that subset contains 41% of the full set of strong mutants. However, in most cases, the percentage of strong

Table 4: Statistics about the percentage of mutants generated by EMT and RS to reach 70%, 80%, 90% and 100% of the minimal adequate test suite. Results derived from 30 executions.

Program	**70%**		**80%**		**90%**		**100%**	
	EMT	RS	EMT	RS	EMT	RS	EMT	RS
TCL								
Avg.	31.62	26.78	42.06	40.60	49.24	47.85	75.79	80.05
Min.	16.78	10.94	20.43	15.32	25.54	25.54	48.90	34.30
Max.	50.36	44.52	67.88	57.66	75.91	68.61	100.00	97.08
SD	9.92	7.92	12.01	12.03	13.41	12.78	16.11	13.43
DPH								
Avg.	43.85	48.87	54.79	59.49	66.33	71.08	88.24	91.63
Min.	32.42	25.11	40.18	33.33	52.51	40.63	68.94	72.60
Max.	57.07	80.36	82.19	84.93	84.93	88.58	99.08	100.00
SD	6.69	10.23	9.69	10.88	8.61	10.45	8.07	8.10
MUP								
Avg.	36.93	39.48	49.08	53.96	63.99	73.06	82.74	91.10
Min.	21.23	18.14	31.41	32.74	47.34	41.15	54.42	61.50
Max.	57.52	61.06	65.04	83.62	79.20	91.59	99.11	100.00
SD	7.75	9.34	9.40	10.68	9.19	12.21	12.00	8.19
KMY								
Avg.	40.93	38.46	50.47	53.13	64.65	70.27	89.26	94.66
Min.	31.36	25.77	37.88	35.40	47.51	50.00	73.60	81.36
Max.	52.79	55.27	65.52	67.08	80.12	84.16	100.00	100.00
SD	5.86	7.56	7.12	8.94	7.88	8.40	7.83	5.68
KAT								
Avg.	32.96	43.02	**42.24**	**54.24**	57.99	66.27	88.20	93.19
Min.	19.22	24.41	27.01	28.57	34.02	43.11	64.67	73.24
Max.	47.79	65.71	65.19	72.46	76.10	84.41	99.74	99.74
SD	7.10	8.98	9.03	10.25	12.22	10.50	8.88	6.11
TXM								
Avg.	16.42	19.92	21.92	32.57	31.93	46.79	**49.30**	**75.92**
Min.	9.93	9.93	13.02	13.84	20.52	24.75	24.10	32.41
Max.	26.38	39.08	29.31	64.65	46.09	86.80	65.14	98.53
SD	3.92	7.48	4.71	11.86	7.13	15.21	10.77	17.27
SQL								
Avg.	30.42	35.87	42.80	52.95	**58.19**	**70.53**	87.25	94.29
Min.	21.08	34.84	28.69	34.84	44.07	55.34	62.81	78.62
Max.	41.14	49.04	54.02	74.37	72.03	93.70	99.12	99.85
SD	5.10	7.33	6.70	9.08	8.49	10.12	8.99	5.08
DOM								
Avg.	11.35	19.78	16.45	34.68	21.41	49.04	**34.69**	**80.51**
Min.	7.06	10.12	10.29	19.89	11.95	26.96	25.65	52.00
Max.	17.53	41.97	25.39	53.49	35.86	81.15	55.58	99.91
SD	2.48	6.96	3.98	9.19	5.00	12.85	7.89	12.42

mutants is over the percentage of mutants generated. This fact is especially evident for $P = 100\%$ in the case of *TXM* (49.3% mutants generated - 58.5% strong mutants) and *DOM* (34.7% - 44.3%).

It is meaningful that most of the case studies show an almost linear increase in the percentage of strong mutants used to improve the test suite (see Figure 7).

However, $P = 100\%$ clearly breaks that linearity, with noticeable increases in all the programs (e.g., the increase from $P = 90\%$ to $P = 100\%$ is 21% in *DPH* and 25% in *KAT*). In any case, these increases are in line with the increases in the percentage of mutants produced to reach the whole test suite (see differences between the columns $P = 90\%$ and $P = 100\%$ in Table 4).

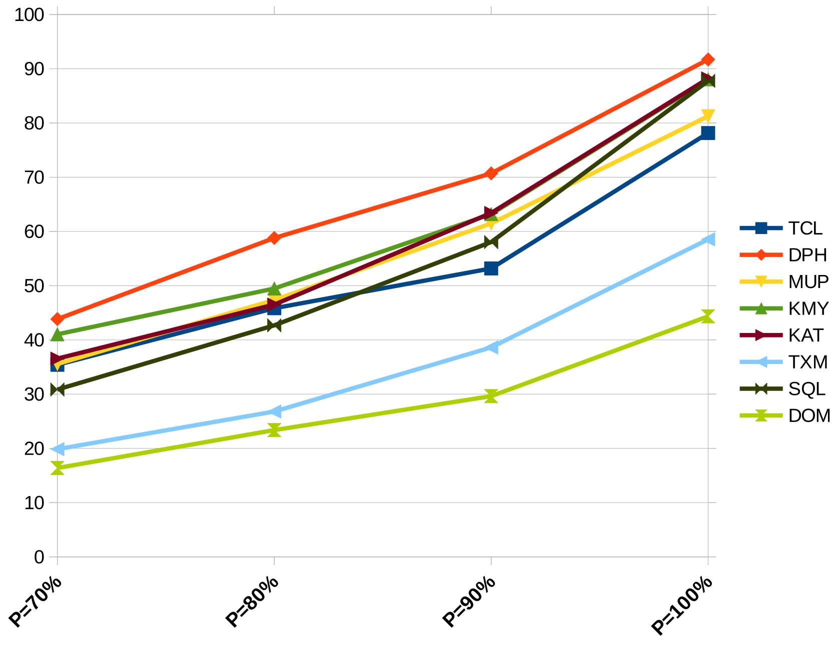

Figure 7: Percentage of strong mutants produced before reaching 70%, 80%, 90% and 100% of the minimal adequate test suite in each case study.

Comparison of the test suite improvement with EMT and RS

Table 4 also furnishes the results of the random algorithm using the new methodology. Focusing on the average, we can observe that EMT performs better than RS in almost all cases, being the differences in *TXM*, *SQL* and *DOM* especially worthy of attention. Despite this, the difference is favorable for RS in *TCL* (except for $P = 100\%$) and *KMY* with $P = 30\%$. As it happened in Table 3 with regard to strong mutants, the standard deviation when using EMT is generally lower than using RS.

Figure 8 shows these differences graphically to help interpret the evolution of the results. We can observe that, in general, the difference favorably increases for EMT in relation to the termination condition. As such, it is remarkable that EMT produces better results than *RS* for $P = 100\%$ in *TCL*, in contrast to the rest of stopping conditions. Nonetheless, the highest difference is not always reported for $P = 100\%$, as in the case of *DPH* (highest gap for $P = 70\%$), *KAT* ($P = 80\%$) and *MUP*, *KMY* and *SQL* ($P = 90\%$).

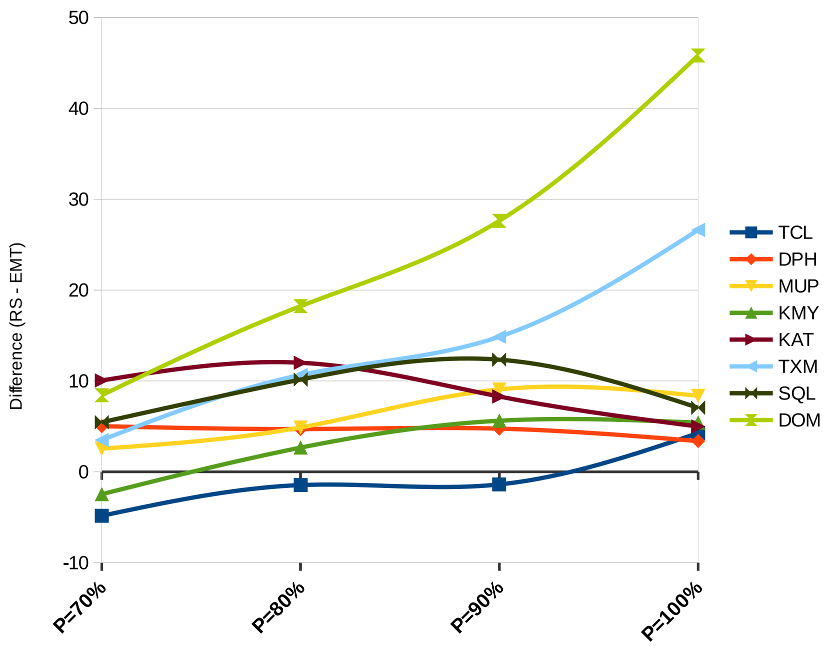

Figure 8: Difference (measured as $RS - EMT$) between the percentage of mutants generated by EMT and RS to reach 70%, 80%, 90% and 100% of the minimal adequate test suite in each case study. Positive differences mean that EMT gets a better result than RS, and vice-versa.

Table 5 collects the results of non-parametric statistical tests based on the results in Table 4. Namely, Mann-Whitney U test and Vargha and Delaney's A_{12} to measure the effect size. The results of the application of Mann-Whitney U lead us to accept that the percentage of mutants that EMT needs to produce to reach different sizes of the minimal adequate test suite is lower than with RS with 0.05 significance level, except for {*TCL*, 100%}, {*DPH*, 80% and 100%}, {*MUP*, 70-80%}, {*KMY*, 80%} and {*TXM*, 70%}. We can describe the effect size as large for {*MUP*, 90-100%}, {*KMY*, 100%}, {*KAT*, 70-80%}, {*TXM*, 80-100%}, *SQL* and especially *DOM*.

5.4. Discussion

This section discusses the above-presented results focusing on the differences between metrics and selection techniques, including selective mutation. Finally, the RQs are answered:

Test suite improvement vs strong mutants

In the view of the percentage of strong mutants found, it seems that *EMT* is less effective when the proportion of strong mutants in the program is high (among other factors that may affect the results). In fact, the worst results in terms of strong mutants found were obtained in *MUP*, *KMY* and *SQL*, where

Table 5: Results of Mann-Whitney U and Vargha and Delaney's A_{12} statistical tests based on the results of EMT and RS in each program and stopping condition. Favorable differences for RS are marked with a dash (-). Differences (D) [21]: small - S (under 0.44), medium - M (under 0.36) and large - L (under 0.29).

Prog.	*70%*			*80%*			*90%*			*100%*		
	p-value	A_{12}	D	p-value	A_{12}	D	p-value	A_{12}	D	p-value	A_{12}	D
TCL	-	-	-	-	-	-	-	-	-	0.304	0.42	S
DPH	0.030	0.34	M	0.060	0.36	S	0.026	0.33	M	0.068	0.36	S
MUP	0.206	0.40	S	0.117	0.38	S	**0.002**	**0.27**	**L**	**0.004**	**0.28**	**L**
KMY	-	-	-	0.246	0.41	S	0.008	0.30	M	**0.004**	**0.28**	**L**
KAT	**3.83e-5**	**0.19**	**L**	**3.26e-5**	**0.19**	**L**	0.016	0.32	M	0.014	0.32	M
TXM	0.081	0.37	S	**3.59e-5**	**0.19**	**L**	**2,96e-5**	**0.19**	**L**	**2.57e-7**	**0.11**	**L**
SQL	**5.66e-6**	**0.16**	**L**	**1.53e-5**	**0.17**	**L**	**2.60e-5**	**0.18**	**L**	**9.77e-4**	**0.25**	**L**
DOM	**6.01e-8**	**0.09**	**L**	**3.01e-10**	**0.03**	**L**	**7.03e-11**	**0.01**	**L**	**3.51e-11**	**0.00**	**L**

the total percentage of strong mutants is 64.2%, 60.2% and 84.1% respectively (see Table 2). The contrary also holds, taking into account that the best result was achieved in *TXM* (strong mutants = 36.7%) and that the results in *TCL* were also positive. This makes sense because the genetic algorithm is good at finding strong mutants when these are mainly concentrated on certain areas of the code and are generated by a subset of mutation operators.

It is curious to observe that, while RS outperformed EMT in finding strong mutants in *MUP* and *KMY*, Table 4 shows that EMT is now better at selecting mutants for the refinement of the test suite in these programs (especially in *MUP* with $P = 90\%$). Exactly the contrary happens in the case of *TCL*, since the percentages in Table 4 contradict its positive results in Table 3. This fact makes evident that there are other important factors beyond the initial number of strong mutants, such as the way in which those strong mutants are distributed. For example, it is likely that the results for *TCL* are affected by the few available mutants (137) and the few mutation operators that produce them (6), as shown in Table 2: the low number of mutants reduces the search space and the low number of mutation operators reduces the effectiveness of reproductive operators. Even in the case of *SQL*, where the original test suite was not mature enough and the percentage of strong mutants found by EMT was not promising, the new method shows that EMT is able to quickly find more useful mutants for the refinement of the test suite when compared to RS.

In broad terms, there exists a correlation between the percentages of strong mutants found and mutants generated to reach certain improvement, that is, the figures in Table 4 and Figure 7 are similar. Again, this fact suggests that the real problem is not about finding a huge number of strong mutants but those with the ability to induce new test cases. In other words, it is more important the nature of the mutants selected than the number of strong mutants found, and EMT shows ability at finding those subgroups of valuable mutants.

EMT vs RS

The overall results give evidence that the application of mutation testing for test suite improvement can greatly benefit from the evolutionary approach. In comparison to the random selection of mutants, EMT gets better results in both the measurement of strong mutants and test suite improvement. This situation becomes more evident with the proposed methodology, where we can find significant differences between both techniques, especially in the programs with a greater number of mutants. Beyond the average results, it is interesting to analyze the standard deviation, which indicates that EMT is also more stable overall. For instance, we can see in Table 4 that the standard deviation associated to RS executions is 11.86 in *TXM* and $P = 80\%$. This translates into executions that range between 13.8% and 64.6% of mutants generated in the best and worst case. EMT, however, is less prone to such great variations.

Figure 8 suggests that the benefits derived from the application of EMT when compared to RS are greater when the desired improvement is high. Still, it seems sometimes that EMT experiences some problems to achieve the complete minimal adequate test suite, as in the case of *DPH*, *MUP*, *KMY*, *KAT* and *SQL*: the difference with RS in those five programs tends to go down at some point. This fact reveals that, in some cases, EMT may get stuck in local maxima and, therefore, may find difficult to generate some valuable mutants which are either:

- Produced by seemingly low-quality mutation operators. This means that some mutation operators can generate many mutants with a low fitness function (weak mutants), but a few of them are contrarily strong.
- Created in some apparently well-covered areas of the code.

Comparison with other selection techniques

Selective mutation [13] works under the assumption that the mutants from some operators can be discarded without losing much effectiveness. As it has been commented previously, EMT should be more effective when a subset of the operators concentrates many of the strong mutants. Therefore, repeating the experiments using selective mutation can shed light on this fact.

Selective mutation requires adding all mutants from each selected operator even if only a small number of its mutants are actually useful. In this experiment, instead, we combine selective mutation with random selection (SM+RS) for a fairer comparison with EMT. To this end, we follow the next two steps:

1. **Select mutation operators at random** until the subset of selected operators reaches the termination condition (P) set.
2. **Select mutants at random** from the selected subset of mutation operators, again, until the selected mutants reach the same stopping condition.

Thanks to the first step, we ensure that the second step always comes to an end (in the worst case, all mutants generated by the subset of operators will be selected). Notice that SM+RS can perform worse than RS when operators with a high ratio of useful mutants to mutants generated (H from now on) are left out: this may decrease the probability to find a useful mutant for the second step. The above process is repeated 30 times. Given that high values for P could lead to the selection of all operators, we focus on $P = 80\%$ in this experiment.

Table 6: Statistics about the percentage of mutants generated by EMT and SM+RS (random selection using a selective set of operators) with $P = 80\%$. Results derived from 30 executions.

Program	**Avg**		**Max**		**Min**		**SD**	
	EMT	SM+RS	EMT	SM+RS	EMT	SM+RS	EMT	SM+RS
TCL	42.1	39.4	67.9	70.8	20.4	**19.7**	12.0	10.9
DPH	54.8	55.4	82.2	74.4	40.2	**34.7**	9.7	11.5
MUP	49.1	50.5	65.0	71.7	31.4	**22.6**	9.4	13.4
KMY	50.5	60.6	65.5	80.4	37.9	**37.6**	7.1	9.9
KAT	42.2	52.8	65.2	85.2	27.0	31.2	9.0	13.4
TXM	21.9	47.2	29.3	76.5	13.0	14.2	4.7	21.2
SQL	42.8	50.6	54.0	79.6	28.7	**24.3**	6.7	16.0
DOM	16.4	53.7	25.4	83.9	10.3	10.3	4.0	20.7

Table 6 shows the difference in performance of SM+RS with respect to EMT. As it can be seen, EMT gets better results than the selective technique (except for *TCL* again). Looking at the best results (*Min*), the fact that SM+RS achieves a better result than EMT in several cases (highlighted in bold) suggests that effectively some mutation operators are more fruitful than others: when operators with high H are selected in first place, the probability to find useful mutants at random increases. Contrarily, when those operators are not selected, that probability decreases, leading to poor results (see column *Max*). This fact is also reflected by the great increase in the standard deviation. Therefore, further research is needed in order to establish a general subset of mutation operators. At this point, we have to remark that, if such subset exists, selective mutation could be integrated into EMT to improve its performance. This study is completed with the repetition of the same statistical tests previously applied, comparing the results of EMT and SM+RS. The results in Table 7 are statistically significant for the complex programs and four of them show a large effect size.

Recent papers have also proposed more sophisticated ways of selecting mutants than pure random selection, such as selecting a statement or method at random prior to selecting the mutant [22]. As explained in Section 3.1, EMT selects a percentage N of mutants randomly in each generation. In further experiments, it would be interesting to add these random techniques as an alternative and assess the impact on the performance.

Answer to research questions

RQ1: *Is EMT able to generate a lower percentage of mutants to find the same percentage of strong mutants than RS?*

Yes, EMT generated fewer mutants than RS in most case studies and percentages of strong mutants required. Differences around 10% between both

Table 7: Results of Mann-Whitney U and Vargha and Delaney's A_{12} statistical tests based on the results of EMT and SM+RS in each program and $P = 80\%$.

Program	***80%***		
	p-value	A_{12}	D
TCL	-	-	-
DPH	0.976	0.50	-
MUP	0.544	0.45	-
KMY	**8.93e-5**	**0.21**	**L**
KAT	**6.73e-4**	**0.24**	**L**
TXM	**1.29e-6**	**0.14**	**L**
SQL	0.044	0.35	M
DOM	**1.37e-8**	**0.07**	**L**

techniques could be observed in some cases, while these differences were not significant in the programs with a high number of strong mutants.

RQ2: *What percentage of mutants does EMT generate to reach different percentages of the size of the minimal adequate test suite?*

The percentage of mutants generated varies among programs and levels of demand, from 11% in *DOM* with $P = 70\%$ to 89% in *KMY* to reach an adequate test suite. In general, even with reduced subsets of mutants, EMT can lead to notable improvements in the quality of test suites, especially in programs with a considerable number of mutants (for instance, 31% of the mutants in *TXM* can lead to design 90% of the test cases regarding the adequate test suite). Interestingly, the percentage of strong mutants found to achieve such improvements is not far from the percentage of mutants generated, which reveals the importance of finding the appropriate strong mutants.

RQ3: *Is EMT able to induce a greater refinement of the test suite than RS?*

Yes, EMT shows a greater potential than RS to guide the tester on the test suite improvement overall. Thanks to the new methodology, we found out that the differences between both algorithms were more notable than when finding a percentage of strong mutants. Overall these results have been shown to be statistically significant. In some programs, the difference narrowed at some point, which suggests that the evolutionary approach may find difficult to select some interesting mutants when they are isolated from the rest of useful mutants.

5.5. Threats to validity

Algorithms. The performance of the genetic algorithm may vary depending on the values given to different parameters (e.g., *PS* or *N*), but the best configuration is unknown in practice. As such, we have used the same configuration that Domínguez Jiménez et al. [16] found to be optimal in their experiments.

Assessing randomized algorithms requires several executions to minimize the impact of their stochastic nature on the results. We run the techniques 30 times, a common number of runs according to the guide by Arcuri and Briand [9].

Methodology to evaluate performance. The experiments in this paper are inevitably affected by the nature of the test suites. The variability in the test cases that can be designed for each of the case studies supposes a threat to the validity of the proposed methodology. Nevertheless, this threat is countered by the fact that we do not intervene in the simulation process of test suite improvement, which prevents taking decisions that could introduce a bias.

Even though, in general, test cases are designed to test a specific functionality, sometimes it is required to set a test scenario that covers a broader area of the code. That means that a test case could exercise and kill other mutants different from those that are expected to cover. In such cases, it is unlikely that all test cases that kill a mutant could be designed only by reviewing that mutant. This fact poses a threat for the simulation process of test suite refinement: the selected matrix can suggest that the subset of mutants generated could induce a subset of test cases, which might differ from those test cases that would be actually designed after mutant inspection. We remark, however, that the new and modified test cases were designed and completed with the utmost care, seeking to address only a particular functionality of the program in each test case. In any case, this fact affects the techniques used in this study evenly.

Recently, Chekam et al. [23] provided evidence that relying on the clean program assumption (i.e., the program under test does not contain any faults) poses a potential threat to validity in test assessments. Future evaluations should consider the methodology proposed in that paper to avoid that assumption and observe if the same results hold. That study also observes that increases in the fault-revealing ability of a test suite are only achieved when reaching high levels of coverage. In our study, we presented the percentage of mutants required to obtain from 70% to 100% of the adequate test suite. Finally, test order has no impact on these experiments because we used an exact algorithm for test suite minimization (all available minimal test suites are of the same size).

The set of equivalent mutants was determined after carefully inspecting all surviving mutants in these programs. There is the threat, however, that we had classified some mutants as equivalent when new test cases could be actually designed to kill them. On the contrary, all non-equivalent mutants are killable with certainty since we added test cases to kill them.

Generalization. Representativeness of the programs under study is a common threat to validity of the results. Nevertheless, we have selected eight applications of different nature in terms of complexity, lines of code, number of mutants or mutation operators applied. Their respective test suites are also quite different among them regarding the size of the original set and the need for refinement to reach an adequate test suite. Besides this, some of the test suites make a more exhaustive use of the classes and their members than others. This diversity minimizes the threat to the generalization because it avoids the partial perspective of the individual case studies.

We have not computed the time of the executions: each of the programs has a different mutant generation, compilation and test suite execution time. For instance, while the execution of *DPH* is a matter of hours, the execution of *SQL* requires several days because each mutation causes a full recompilation of the project. This makes difficult to give an average measure of the time saved by using EMT beyond the difference in the percentage of mutants generated.

In terms of cost, provided that not all mutants are generated, EMT reduces the times of mutant generation, mutant compilation and execution, and mutant review (when a subset of live mutants is indirectly discarded). Still, EMT incurs two kinds of additional cost when compared to the conventional mutation process: the execution of all test cases –required to obtain the execution matrix and compute the fitness function– and the execution of the genetic algorithm. By matching the code coverage of the test suite with the mutation locations, we can greatly reduce the former [24]; as reported by Domínguez et al. [16], the time taken by the latter is marginal when compared to the whole execution time, especially in cases like *SQL* where the compilation time clearly predominates.

In terms of test suite improvement, EMT should induce the same level of refinement than reviewing all mutants as long as all useful mutants are selected by the genetic algorithm. However, the lower the percentage of mutants generated, the higher the risk that some of those test cases in the adequate test suite are not designed.

We should remark that these experimental procedures were carried out using a set of class mutation operators, which are known to produce many equivalent mutants when compared to traditional operators [25]. Therefore, it is unknown if the results hold in other contexts.

6. Lessons learned and future development

Test suite

The design of the original test suite impacts the effectiveness of the genetic algorithm in several ways:

- Firstly, an important aspect is how mature the initial test suite is, which determines the percentage of strong mutants and how much information is available to guide the search. Indeed, this factor was purposely assessed with the inclusion of *SQL* in the study, since only 84 out of 683 mutants were initially killed. The results in *TXM* (dead mutants = 274 out of 614) lead us to think that EMT can be more helpful in cases where the current test suite is at an advanced stage. However, the results in *KAT* and *SQL* show that EMT is also effective even when the information is scarce.

- Secondly, another aspect to take into consideration is whether test cases are *specific* (i.e., each of them tests a particular functionality) or *general* (i.e., they test several functionalities at the same time or cover a broad area of the code). The reason is that general test cases may unintentionally hide valuable mutants. This would happen when the same functionality is covered by several test cases: in that case, the mutant would be detected by several test cases and the fitness function would attach it a low value. We became aware of this possibility when experimenting with *TCL*, as the accompanying test suite was comprised of large test cases (in fact, some of them were split for these experiments as they merged completely different functionalities). This fact might also be the cause of the poor results for this program. As a result of the above commented, the genetic algorithm should be more effective with specific and well-designed test suites.

6.1. Mutants and mutation operators

In the light of the results in the more complex programs, it seems that the genetic algorithm performs better with a wide search space, that is, when there is a considerable number of mutants. It is unclear, though, whether the interesting groups of mutants for the test suite improvement will be concentrated on certain areas (or generated by a subset of the operators) or, conversely, will be spread all over the code (and mutation operators). The experiments by Delgado-Pérez et al. [12] suggest that all mutation operators can produce valuable mutants, even if they are created by not very productive operators. This is the reason why the search can have difficulties finding some strong mutants. This leads us to think that, in the future, we could refine the algorithm to help it escape local maxima and ease the selection of those concealed strong mutants. This could be done by favoring the generation of mutants from all operators in the part of mutants generated randomly in each generation (see Figure 1). The fact that all operators can produce useful mutants could be extrapolated to the distribution of mutants across the code: a valuable mutant could be generated anywhere. As such, we could favor that all different areas of the code were covered by at least one mutant. These improvements deserve further investigation.

Invalid mutants can also have an influence on the performance of EMT. Invalid mutants do not lead to the refinement of the test suite and, therefore, they are not assigned a fitness nor selected to produce new individuals. This means that the genetic algorithm should be able to escape from places and operators that produce several of these mutants. This can be one of the factors for the good results obtained in *DOM* since most of the invalid mutants were generated by a single mutation operator. As a result, avoiding the generation of invalid mutants can be a further advantage of using this search-based approach.

6.2. Equivalent mutants

Despite the positive results shown in this paper, there is a major problem that EMT does not directly address: the existence of equivalent mutants. Given that we only generate a subset of the mutants, the number of equivalent mutants consequently decrease. However, potentially-equivalent mutants still can turn out to be equivalent, as this is an undecidable problem. Some techniques have been proposed recently to alleviate the effect of this kind of mutants [26, 27]. In future refinements of the algorithm, we could integrate some of those techniques to provide EMT with information that helps differentiate between non-equivalent and equivalent mutants. In that way, we could avoid that mutants flagged as equivalent (with certainty or high probability) are selected to breed new mutants. We have to note, however, that it is currently unknown whether there is a correlation among equivalent mutants (i.e., whether or not equivalent mutants tend to appear in the same areas of the code or are produced by a subset of the operators). Investigating this correlation is required to support this possible improvement. Otherwise, we could be misleading the search.

7. Related work

Harman et al. [1] studied in 2009 the techniques applied in search-based software engineering so far, classifying the works found in the literature by areas, including testing. Recently, Silva et al. [2] focused on research studies that analyzed the use of search-based techniques in mutation testing. As it can be seen from the works included in these review studies, search-based strategies have been applied to reduce the cost with regard to several problems related to software testing. However, the application of this kind of techniques for the selection of mutants has mostly been addressed in recent years. Reducing the high cost of mutation testing has been a key issue tackled by researchers in this field [10]. As such, different cost reduction techniques have been proposed: mutant sampling [28], mutant clustering [29], selective mutation [13], weak mutation [30] or high order mutation [31] have been successfully applied in diverse contexts. The novelty of the approach in this paper lies on the application of an evolutionary strategy to select mutants for the improvement of the test suite. Therefore, the technique uses the mutants as objectives for improving test suites instead of using them as means for test assessment.

Adamopoulos et al. [32] were the first to propose a co-evolutionary approach in order to evolve mutant population and test cases in parallel. Later, Oliveira et al. [33] also applied a similar approach but using a new representation with new genetic operators. Domínguez-Jiménez et al. [16] devised an evolutionary algorithm for the refinement of the test suite at a lower cost by selecting a subset of the mutants, which they called Evolutionary Mutation Testing. They developed a genetic algorithm [8] in the tool *GAmera* [18] and applied the algorithm to test three WS-BPEL compositions. The same technique was later implemented for C++ code with the development of the tool *GiGAn* [6]. Our work supports the findings pointed out by Domínguez-Jiménez et al. [16] regarding the ability of the technique to find strong mutants, but we go a step beyond by measuring the improvement that can be achieved thanks to the mutants selected in eight programs of varying nature. Swarchz et al. [34] also made use of a genetic algorithm for the enhancement of the test suite based on mutations spread throughout the code and that cause a high change in the program state.

The use of search-based techniques has shown to be especially useful to reduce the number of mutants generated in high order mutation testing since there are many more high order mutants (HOMs) than first order mutants (FOMs). More specifically, these techniques have been used to find subsuming HOMs [35, 36], that is, HOMs that are more difficult to kill than their constituent FOMs. Harman et al. [37] provided a more restrictive fitness function to find strongly subsuming high order mutants (SSHOMs). Lima et al. [38] recently compared HOM-based strategies and traditional strategies, such as random mutant selection, selective mutation and search-based mutant selection (genetic algorithm). This comparison was based on the number of mutants, the number of test cases and the mutation score. Randomly selecting 20% of mutants (random mutant selection) and removing the five most prolific operators (selective mutation) were the best strategies overall, though most of the strategies performed in a similar way.

While EMT follows a single-objective approach, several related works approach the selection of mutants as a multi-objective optimization problem. In this category, we can cite the work by Banzi et al. [39], where a genetic algorithm is used for the selection of mutation operators instead of individual mutants. The authors of that study sought to maximize the mutation score and minimize the number of mutants generated. The work by Lima and Vergilio [40] also follows a multi-objective approach for the selection of second order mutants (SOMs). They considered three different objectives: number of SOMs generated, ability to reveal subtler faults and capacity to replace the constituent FOMs. Likewise, Langdon et al. [41] tried to find hard to kill HOMs as similar as possible to the original program based on a multi-objective approach, showing that these HOMs could simulate complex faults beyond those modeled with FOMs.

Despite the above commented, most of the research regarding search-based software testing has been focused on the problem of test case generation [14, 42]. Note that the term Evolutionary Mutation Testing was used in another work but with a different purpose: their authors presented a fitness function based on the information provided by mutants to find effective test cases for object-oriented systems [43]. The problem of test data generation driven to kill mutants for object-oriented programs has also been addressed by Shamshiri et al. [44]. They reported that there is no significant difference between guiding the test case generation with a random or a genetic algorithm for this kind of applications.

8. Conclusion

The application of search-based techniques to solve software engineering problems requires a deep evaluation that shed light on its benefits. In this paper, we have addressed the use of a genetic algorithm to reduce the generation of mutants for test suite enhancement. To improve the understanding about the effects of its application, we have proposed and assessed in depth an evaluation method that takes into account the ability of each mutant to actually refine the test suite and not only whether it is a strong mutant or not.

The experiments in this paper support the assumption that an evolutionary approach can help reduce the cost of applying mutation testing. It is relevant the potential shown by this technique to induce the improvement of the test suite even with reduced subsets of mutants. Among other interesting results, the evolutionary algorithm works better when we seek for great improvements of the test suite, and with large sets of mutants –exactly when a cost reduction technique becomes more necessary.

We have also hinted at possible future enhancements for the genetic algorithm based on the lessons learned and recent studies. In this sense, we foresee that the new evaluation method will be particularly useful. For instance, reducing the probability of selecting equivalent mutants could result in a fewer number of strong mutants found, which would be misleading. However, the new evaluation method could detect a greater refinement of the test suite in case of a reduction in the number of equivalent mutants selected.

9. Acknowledgement

This work was partially supported by the European Commission (FEDER) and the Spanish Government projects DArDOS (TIN2015-65845-C3-3-R) and the Excellence Network SEBASEnet (TIN2015-71841-REDT).

10. References

[1] M. Harman, S. A. Mansouri, Y. Zhang, Search based software engineering: A comprehensive analysis and review of trends techniques and applications, Department of Computer Science, King's College London, Tech. Rep. TR-09-03.

[2] R. A. Silva, S. do Rocio Senger de Souza, P. S. L. de Souza, A systematic review on search based mutation testing, Information and Software Technology 81 (Supplement C) (2017) 19 – 35. doi:10.1016/j.infsof.2016.01.017.

[3] M. Papadakis, N. Malevris, Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing, Software Quality Journal 19 (4) (2011) 691. doi:10.1007/s11219-011-9142-y.

[4] J. Kim, B. You, M. Kwon, P. McMinn, S. Yoo, Evaluating CAVM: A New Search-Based Test Data Generation Tool for C, Springer International Publishing, Cham, 2017, pp. 143–149. doi:10.1007/978-3-319-66299-2_12.

[5] G. Fraser, A. Arcuri, EvoSuite: Automatic test suite generation for object-oriented software, in: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, ESEC/FSE '11, ACM, New York, NY, USA, 2011, pp. 416–419. doi:10.1145/2025113.2025179.

[6] P. Delgado-Pérez, I. Medina-Bulo, S. Segura, A. García-Domínguez, J. J. Domínguez-Jiménez, GiGAn: Evolutionary mutation testing for C++ object-oriented systems, in: Proceedings of the Symposium on Applied Computing, SAC '17, ACM, New York, NY, USA, 2017, pp. 1387–1392. doi:10.1145/3019612.3019828.

[7] P. Delgado-Pérez, I. Medina-Bulo, M. Núñez, Using evolutionary mutation testing to improve the quality of test suites, in: Proceedings of the IEEE Congress on Evolutionary Computation, CEC'17, 2017, pp. 596–603. doi:10.1109/CEC.2017.7969365.

[8] K. Sastry, D. E. Goldberg, G. Kendall, Genetic Algorithms, Springer US, Boston, MA, 2014, pp. 93–117. doi:10.1007/978-1-4614-6940-7_4.

[9] M. Dorigo, T. Stützle, Ant Colony Optimization, The MIT Press, 2004.

[10] M. Usaola, P. Mateo, Mutation testing cost reduction techniques: A survey, Software, IEEE 27 (3) (2010) 80–86. doi:10.1109/MS.2010.79.

[11] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE Transactions on Software Engineering 37 (5) (2011) 649–678.

[12] P. Delgado-Pérez, S. Segura, I. Medina-Bulo, Assessment of C++ object-oriented mutation operators: A selective mutation approach, Software Testing, Verification and Reliability 27 (4-5) (2017) e1630–n/a. doi:10.1002/stvr.1630.

[13] E. F. Barbosa, J. C. Maldonado, A. M. R. Vincenzi, Toward the determination of sufficient mutant operators for C, Software Testing, Verification and Reliability 11 (2) (2001) 113–136. doi:10.1002/stvr.226.

[14] R. P. Pargas, M. J. Harrold, R. R. Peck, Test-data generation using genetic algorithms, Software Testing, Verification and Reliability 9 (4) (1999) 263–282. doi:10.1002/(SICI)1099-1689(199912)9:4<263::AID-STVR190>3.0.CO;2-Y.

[15] A. E. Eiben, J. E. Smith, Introduction to Evolutionary Computing, 2nd Edition, Springer Publishing Company, Incorporated, 2015.

[16] J. J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, I. Medina-Bulo, Evolutionary mutation testing, Information and Software Technology 53 (10) (2011) 1108–1123. doi:10.1016/j.infsof.2011.03.008.

[17] X. Yao, M. Harman, Y. Jia, A study of equivalent and stubborn mutation operators using human analysis of equivalence, in: Proceedings of the 36th International Conference on Software Engineering, ICSE'14, ACM, New York, NY, USA, 2014, pp. 919–930. doi:10.1145/2568225.2568265.

[18] J. J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, I. Medina-Bulo, GAmera: an automatic mutant generation system for WS-BPEL compositions, in: R. Eshuis, P. Grefen, G. A. Papadopoulos (Eds.), Proceedings of the 7th IEEE European Conference on Web Services, IEEE Computer Society Press, Eindhoven, The Netherlands, 2009, pp. 97–106.

[19] D. E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning, 1st Edition, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[20] P. Delgado-Pérez, I. Medina-Bulo, F. Palomo-Lozano, A. García-Domínguez, J. J. Domínguez-Jiménez, Assessment of class mutation operators for C++ with the MuCPP mutation system, Information and Software Technology 81 (2017) 169–184. doi:10.1016/j.infsof.2016.07.002.

[21] A. Vargha, H. D. Delaney, A critique and improvement of the CL common language effect size statistics of McGraw and Wong, Journal of Educational and Behavioral Statistics 25 (2) (2000) 101–132. doi:10.3102/10769986025002101.

[22] L. Zhang, M. Gligoric, D. Marinov, S. Khurshid, Operator-based and random mutant selection: Better together, in: Proceedings of the IEEE/ACM 28th International Conference on Automated Software Engineering (ASE'13), 2013, pp. 92–102. doi:10.1109/ASE.2013.6693070.

[23] T. T. Chekam, M. Papadakis, Y. L. Traon, M. Harman, An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption, in: Proceedings of the IEEE/ACM 39th International Conference on Software Engineering (ICSE'17), 2017, pp. 597–608. doi:10.1109/ICSE.2017.61.

[24] P. R. Mateo, M. P. Usaola, Reducing mutation costs through uncovered mutants, Software Testing, Verification and Reliability 25 (5-7) (2015) 464–489. doi:10.1002/stvr.1534.

[25] Y.-S. Ma, Y. R. Kwon, S.-W. Kim, Statistical investigation on class mutation operators, ETRI Journal 31 (2) (2009) 140–150. doi:10.4218/etrij.09.0108.0356.

[26] M. Papadakis, Y. Jia, M. Harman, Y. Le Traon, Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique, in: Proceedings of the 37th International Conference on Software Engineering - Volume 1, ICSE'15, IEEE Press, Piscataway, NJ, USA, 2015, pp. 936–946. doi:10.1109/ICSE.2015.103.

[27] M. Papadakis, M. Delamaro, Y. L. Traon, Mitigating the effects of equivalent mutants with mutant classification strategies, Science of Computer Programming 95, Part 3 (2014) 298 – 319, special Section: ACM SAC-SVT 2013 + Bytecode 2013. doi:10.1016/j.scico.2014.05.012.

[28] T. A. Budd, Mutation analysis of program test data, Ph.D. thesis, Yale University (1980).

[29] S. Hussain, Mutation clustering, Master's thesis, King's College London (2008).

[30] D. Gong, G. Zhang, X. Yao, F. Meng, Mutant reduction based on dominance relation for weak mutation testing, Information and Software Technology 81 (C) (2017) 82–96. doi:10.1016/j.infsof.2016.05.001.

[31] Y. Jia, M. Harman, Higher order mutation testing, Information and Software Technology 51 (10) (2009) 1379–1393. doi:10.1016/j.infsof.2009.04.016.

[32] K. Adamopoulos, M. Harman, R. M. Hierons, How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution, in: Proceedings of the Genetic and Evolutionary Computation Conference, GECCO'04, 2004, pp. 1338–1349. doi:10.1007/978-3-540-24855-2_155.

[33] A. A. L. de Oliveira, C. G. Camilo-Junior, A. M. R. Vincenzi, A coevolutionary algorithm to automatic test case selection and mutant in mutation testing, in: Proceedings of the IEEE Congress on Evolutionary Computation, CEC'13, 2013, pp. 829–836. doi:10.1109/CEC.2013.6557654.

[34] B. Schwarz, D. Schuler, A. Zeller, Breeding high-impact mutations, in: Proceedings of the 4th IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW'11, 2011, pp. 382–387. doi:10.1109/ICSTW.2011.56.

[35] Y. Jia, M. Harman, Constructing subtle faults using higher order mutation testing, in: Eighth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM'08, 2008, pp. 249–258. doi:10.1109/SCAM.2008.36.

[36] E. Omar, S. Ghosh, D. Whitley, Homaj: A tool for higher order mutation testing in AspectJ and Java, in: IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops, ICSTW'14, 2014, pp. 165–170. doi:10.1109/ICSTW.2014.19.

[37] M. Harman, Y. Jia, P. Reales Mateo, M. Polo, Angels and monsters: An empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation, in: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ASE '14, ACM, New York, NY, USA, 2014, pp. 397–408. doi:10.1145/2642937.2643008.

[38] J. A. P. Lima, G. Guizzo, S. R. Vergilio, A. P. C. Silva, H. L. J. Filho, H. V. Ehrenfried, Evaluating different strategies for reduction of mutation testing costs, in: Proceedings of the 1st Brazilian Symposium on Systematic and Automated Software Testing, SAST, ACM, New York, NY, USA, 2016, pp. 4:1–4:10. doi:10.1145/2993288.2993292.

[39] A. S. Banzi, T. Nobre, G. B. Pinheiro, J. C. G. Árias, A. Pozo, S. R. Vergilio, Selecting mutation operators with a multiobjective approach, Expert Systems with Applications 39 (15) (2012) 12131–12142. doi:10.1016/j.eswa.2012.04.041.

[40] J. A. P. Lima, S. R. Vergilio, A multi-objective optimization approach for selection of second order mutant generation strategies, in: Proceedings of the 2nd Brazilian Symposium on Systematic and Automated Software Testing, SAST, ACM, New York, NY, USA, 2017, pp. 6:1–6:10. doi:10.1145/3128473.3128479.

[41] W. B. Langdon, M. Harman, Y. Jia, Efficient multi-objective higher order mutation testing with genetic programming, Journal of Systems and Software 83 (12) (2010) 2416 – 2430, TAIC PART 2009 - Testing: Academic and Industrial Conference - Practice And Research Techniques. doi:10.1016/j.jss.2010.07.027.

[42] X. Yao, D. Gong, G. Zhang, Constrained multi-objective test data generation based on set evolution, IET Software 9 (4) (2015) 103–108. doi:10.1049/iet-sen.2014.0058.

[43] M. B. Bashir, A. Nadeem, A fitness function for evolutionary mutation testing of object-oriented programs, in: IEEE 9th International Conference on Emerging Technologies ICET'13, 2013, pp. 1–6. doi:10.1109/ICET.2013.6743531.

[44] S. Shamshiri, J. M. Rojas, G. Fraser, P. McMinn, Random or genetic algorithm search for object-oriented test suite generation?, in: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, GECCO '15, ACM, New York, NY, USA, 2015, pp. 1367–1374. doi:10.1145/2739480.2754696.