Using Evolutionary Mutation Testing to Improve the Quality of Test Suites

Pedro Delgado-Pérez*, Inmaculada Medina-Bulo*, Manuel Núñez†
*Departmento de Ingeniería Informática, Escuela de Ingeniería
Universidad de Cádiz, Spain
Email: pedro.delgado@uca.es, inmaculada.medina@uca.es
† Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Spain
Email: manuelnu@ucm.es

***Abstract*—Mutation testing is a method used to assess and improve the fault detection capability of a test suite by creating faulty versions, called *mutants*, of the system under test. Evolutionary Mutation Testing (EMT), like selective mutation or mutant sampling, was proposed to reduce the computational cost, which is a major concern when applying mutation testing. This technique implements an evolutionary algorithm to produce a reduced subset of mutants but with a high proportion of mutants that can help the tester derive new test cases (strong mutants). In this paper, we go a step further in estimating the ability of this technique to induce the generation of test cases. Instead of measuring the percentage of strong mutants within the subset of generated mutants, we compute how much the test suite is actually improved thanks to those mutants. In our experiments, we have compared the extent to which EMT and the random selection of mutants help to find missing test cases in C++ object-oriented systems. We can conclude from our results that the percentage of mutants generated with EMT is lower than with the random strategy to obtain a test suite of the same size and that the technique scales better for complex programs.**

***Index Terms*—Mutation testing, evolutionary computation, object-oriented programming, C++.**

I. Introduction

Mutation testing is a suitable technique to determine the quality of test suites for a particular program [1], [2], [3]. It is based on the generation of *mutants*, that is, versions of the original program with an intentionally introduced fault in the source code. These faults are actually subtle modifications in the syntax of the system under test (SUT) and they are inserted using predefined *mutation operators*. The underlying idea in mutation testing is that the injected mutations represent plausible coding errors when developing a system with a certain programming language. Then, the test suite designed for the SUT is evaluated by measuring its ability to detect a change in the behaviour of the mutants when they are executed on the same test suite as the original program.

Mutation testing is effective at finding test deficiencies, but it is also a costly testing technique because of the large number of mutants that can be generated. Several methods have been proposed to reduce the computational cost of mutation testing, mainly discarding a subset of the mutants [4].

This paper aims at analysing Evolutionary Mutation Testing (EMT) [5] as a tool to reduce the number of mutants generated and, as a result, improve the quality of test suites. This technique uses an evolutionary algorithm to favour the generation of *strong mutants*, which can guide the tester on the creation of new test cases. *Potentially equivalent mutants*, which are not detected by the initial test suite, and *difficult to kill mutants*, which are detected by one test case which only detects this mutant and no other, are both considered as strong mutants.

The experiments conducted regarding EMT so far [5], [6] have evaluated how useful EMT is in finding strong mutants and have shown that EMT outperforms random selection. However, EMT aims at improving the test suite with the set of generated mutants ultimately. In this line, some strong mutants may be later found to be *equivalent mutants* (i.e., mutants with the same functionality as the original SUT despite their mutation). Being aware of this fact, finding a subset of strong mutants does not ensure that the test suite is proportionally augmented with new test cases. As a result, previous work was not able to clarify the extent to which using EMT leads to the generation of missing or incomplete test cases when compared to a random strategy.

In this paper we go beyond previous experiments by introducing a new methodology to assess test suites improvement when applying EMT. The experiments underlying our methodology simulate a real process in which the generated mutants are reviewed and new test cases are added to detect surviving mutants (i.e., the mutations modelled by these mutants are not identified by the current test suite). This simulation is possible thanks to a previous execution of all the mutants and a process of test suite improvement to obtain an *adequate test suite*, that is, a test suite detecting all non-equivalent mutants. In order to evaluate the usefulness of our methodology, in this paper we evaluate EMT with respect to a set of class mutation operators [7] (related to the object-oriented paradigm) and to real open-source C++ programs, together with the test suite distributed with each of them. The results show that mutation testing can benefit from EMT because it is possible to generate a reduced subset of mutants, leading to a further test suite refinement, in comparison with random selection of the same number of mutants. This is especially important for those

SUTs that produce a large set of mutants.

Sections II and III describe mutation testing and EMT, respectively, with a focus on C++ object-oriented systems. Section IV explains the empirical evaluation in this paper to asses the test suite improvement when applying EMT, whereas Section V shows the results of the experimental procedure and discusses them. Related work is presented in Section VI. Finally, Section VII presents our conclusions and future research lines.

II. Mutation Testing

Mutation testing is a fault-based testing technique in which a set of mutation operators generates mutated versions of the subject program. Mutation operators are mainly derived from the analysis of the most common mistakes in the development of applications in a certain language. For instance, we can consider that the programmers are likely to make mistakes with arithmetic operators when coding a certain program *P*. Then, a mutation operator replacing arithmetic operators can be applied. Let us consider that after analysing a fragment of code in this program, four different mutants (*M1*, *M2*, *M3* and *M4*) can be produced. The next table shows the original program and the mutants.

(P)	*int add (int a, int b)*	*{return a + b;}*
(M1)	*int add (int a, int b)*	*{return a - b;}*
(M2)	*int add (int a, int b)*	*{return a * b;}*
(M3)	*int add (int a, int b)*	*{return a / b;}*
(M4)	*int add (int a, int b)*	*{return a % b;}*

These four mutants are executed on the same test suite as *P*. The test suite should be able to detect these mutations in the code when comparing the output of the original and the mutated programs. That is, for certain inputs the mutant and the original program should produce different outputs. In that case, the mutants are *killed*. On the contrary, a mutant that remains *alive* reveals a potential test deficiency.[1] That means that, in the instance that the code contained that fault, the current test suite would not be able to identify it. For example, a test case exercising the function "add" with the values $a = 2$ and $b = 2$ kills the mutants *M1*, *M3* and *M4*. However, this test case is not able to kill *M2* because this function returns the same value (4) when both *P* and *M2* are executed. Therefore, this technique is effective in assessing and improving test suites. However, because of the number of mutants that can be generated and the time that their execution can take, it is considered to be computationally expensive.

Several faults, like the example above, are common in many general purpose languages, but the set of mutation operators should be specifically designed according to the features of each language. Thus, there are many different approaches to define operators for a great range of languages [3]. The similarity of the operators for several languages has also been explored [8]. The language C++, used for our experiments in this paper, has not been tackled in mutation testing until recently [7]. According to the different levels at which mutation testing can be applied [3], the mutation operators defined for C++ [7] can be categorised as *class-level operators* because they deal with object-oriented features. This set of mutation operators has been implemented in the mutation system *MuCPP* [9].

[1]This is not the case if the mutant is *equivalent* to the original program, that is, the mutation does not produce any semantic change in the program.

III. Evolutionary Mutation Testing with GiGAn

In this section we present how EMT is implemented in the GiGAn system. First, we present a general overview of EMT and then we explain the main characteristics of GiGAn.

A. Evolutionary Mutation Testing

EMT aims at generating a reduced set of mutants by means of an evolutionary algorithm. The algorithm is guided towards the generation of those mutants that may provide the tester with the possibility of adding new test cases to the test suite. These special mutants are called *strong mutants*. Two types of mutants are considered as strong mutants: *potentially equivalent*, which are not detected by the test suite under evaluation, and *difficult to kill* mutants, detected by one test case only killing that mutant. In the case of potentially equivalent mutants, either they can turn out to be equivalent once they are reviewed or they can induce the generation of new test cases. This happens when a mutation is not exercised by the current test cases or they are not able to uncover the mutation. Checking equivalence is undecidable but recent work provides a good methodology to automatically detect a subset of equivalent mutants [10].

Any EMT approach needs to apply the following steps:

1) Produce (randomly) a subset of mutants in a first generation.
2) Execute the subset of mutants against the test cases in the supposedly non-adequate test suite.
3) Compute the fitness of each of the mutants in the generation.
4) Apply the evolutionary algorithm to produce a new generation of mutants based on the calculated mutants fitness. A percentage of the mutants in each generation is generated ramdomly.
5) Stop the algorithm if the stopping condition is met. Otherwise, repeat the process from step 2.

An important concept in any EMT is the notion of *fitness function*. This function measures the quality of a solution and therefore is devised for each particular problem.

$$\mathrm{F}(I) = M \times T - \sum_{j=1}^{T} \left(m_{Ij} \times \sum_{i=1}^{M} m_{ij} \right) \qquad (1)$$

Where:

- M is the number of mutants and T is the number of test cases.
- m_{ij} is 1 when mutant i is detected by test case j, and 0 otherwise.

The fitness function of a mutant I in EMT (see Equation 1) assigns the best value to strong mutants (potentially equivalent mutants: $M \times T$; difficult to kill mutants: $M \times T - 1$) so that they are selected for reproduction with high probability. Roughly speaking, the fitness of the rest of the mutants (weak mutants) decreases as the number of test cases detecting the mutant increases and the number of mutants detected by those test cases increases.

In order to calculate the fitness function, each mutant needs to be run against each of the test cases in the test suite. This process produces a result which is stored in an *execution matrix*. The rows in the execution matrix represent the mutants while the columns represent the test cases. The format of a general execution matrix is shown in Figure 1. The execution matrix allows the tester to quickly compute the number of test cases killing a certain mutant and the number of mutants being killed by those test cases. This execution matrix is, therefore, useful for the calculation of the fitness of the mutants. We will also generate execution matrices later on for the experiments.

$$\begin{array}{c c} & \begin{array}{cccc} \text{test}_1 & \text{test}_2 & \dots & \text{test}_m \end{array} \\ \begin{array}{c} \text{mutant}_1 \\ \text{mutant}_2 \\ \dots \\ \text{mutant}_n \end{array} & \begin{pmatrix} v_{1,1} & v_{1,2} & \dots & v_{1,m} \\ v_{2,1} & v_{2,2} & \dots & v_{2,m} \\ \dots & \dots & \dots & \dots \\ v_{n,1} & v_{n,2} & \dots & v_{n,m} \end{pmatrix} \end{array}$$

where $v_{i,j}$ is the result (0: alive, 1: killed, 2: invalid) of the application of test j to mutant i.

Figure 1. Matrix execution format

A mutant is *invalid* when it cannot be executed because it does not comply with the syntax of the language. Invalid mutants neither are assigned a fitness nor affect the fitness computation of the rest of valid mutants, as they are removed from the execution matrix. We should note that the fitness function:

- Penalises groups of mutants killed by the same test cases. Even if few mutants from one of those groups are produced in a generation and they are selected to breed a new generation, it is likely that several mutants from that group are created and the fitness of the mutants in that group drops.
- Similarly, when those mutants are derived from the same mutation operator, the algorithm will penalise this operator focusing on others in successive generations.

The GiGAn system

In this paper we use *GiGAn*[6], a system implementing EMT for object-oriented programs written in C++. *GiGAn* connects the genetic algorithm implemented in *GAmera* [11] with the mutation system *MuCPP* [9], where a set of class mutation operators for C++ programs is implemented. The process orchestrated by *GiGAn* is as follows:

1) *MuCPP* analyses the C++ source code of the project. This generates a report with a list of the mutants that each mutation operator can produce in the code. The genetic algorithm uses the report to know which mutants (individuals) can be generated.
2) The genetic algorithm produces a generation of individuals selecting as many mutants as previously set in the configuration.
3) A converter transforms the individuals into usable mutant identifiers for *MuCPP*.
4) *MuCPP* executes the mutants against the test suite, resulting in an execution matrix that is used by the genetic algorithm to compute the fitness of each individual.
5) Steps 2 to 5 are repeated until the stopping condition is satisfied, for instance, when reaching a percentage of the full set of mutants or a number of generations.

The encoding scheme of an individual is a pair ($operator$, $location$), where the field $operator$ is an integer indicating the mutation operator that generates the mutant, and the field $location$ is an integer representing the order in the code of the mutants generated by an operator. We should note that the genetic algorithm automated in *GAmera* also contemplates a third field called $attribute$ (variant inserted into a location), but that field has been disabled in *GiGAn* because all class mutation operators in *MuCPP* present $attribute = 1$. Regarding Step 3, the conversion is required because the genetic algorithm uses a normalised version of the field $location$ to avoid selecting invalid representations of mutants (the mutant that results after applying a reproductive operator can always be generated), and also because *MuCPP* uses a different format for mutant identifiers.

The genetic algorithm used by *GiGAn* maximises the sum of the fitness of the individuals in each generation to evolve toward better solutions (finding strong mutants). The algorithm assumes that nearby individuals are likely to be similar to those that induced their generation. As usual, our genetic algorithm depends on two kind of operators: selection and reproductive operators. Our genetic algorithm applies the *roulette wheel method* [12] as selection operator. The quick convergence of this selection method of mutants is convenient in the case of EMT because we are interested in obtaining the set of strong mutants with a reduced set of mutants. As for reproductive operators, the genetic algorithm can apply *mutation operators*[2] and *crossover operators*:

- *Mutation operators*. They modify the information of one of the selected individuals to generate a new individual. As such, one of the two fields to identify an individual ($operator$ or $location$) is mutated, producing either the individual $(operator', location)$ or $(operator, location')$.
- *Crossover operators*. They combine the information of two individuals (parents) to generate two new individuals (children), which inherit information from both parents. To that end, a crossover point related to the encoding scheme is selected. As it is shown in Figure 3, the $operator$ of both parents is swapped.

[2]Do not confuse these mutation operators with the mutation operators applied in mutation testing.

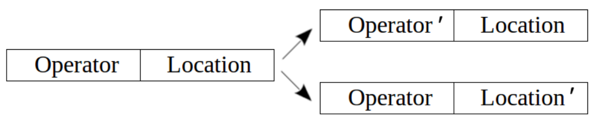

Figure 2. Mutation operators

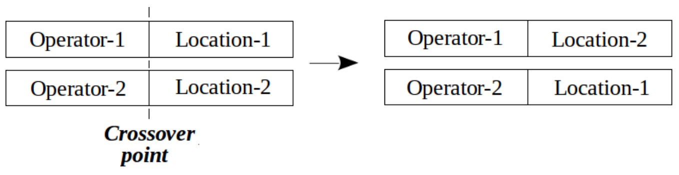

Figure 3. Mutant crossover

Finally, we should note that the algorithm does not only rely on the fitness function of the mutants in a generation to detect groups of similar individuals but also on all the mutants generated so far (the mutants in a generation learn from the previous generations). These mutants are saved in a second population, which is the output of the algorithm at the end of its execution.

IV. Methodology

The main goal of our research consists in producing a *good* subset of mutants and then use the information of their execution against a test suite to design new test cases. In this section we describe the two sequential phases conforming our methodology. In the next section we show how these two phases are applied to four case studies.

First phase

The purpose of this phase is to obtain adequate test suites for the SUTs. These adequate test suites will be used for the simulation of the test suite improvement in the next phase. The process is as follows:

1) Execute all the mutants against the current non-adequate test suite (T_{NA}).
2) Inspect the surviving mutants to distinguish between equivalent and non-equivalent mutants.
3) Design new test cases to kill all the surviving non-equivalent mutants, thereby reaching an adequate test suite (T_A).
4) Execute all the mutants against all the test cases in T_A, obtaining the final execution matrix associated with T_A (EM).
5) Minimise T_A using the information in EM. The result is a *minimal and adequate test suite* (T_{MA}).

A test suite is labelled as minimal when it contains the minimum number of test cases needed to kill the set of non-equivalent mutants. In this study, using minimal test suites is fundamental to avoid that redundant test cases skew results (a test case is redundant with respect to a set of mutants if the same mutants are still killed when that test case is removed).

At the end of this phase, we have an adequate test suite for the set of mutants and information in an execution matrix about which mutants can induce the generation of which test cases in that test suite.

Second phase

In this phase we use the information in EM to know how many mutants need to be generated by the genetic algorithm in order to reach the stopping condition: the algorithm stops when reaching a given percentage (P) of the number of test cases in the minimal and adequate test suite ($|T_{MA}|$) with the subset of mutants generated. For this simulation, we have the following components:

- The current test suite (T_{NA}) for the execution of EMT.
- The future test suite (T_A) after being refined.
- The result of the execution of the mutants against the adequate test suite in the future producing a new execution matrix (EM).

The idea is to resemble a real process of test suite improvement; we stop the algorithm when it has generated enough mutants to improve the test suite in a certain percentage with respect to T_A. By extracting from EM the information of the execution of the mutants generated by the algorithm, we can estimate how many test cases would be induced by those mutants. This estimation is based on the comparison between the size of the current test suite and the size of the improved test suite (through the mutants selected) when both test suites are minimised.

We run EMT and the genetic algorithm follows the following steps:

1) Let $i = 1$.
2) Select the set of mutants, as explained in Section III, to be added in generation i (M_i).
3) Select the rows of EM corresponding to the mutants in the set of generations $M_0, \ldots, M_i$. We produce EM_i, a new execution matrix associated with generation i and using the information in EM of the mutants generated so far by the algorithm.
4) Minimise T_A using the information in EM_i. This step leads to a minimal and adequate test suite for the mutants selected by the algorithm so far (T_{MA_i}).
5) If the stopping condition $|T_{MA_i}| \geq |T_{MA}| \times P$ is satisfied then we stop; otherwise, we increase i and go to Step 2.

In the experiments reported in the next section we used two stopping conditions (that is, two values for P): reaching, respectively, 75% and 90% of the minimal and adequate test suite (T_{MA}). We executed the second phase 30 times with different seeds. The same process is followed for the random algorithm (*Random* from now on), where only one mutant is randomly selected in step 2.

V. Experiments, Results and Discussion

In this section we present how our methodology is put into practice. We apply it to generate test suites for four different

Table I
METRICS ABOUT THE SUTs USED IN THE EXPERIMENTS

		TCL	DPH	TXM	DOM
Mutants	Total	137	219	614	1,146
	Valid	135	208	433	681
	Strong	45	103	159	348
	% Strong mutants	33.3%	49.5%	36.7%	51.1%
Test suite	$\|Original\ T\|$	17	61	57	46
	$\|Adequate\ T\|$	24(3)	70(5)	62(3)	56(4)
	$\|Minimal\ T\|$	15	22	15	25

real C++ programs. In addition, we show the goodness of the test suites by comparing them with the ones generated by a random algorithm to select mutants. We will see that we are able to augment the test suite while generating fewer mutants.

A. First phase: test suite improvement

The first part of this experiment requires that adequate test suites are obtained for each of the SUTs used in this study. We make use of the same case studies as in the previous experiments conducted to find different percentages of strong mutants [6]. These programs are briefly described below:

- *Matrix TCL Pro* (*TCL*) [13]: a library to perform matrix algebra calculations.
- *Dolphin* (*DPH*) [14]: the default navigational file manager in KDE desktop applications.
- *Tinyxml2* (*TXM*) [15]: a lightweight and efficient XML parser that can be integrated into C++ applications.
- *QtDom* (*DOM*) [16]: a module of the known Qt framework that provides a C++ implementation of the DOM standard.

Table I shows different metrics related to these programs. These quantities are divided in two main groups:

- *Distribution of mutants*. We provide the amount of total, valid and strong mutants (the percentage of strong mutants with respect to the set of valid mutants is also shown).
- *Size of the test suites*. We provide the number of test cases in the *original* test suite (the suite distributed with the program), in the *adequate* test suite (the one produced after adding new test cases to kill surviving mutants) and in the *minimal* test suite (the one derived from the application of an exact algorithm for test suite minimisation using the execution matrix EM). In the information corresponding to the adequate test suite, we have also shown, between parentheses, the number of test cases additionally modified. We modify a test case, instead of inserting new ones, when the assertion needed to kill a mutant is closely related to the logic of that test case.

Let us remark that the percentage of strong mutants created by *MuCPP* is not the same for different applications (it also depends on the test suite distributed with each application).

B. Second phase: simulation results

Table II shows the results of the second phase. Different statistics have been computed (mean, median, minimum, maximum and standard deviation). The values shown in the table represent the percentage of mutants that have been generated to reach the stopping conditions both by EMT and *Random*. Thus, the lower the values in this table the better the result.

Before interpreting these results, we should note that the evaluation is impacted by the test suite itself. For instance, not in all the SUTs the current non-adequate suite (T_{NA}) is at the same distance of the adequate test suite (T_A): the difference between the size of the minimised current test suite and the size of the minimal and adequate test suite is not the same for all the programs. The size of the test suite may also affect the search and the results of the genetic algorithm. Moreover, each test suite presents a different nature and is comprised of *general* and *specific* test cases. We consider a test case to be general when several invocations are needed before testing a particular functionality and some other test cases cover a subset of related functionalities instead of a single functionality (as in a specific test case). This fact has an impact on the power of the test suite in killing mutants, that is, whether mutants are killed by few or many of its test cases overall. As a result, we can observe that, on average, EMT needs to generate 49.75% of mutants to reach 75% of T_{MA} in *DPH*, whereas it only needs to generate 13.33% in *DOM* for the same end. As such, we should not directly compare the results among applications.

The results for EMT are better than the results for *Random* in *DPH*, *TXM* and *DOM*, but worse in *TCL*. We can infer that the results scale with the size of the program, given that the best result is obtained in *DOM*, followed by *TXM* and *DPH* (in descending order of the number of mutants). If we focus on the 75% stopping condition, the difference between EMT and *Random* increases from about 5% in *DPH* to 10% in *DOM*. We can also note that the outcome is better for the most demanding condition (90%). Again, the difference between EMT and *Random* increases from about 5% in *DPH* to 28% in *DOM*. When comparing the results for both stopping conditions, we can highlight the gap between EMT and *Random* in the case of *TXM* and *DOM*:

- P = 75%: approximately 6% and 10% respectively.
- P = 90%: approximately 15% and 28% respectively.

Table II
PERCENTAGE OF MUTANTS GENERATED WITH THE EVOLUTIONARY AND THE RANDOM STRATEGY TO REACH THE STOPPING CONDITIONS (75% AND 90% OF THE MINIMAL AND ADEQUATE TEST SUITE) IN THE SUTS

P		***75%***		***90%***	
Program	Statistic	*EMT*	*Random*	*EMT*	*Random*
TCL	Mean	37.24	32.45	49.24	47.85
	Med.	38.32	33.57	50.36	50.36
	Min.	18.97	14.59	25.54	25.54
	Max.	54.74	52.55	75.91	68.61
	SD	10.77	9.09	13.41	12.78
DPH	Mean	49.75	54.10	66.33	71.08
	Med.	48.63	52.05	65.29	69.63
	Min.	36.52	30.13	52.51	40.63
	Max.	74.42	84.01	84.93	88.58
	SD	8.51	9.95	8.61	10.45
TXM	Mean	19.26	25.75	31.93	46.79
	Med.	18.48	24.34	32.25	43.24
	Min.	10.58	11.88	20.52	24.75
	Max.	27.36	53.09	46.09	86.80
	SD	4.38	8.98	7.13	15.21
DOM	Mean	13.33	23.74	21.41	49.04
	Med.	13.00	21.90	21.16	46.68
	Min.	7.85	11.16	11.95	26.96
	Max.	23.29	49.04	35.86	81.15
	SD	3.35	7.99	5.00	12.85

It is also interesting to observe the standard deviation, which is lower in EMT than in *Random* for *DPH*, *TXM* and *DOM* and both stopping conditions (it is especially low for EMT in comparison with *Random* in *TXM* and *DOM*). However, we should note that the difference between both techniques may be impacted by the number of invalid mutants, which is higher in *TXM* and *DOM* than in *TCL* and *DPH*. Still, this fact also means that EMT has the ability to avoid the generation of invalid mutants, especially when they are generated by a subset of the operators (as it was commented in Section III, the algorithm tends to focus on operators providing a high fitness).

Regarding *TCL*, where the performance of *Random* was better, we suspect that these results are due to the test suite originally associated with this application. The test cases in this test suite can be labelled as general and, as mentioned before, these test cases do not usually lead to mutants killed by few test cases. On the contrary, given that these test cases cover a great part of the code, they tend to kill several mutants. Thus, some of the new or modified test cases (those that were manually designed) are likely to appear in this experiment without generating the mutants that led to the design of those test cases. In summary, it is possible to reach 75% and 90% of the minimal and adequate test suite with different combinations of mutants. This fact can be disadvantageous for EMT because this strategy is guided by the fitness function to find a specific subset of mutants, which also includes equivalent mutants that do not contribute to the refinement of the test suite.

In order to confirm this assumption, we repeated the experiments to construct the whole minimal and adequate test suite using as stopping condition P = 100%. We expect that EMT will benefit from the fitness function to find the most specific test cases quicker than the random strategy. The average results for all the applications are shown in Table III, which confirms that EMT is more effective than *Random* in finding the whole adequate and minimal test suite for *TCL*. Again, this situation shows that this evaluation depends on the test suite. Finally, we should remark that the results are again more positive for larger programs in terms of mutants generated, where the difference between EMT and *Random* is around 26% and 45% in *TXM* and *DOM* respectively.

Table III
AVERAGE PERCENTAGE OF MUTANTS GENERATED WITH THE EVOLUTIONARY AND THE RANDOM STRATEGY TO FIND THE WHOLE MINIMAL AND ADEQUATE TEST SUITE IN THE SUTS

Program	*EMT*	*Random*
TCL	75.79	80.05
DPH	88.24	91.63
TXM	49.30	75.92
DOM	34.69	80.51

C. Threats to Validity

These experiments have been conducted on C++ programs using mutation operators at the class level. New experiments should be carried out to ensure that the same results hold in other contexts, such as using a different programming language or a different set of mutation operators. Also, the genetic algorithm in these experiments was configured with the optimal combination of values for the parameters found by Domínguez-Jiménez et al. [5], but the results may vary with other configurations.

The modified and new test cases have been designed with the utmost care. Nevertheless, this is a challenging and error-prone task as we are dealing with third-party libraries and test suites developed by different testers. Therefore, the improvement of this test suite should be considered as an estimation. Being the identification of equivalent mutants a manual task (as we have already said, this is an undecidable problem), the results may be inaccurate. As aforementioned, the varying nature of the test suite accompanying the analysed subjects also impacts this evaluation and supposes a threat to the validity of the results. We should also remark that, in our experiments, selecting a mutant can change the minimal and adequate test suite in a generation (T_{MA_i}). However, that does not mean that the selected mutant has the potential to help the tester to create missing test cases, especially when the test suite is comprised of general test cases.

VI. Related Work

Many studies on mutation testing have sought to reduce its computational cost by selecting a subset of mutants, such as *mutant sampling* [17], *selective mutation* [18] or *higher order mutation* (HOM) [19]. Search-based techniques have also been used in software testing [20] in order to reduce the cost, especially using genetic algorithms to this purpose. Nevertheless, most of the studies applying genetic algorithms in mutation testing were limited to test case generation [21] and only a few, as raised by Domínguez-Jiménez et al. [5], to mutant generation.

Mutation testing has been applied in conjunction with evolutionary algorithms to generate test cases also for object-oriented software [22], [23]. Bashir and Nadeem [23] used the term Evolutionary Mutation Testing but in order to propose a fitness function to find effective test cases for object-oriented programs. Adamopoulos et al. [24] used a genetic algorithm for the co-evolution of mutant and test suite population, where difficult to kill mutants are favoured and equivalent mutants are penalised. Banzi et al. [25] also applied a genetic algorithm but at the mutation operator level for selective mutation. They used a multi-objective approach to select mutation operators that maximise the adequacy of the test suite and minimise the number of mutants generated.

However, there is an increasing body of research in the last years applying genetic programming to select a reduced set of mutants. Silva et al. [26] collected the studies applying search-based techniques in the context of mutation testing for mutant generation. Oliveira et al. [27] studied the evolution in parallel of the population of mutants and test cases, as done by Adamopoulos et al. [24]. However, they used a new representation with new genetic operators: Effective Son crossover and Muta Genes mutation (instead of the crossover and mutation operators used by Adamopoulos et al.). Schwarz et al. [28] used a genetic algorithm to find mutations not detected by the test suite, which have a high impact and are also spread throughout the tested code.

The studies in the context of mutant generation have mainly focused on genetic programming to generate interesting HOMs [19], [29]. Jia and Harman [19] defined the concept of *subsuming HOM* as a HOM which is hard to kill when compared to the difficulty of killing the First Order Mutants (FOMs) from which it is constructed. The authors applied several search-based techniques in order to find subsuming HOMs, concluding that the genetic algorithm yielded the best results of all them. However, Omar et al. [29], who have also explored the performance of search-based techniques (including genetic programming), found that guided local search obtained the best results in general when finding subtle HOMs for Java and AspectJ programs.

As for EMT, Domínguez-Jiménez et al. [5] analysed its performance when finding strong mutants with three WS-BPEL compositions and Delgado-Pérez et al. [6] replicated their experience with the same four C++ programs used in the experiments in this paper. In the latter study, EMT was better than *Random* in all the SUTs, but the differences in *TXM* and *DOM* were not so relevant as reported here. In fact, the shortest difference of all the SUTs in that study was obtained in *DOM*, so the results did not scale with the size of the program as they do in this paper.

VII. Conclusions and Future Work

Evaluating the performance of cost reduction methods for mutation testing is a necessary task for the success of this testing technique. Evolutionary Mutation Testing, a search-based technique to select a reduced set of mutants with the focus on test suites improvement, had reported good results regarding its capability to find those mutants with the potential to induce the creation of new test cases (strong mutants). However, finding strong mutants is not the ultimate goal in practice, but the augmentation of the test suite. The experiments in this paper do give evidence that using EMT can lead to a further refinement of the test suite when generating only a subset of mutants. Overall, the application of EMT together with class mutation operators for C++ produced a subset of selected mutants that were able to generate/modify a greater number of test cases when compared to the same percentage of mutants randomly selected. In addition, EMT scaled better for large programs (those that derived a higher number of mutants).

Further research on this cost reduction technique is still required, as well as applying EMT to other domains and implementing new refinements in the mutant selection method for a better performance of the genetic algorithm (e.g., integrating techniques to reduce the selection of equivalent mutants). In future experiments, performing statistical comparisons between the genetic and the random algorithm and studying how much of the difference between these algorithms is due to the different nature of the test suites or to the existence of invalid mutants would also be interesting. Finally, a study of the best-valued operators following the quality metric by Estero-Botaro et al. [30] for test suite refinement could help us to further reduce the cost by applying selective mutation.

VIII. Acknowledgments

The research reported in this paper was partially supported by the research scholarship PU-EPIF-FPI-PPI-BC 2012-037 of the University of Cádiz and the projects DArDOS (TIN2015-65845-C3-R (MINECO/FEDER)), SICOMORo-CM (S2013/ICE-3006) and the Excellence Network SEBASEnet (TIN2015-71841-REDT (MINECO)). We would like to thank the anonymous reviewers for their helpful comments.

References

[1] R. M. Hierons, M. G. Merayo, and M. Núñez, *Encyclopedia of Software Engineering*. Taylor & Francis, 2010, ch. Mutation Testing, pp. 594–602. [Online]. Available: http://dx.doi.org/doi:10.1081/E-ESE-120044190

[2] P. Delgado-Pérez, I. Medina-Bulo, and J. Domínguez-Jiménez, *Encyclopedia of Information Science and Technology, Third Edition*. IGI Global, 2014, ch. Mutation Testing, pp. 7212–7221. [Online]. Available: http://dx.doi.org/doi:10.4018/978-1-4666-5888-2.ch710

[3] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, Oct. 2011. [Online]. Available: http://dx.doi.org/10.1109/TSE.2010.62

[4] M. Polo-Usaola and P. Reales-Mateo, "Mutation testing cost reduction techniques: A survey," *IEEE Software*, vol. 27, no. 3, pp. 80–86, 2010. [Online]. Available: http://dx.doi.org/10.1109/MS.2010.79

[5] J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, and I. Medina-Bulo, "Evolutionary mutation testing," *Information and Software Technology*, vol. 53, no. 10, pp. 1108–1123, Oct. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2011.03.008

[6] P. Delgado-Pérez, I. Medina-Bulo, S. Segura, J. J. Domínguez-Jiménez, and A. García-Domínguez, "GiGAn: Evolutionary mutation testing for C++ object-oriented systems," in *The 32nd ACM Symposium On Applied Computing (SAC 2017)*, 2017.

[7] P. Delgado-Pérez, I. Medina-Bulo, J. Domínguez-Jiménez, A. García-Domínguez, and F. Palomo-Lozano, "Class mutation operators for C++ object-oriented systems," *Annals of Telecommunications*, vol. 70, no. 3, pp. 137–148, 2015. [Online]. Available: http://dx.doi.org/10.1007/s12243-014-0445-4

[8] J. Boubeta-Puig, A. García-Domínguez, and I. Medina-Bulo, "Analogies and differences between mutation operators for WS-BPEL 2.0 and other languages," in *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, IEEE. Berlin, Germany: IEEE, 2011, p. 398–407. [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2011.52

[9] P. Delgado-Pérez, I. Medina-Bulo, F. Palomo-Lozano, A. García-Domínguez, and J. J. Domínguez-Jiménez, "Assessment of class mutation operators for C++ with the MuCPP mutation system," *Information and Software Technology*, vol. 81, pp. 169–184, 2017. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2016.07.002

[10] M. Papadakis, Y. Jia, M. Harman, and Y. L. Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *37th IEEE/ACM Int. Conf. on Software Engineering, ICSE'15*. Florence, Italy: IEEE Computer Society, May 2015, pp. 936–946. [Online]. Available: http://dx.doi.org/doi:10.1109/ICSE.2015.103

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

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

[13] "Matrix TCL Pro, version 2.2," http://www.techsoftpl.com/matrix/download.php, 2016, [Online; accessed 12-Dec-2016].

[14] "Dolphin," https://www.kde.org/applications/system/dolphin, 2016, [Online; accessed 12-Dec-2016].

[15] "Tinyxml2," https://github.com/leethomason/tinyxml2, 2016, [Online; accessed 12-Dec-2016].

[16] "QtDOM," https://github.com/qtproject/qtbase/tree/dev/src/xml/dom, 2016, [Online; accessed 12-Dec-2016].

[17] T. A. Budd, "Mutation analysis of program test data," Ph.D. dissertation, Yale University, 1980.

[18] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi, "Toward the determination of sufficient mutant operators for C," *Software Testing, Verification and Reliability*, vol. 11, no. 2, pp. 113–136, 2001. [Online]. Available: http://dx.doi.org/10.1002/stvr.226

[19] Y. Jia and M. Harman, "Constructing subtle faults using higher order mutation testing," in *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation, 2008*, Sept 2008, pp. 249–258. [Online]. Available: http://dx.doi.org/10.1109/SCAM.2008.36

[20] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios, "Application of genetic algorithms to software testing," in *Proceedings of the 5th International Conference on Software Engineering and Applications*, 1992, pp. 625–636.

[21] R. P. Pargas, M. J. Harrold, and R. R. Peck, "Test-data generation using genetic algorithms," *Software Testing, Verification and Reliability*, vol. 9, no. 4, pp. 263–282, 1999. [Online]. Available: http://dx.doi.org/10.1002/(SICI)1099-1689(199912)9:4<263::AID-STVR190>3.0.CO;2-Y

[22] G. Fraser and 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*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 416–419. [Online]. Available: http://dx.doi.org/10.1145/2025113.2025179

[23] M. B. Bashir and A. Nadeem, "A fitness function for evolutionary mutation testing of object-oriented programs," in *IEEE 9th International Conference on Emerging Technologies (ICET), 2013*, Dec 2013, pp. 1–6. [Online]. Available: http://dx.doi.org/10.1109/ICET.2013.6743531

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

[25] A. S. Banzi, T. Nobre, G. B. Pinheiro, J. C. G. Árias, A. Pozo, and S. R. Vergilio, "Selecting mutation operators with a multiobjective approach," *Expert Systems with Applications*, vol. 39, no. 15, pp. 12 131–12 142, 2012. [Online]. Available: http://dx.doi.org/10.1016/j.eswa.2012.04.041

[26] R. A. Silva, S. do Rocio Senger de Souza, and P. S. L. de Souza, "A systematic review on search based mutation testing," *Information and Software Technology*, 2016. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2016.01.017

[27] A. A. L. de Oliveira, C. G. Camilo-Junior, and A. M. R. Vincenzi, "A coevolutionary algorithm to automatic test case selection and mutant in mutation testing," in *IEEE Congress on Evolutionary Computation, 2013*, June 2013, pp. 829–836. [Online]. Available: http://dx.doi.org/10.1109/CEC.2013.6557654

[28] B. Schwarz, D. Schuler, and A. Zeller, "Breeding high-impact mutations," in *Proceedings of the 4th IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2011*, 2011, pp. 382–387. [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2011.56

[29] E. Omar, S. Ghosh, and 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), 2014*, March 2014, pp. 165–170. [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2014.19

[30] A. Estero-Botaro, F. Palomo-Lozano, I. Medina-Bulo, J. Domínguez-Jiménez, and A. García-Domínguez, "Quality metrics for mutation testing with applications to WS-BPEL compositions," *Software Testing, Verification and Reliability*, 2014.