This is a post-peer-review, pre-copyedit version of an article published in Software Quality Journal. The final authenticated version is available online at:

https://doi.org/10.1007/s11219-020-09530-1

Software Quality Journal manuscript No.
(will be inserted by the editor)

Evaluation of Alternative Design Choices for Evolutionary Mutation Testing by means of Automated Configuration

Pedro Delgado-Pérez · Francisco Chicano

Received: date / Accepted: date

Abstract Mutation testing is a well-established but costly technique to assess and improve the fault detection ability of test suites. This technique consists of introducing subtle changes in the code of a program, which are expected to be detected by the designed test cases. Among the strategies conceived to reduce its cost, *evolutionary mutation testing* (EMT) has revealed as a promising approach to select a subset of the whole set of mutants based on a genetic algorithm (GA). However, like any other metaheuristic approach, EMT's execution depends on a set of parameters (both classical of GAs and context-specific ones), so different configurations can greatly vary its performance. Currently, it is difficult to clarify what are the best values for those parameters by applying manual parameter tuning, and whether new design choices could improve its effectiveness with other combinations of values. The experience carried out in this paper applying iterated racing, a well-known automated configuration algorithm, reveals that EMT's performance has been undervalued in previous studies; the new configuration found by iterated racing was able to enhance EMT's results in all C++ object-oriented programs used in the experiments. This study also confirms alternative design choices as convenient options to improve EMT in this context, namely, detecting and penalizing equivalent mutants by means of Trivial Compiler Equivalence, and

This work was partially supported by the European Commission (European Regional Development Fund - ERDF), the Spanish Ministry of Science, Innovation and Universities under projects RTI2018-093608-B-C33 and TIN2017-88213-R, the excellence network RED2018-102472-T, the University of Málaga, and Consejería de Economía y Conocimiento de la Junta de Andalucía (grant number UMA18-FEDERJA-003)

✉ Pedro Delgado-Pérez
Departmento de Ingeniería Informática, Escuela de Ingeniería, Universidad de Cádiz, Spain.
E-mail: pedro.delgado@uca.es
Tel.: +34 956 483243

Francisco Chicano
Escuela Técnica Superior de Ingeniería Informática, Universidad de Málaga, Spain.
E-mail: chicano@lcc.uma.es

learning which mutation operators produced live mutants in the past generations.

Keywords Mutation testing · evolutionary computation · genetic algorithm · automated configuration · iterated racing · equivalent mutants.

1 Introduction

Testing is widely accepted as a method to validate software systems. Its main objective is to detect faults and fix them before a version of our software is released. However, exhaustive testing is usually unfeasible. Therefore, we should reach a manageable set of tests that provides high coverage, thereby reducing the likelihood that faults will remain undetected. In this context, mutation testing is gaining increasing attention to assess and improve the quality of test suites (Papadakis et al., 2019). This is a fault-based technique that imposes more stringent criteria than traditional test coverage criteria (e.g., statement or branch coverage); mutation testing challenges the test suite to reveal the existence of different changes in the source code. These changes are introduced following some predefined transformations, guided by the so-called *mutation operators*. These modifications are known as *mutations*, and the resulting versions of the program as *mutants*. We then say that a mutant is killed if the outputs of the test suite execution differ from those of the original program. Otherwise, the mutant is alive and may represent an opportunity to improve the test suite.

Mutation testing usually involves a trade-off between cost and effectiveness: the probability to discover deficiencies in the test suite increases with the number of mutants generated, but the more mutants the more expensive the process will also be. With the aim to resolve this trade-off, different techniques have been proposed in the past, such as *mutant sampling* (random selection of a subset of mutants, Budd (1980)), *selective mutation*, (selection of a subset of mutation operators, Barbosa et al. (2001)) or *higher order mutation* (combination of two or more mutations in the same mutant, Jia and Harman (2008)). In this paper, we focus on *evolutionary mutation testing* (EMT) (Delgado-Pérez and Medina-Bulo, 2018), a search-based technique that uses a genetic algorithm (GA) to favor the selection of mutants with the potential to assist a tester in the refinement of a test suite (i.e., mutants that remain alive). Notably, this technique has shown to be able to outperform mutant sampling and selective mutation (Domínguez-Jiménez et al., 2011; Delgado-Pérez and Medina-Bulo, 2018) in previous experimental procedures.

As other metaheuristic methods, the execution of EMT depends on a number of configurations options. Thus, the selected configuration can greatly affect its effectiveness in practice. Therefore, setting appropriate values for these parameters and optimizing the algorithm's performance is required by means of previous experiments. Experiments to find good parameter settings were conducted by Domínguez-Jiménez et al. (2011). Despite the efforts, we note

some issues with their approach: (1) the parameter tuning was done manually, so the authors could only focus on a limited set of configuration options and values for those parameters; (2) the experiments were undertaken using a specific set of instances (programs coded in WS-BPEL language) and mutation operators for that language. As a result, it is unclear whether the same combination of values for those parameters is suitable when we move to other contexts; (3) it is also uncertain how a refinement of the GA will impact the best configuration known so far.

Taking into account the aforementioned problems, this means that we might have underestimated the effectiveness of EMT —especially, if each particular context requires different parameter values or if an optimal configuration was not found in the original experiments–. In fact, all experiments applying EMT after that (Delgado-Pérez et al., 2017), (Delgado-Pérez and Medina-Bulo, 2018), (Gutiérrez-Madroñal et al., 2019) have followed the presumably best parameter settings reached in that paper (Domínguez-Jiménez et al., 2011). This is because restarting the same experiments to find a tailored combination in each new context is tedious and costly, especially if the parameter tuning has to be done completely by hand. Currently, it is also especially challenging to give evidence on whether suggested refinements of the GA provide any benefits; as a consequence of this, we might be missing a combination of parameters for the GA that better suits the new design choices.

In this paper, we assess the application of automated configuration algorithms to better exploits the benefits of this evolutionary approach for the selection of mutants. We make use of the package *irace* (López-Ibáñez et al., 2016) to this end, which offers a method called *iterated racing* for the automatic configuration of parameters. This procedure iteratively performs a number of races with a set of candidate configurations to select the best ones. This methodology allows dealing with the application of EMT in different contexts, where the best configuration settings can change, and allows assessing the behavior of alternative design choices for the algorithm. Taking advantage of this methodology, we add four alternative design options to the study in this paper (including two previously proposed by Gutiérrez-Madroñal et al. (2019)) in order to evaluate their ability to improve EMT's performance. These design options, that have not undergone a configuration process yet, are:

1. **Tournament selection** as the method for the selection of mutants for reproduction, with the possibility to adjust the tournament size.
2. **Equivalent mutant detection and penalization**. Equivalent mutants, despite the injected mutation, do not change the program's functionality. As a result, they are not helpful to improve the test suite. With this in mind, this option penalizes the fitness of equivalent mutants to prevent the propagation of their features in the search. This can be achieved with the help of Trivial Compiler Equivalence (Papadakis et al., 2015), a well-known technique to detect some equivalent mutants automatically.

3. **Guided mutation operator**, which implements a strategy to favor the application of operators that produced live mutants in previous generations of the execution.
4. **Pure random individual generator**, which achieves a uniform distribution over all mutants, unlike the current individual generator.

This study investigates the automated configuration of parameters for EMT in the context of mutants derived from real C++ object-oriented programs (Delgado-Pérez et al., 2017). The results of the experimental procedure reveal that the best configuration automatically found differs from the configuration used so far, and it includes two of the new extensions: equivalent mutant detection and penalization and guided mutation operator. With the new combination of values for these parameters, we observe reductions in the percentage of mutants generated in all the 7 programs used in the experiments (3.8% on average in the best case), including one program reserved for validation. In addition, the new configuration also reduces the standard deviation (3.1% on average in the best case), which means that the variability of different executions is lower now. In summary, this paper reinforces the usefulness of this search-based strategy for the efficient selection of mutants, and provides an analysis of a wider range of design choices. The methodology proposed in the paper overcomes the problem of parameter configuration, which is an important step towards building operative tools for companies that adapt to the particularities of their developments.

The structure of the paper is as follows. Section 2 describes the principles behind EMT and the current configuration of its GA. Section 3 explains the four design choices included in the study, and Section 4 presents the proposed methodology to automatically adjust EMT to each context. Section 5 introduces the research questions and shows the setup of the experimental procedure. Section 6 presents and discusses the results. Finally, Section 7 and Section 8 presents related work and the conclusions and future work, respectively.

2 Evolutionary Mutation Testing

2.1 Description

Evolutionary mutation testing (EMT) seeks to reduce the cost of applying mutation testing without a significant reduction of its effectiveness. EMT runs in the hope that the subset of selected mutants contains a large proportion of those that can be used by a tester to enhance the fault detection capability of the test suite. To achieve this, EMT relies on a genetic algorithm (GA) (Goldberg, 1989), which propagates the information on the most useful mutants in a generation to the mutants produced in the following generations. In summary, the search performed by the GA is expected to gather as many live mutants as possible in the resulting subset of mutants.

2.2 Genetic algorithm: steps

EMT works on the basis that a GA will be able to find useful mutants for the test suite improvement starting from a small subset of randomly selected mutants (first generation). This can be done by iteratively evolving and transferring the information on the most promising mutants to the rest of the generations. To that end, the GA follows these steps in each generation:

1. The mutants are executed against the current test suite. The output of this step is an *execution matrix*, that contains which test cases detect which mutants.
2. The *fitness function* is calculated for each of the mutants based on the number of test cases that kill them.
3. The next generation of mutants is then produced as follows:
   - A small subset of the mutants in the generation is randomly generated to preserve the diversity.
   - The rest of the mutants are derived from the mutants in the current generation. A *selection method* uses the previously calculated fitnesses to pick the mutants that will transfer their information to the next generation. *Reproductive operators* are then applied to the selected mutants.
4. A termination condition is evaluated. If the condition is met, the algorithm stops the execution and returns the mutants generated during the process. Otherwise, all these steps are repeated.

The fitness function, the relevant features of the mutants for the encoding scheme, and the selection and reproductive operators will be described in the following subsections.

2.3 Fitness function

EMT's aim is to select mutants for the refinement of the test suite, and this can be mainly achieved with mutants that require specific test cases to be killed. Therefore, EMT's fitness function rewards each mutant depending on the number of test cases that are able to reveal the mutation. This information is extracted from the aforementioned execution matrix (see step 1 in the previous section). In this regard, from the perspective of the fitness function:

- The *best valued* or *most useful mutants* are those not killed by the current test suite. As such, they receive the maximum fitness. Being M the number of mutants and T the size of the test suite, the fitness of live mutants is:

$$fitness = M \times T \tag{1}$$

- As for killed mutants, their fitness is lower than $M \times T$. The more test cases kill the mutant, the less valuable the mutant is and its fitness is penalized proportionally. Furthermore, the fitness of a mutant decreases when the

Operator	Location		Operator	Location
5	**7**	→	**5**	**8**

Fig. 1 Example of application of mutation operator (the mutation location is perturbed).

Operator	Location		Operator	Location
5	**7**		**5**	**4**
Operator	Location	→	Operator	Location
9	**4**		**9**	**7**

Fig. 2 Example of application of crossover operator between two mutants.

test cases killing that mutant kill many other mutants in turn. Being t the number of test cases killing that mutant, and m the number of mutants killed by those test cases (counting repeated mutants killed by different test cases), then the final fitness of a killed mutant is:

$$fitness = (M \times T) - t - m \tag{2}$$

As such, since the selection method promotes the selection of mutants with a high fitness, it is expected that the sum of the fitnesses in each generation will increase over time.

2.4 Mutant characteristics and encoding scheme

Two are the main features that characterize a mutant: the mutation operator that generates the mutation and the part of the code mutated. The GA uses these two characteristics to uniquely identify each mutant or individual. As such, an individual is encoded with two fields: an identifier to indicate *the mutation operator* and a number to represent the *mutation location* in ascending order of appearance in the code.

In fact, the GA works under the assumptions that:

- A mutation operator that generates apparently useful mutants is likely to produce similar mutants.
- A mutation location that apparently is not well covered by the current test suite is likely to be surrounded by other locations in a similar situation. The notion of the proximity of locations varies depending on the domain; for instance, two different mutations may be near when they were in the same statement, function or class.

The GA then searches for subgroups of useful mutants based on those assumptions thanks to the application of selection and reproductive operators.

2.5 Selection and reproductive operators

EMT counts with two classical variation operators to generate new individuals:

- *Mutation operators*[1], where the value of one of the fields representing an individual (i.e., *Operator* or *Location*) is slightly perturbed to produce new nearby mutants.
- *Crossover operators*, where two individuals (parents) swap part of their information to generate two new individuals (children).

Figure 1 and Figure 2 present an example of application of the mutation and crossover operator, respectively.

2.6 Current configuration parameters

EMT requires to adjust a number of configuration parameters previous to its execution. In its original version, EMT's execution can be configured through the following classical parameters in GAs: *population size* or P_s, as a percentage of the total number of mutants given a source code and a set of operators; *crossover and mutation probability* or p_c and p_m, respectively, to adjust the frequency in which these two variation operators are applied; and *new individuals generated randomly* or N, which indicates the percentage of mutants randomly produced in each generation to preserve the diversity (except for the first generation, where all mutants are randomly generated). The rest of the mutants in a generation, that is 100% - N, are therefore produced by the application of reproductive operators; this percentage is referenced in this paper with the letter R.

Table 1 Current parameter settings for EMT.

Parameter	Description	Value
P_s	Population size	5%
N	Individuals generated randomly	10%
R($100\% - N$)	Individuals generated by reproductive operators	90%
p_c	Crossover probability	70%
p_m	Mutation probability	30%

Table 1 presents the best values for these parameters that Domínguez-Jiménez et al. (2011) found in their experiments. As an example, if the mutation tool generates 200 mutants in a program, the population size in each generation is 10 ($P_s = 5\%$); after the first generation, 9 out of those 10 mutants are produced by reproductive operators ($R = 90\%$) while only one is

[1] Please note that the term *mutation operator* has a different meaning in the context of mutation testing and in the context of GAs. To avoid confusion, the former will be referred to as *operator* and the latter as *mutation operator* from now on.

generated randomly ($N = 10\%$). Finally, the probability of each of those 9 mutants to be generated by crossover or mutation is $p_c = 70\%$ and $p_m = 30\%$, respectively.

3 Design choices

Even though the original version of EMT performed reasonably well in the past, other design choices can be considered with a view to enhancing its performance. In this work, we assess several extensions (both general of GAs and specific to EMT) to know if they are able to further improve the effectiveness of this technique. The final goal of our work is to evaluate whether some of these options lead to improved results, especially when they are applied in combination instead of isolation. In the following, we explain the alternative options that will be studied in this work:

1) Tournament selection The original authors of EMT opted for the *roulette wheel method* as the selection operator of individuals. In the roulette wheel method, the probability of an individual to be selected is proportional to its fitness. As noted by Domínguez-Jiménez et al. (2011), this method is characterized by its quick convergence, which fits well with the purpose of EMT (e.g., reduce the number of mutants as much as possible). However, such an assumption has not been confirmed empirically. As such, we will evaluate the well-known *tournament selection method* as an alternative to the roulette wheel method (Eiben and Smith, 2015). Figure 3 shows an example of this selection method, where a number of mutants are selected at random (in the example, the size of the tournament is 2 and the mutants *M1* and *M3* are randomly selected). Then, the winner of the tournament from the selected candidates is the one with the highest fitness.

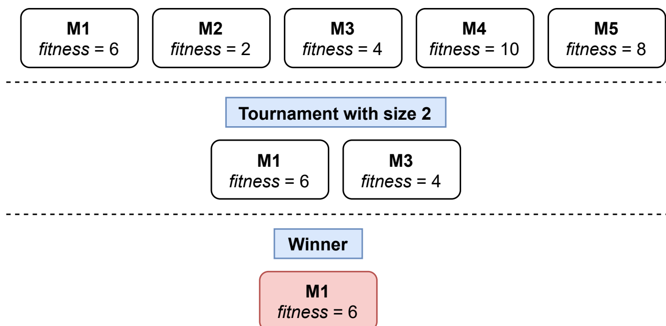

Fig. 3 Example of the tournament selection method, where each box represents a mutant.

2) Equivalent mutant detection and penalization Mutation testing suffers from two major problems (Papadakis et al., 2019). The first is the large number of mutants that can be generated, which increases with the size of the source code and the set of operators. The second problem relates to mutations that do not actually change the program's functionality; mutants containing such mutations are known as *equivalent mutants*. Equivalent mutant identification is an undecidable problem, and thus they require manual and time-consuming revision. Fortunately, different techniques can help cope with equivalent mutants (Madeyski et al., 2014). One of such techniques is *Trivial Compiler Equivalence* (TCE) (Papadakis et al., 2019), a well-established technique to detect some equivalent mutants thanks to compiler optimizations. Once the compiler has optimized the code of the original program and a mutant, if the resulting binary files are exactly the same, then the mutant can be determined as equivalent.

This option consists of two steps. In the first step, TCE is applied to the mutants in a generation to reveal some equivalent mutants. In the second step, that information is used to penalize the fitness of those mutants (their fitness changes to 0). This modification, assigning the lowest fitness to those mutants, avoids that the GA selects them for reproduction. Figure 4 illustrates the two steps added to the GA when this design choice is enabled. As it can be seen, mutants *M2* and *M4* turn out to be equivalent to the original program, and their fitness is punished with 0.

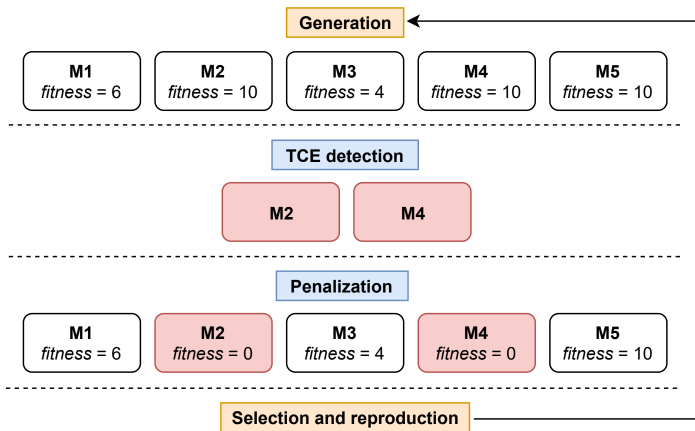

Fig. 4 Example of equivalent mutant detection and penalization integrated into the GA.

3) Guided mutation operator According to Section 2.5, the current mutation operator implemented in the GA perturbs the value of the *Operator* field when that field is selected for mutation. Gutiérrez-Madroñal et al. (2019) proposed the use of a new mutation operator that learns from the behavior of each operator in past generations, based on the work by Qingfu Zhang et al.

(2005). The goal of this new option, known as *guided mutation operator*, is to favor the application of operators that are seemingly useful for test suite improvement. More specifically, the guided mutation operator selects an operator with a probability proportional to the percentage of all live mutants generated by that operator in the previous generations of the execution.

As an example, consider a case where we have three operators, as illustrated by Figure 5. The GA, in the generations preceding the current generation, has produced a number of mutants from those operators, including the following live mutants:

- *Operator 1*: 2 live mutants.
- *Operator 2*: 3 live mutants.
- *Operator 3*: 1 live mutants.

In this example, operator 2 has produced the greatest number of live (and apparently useful) mutants so far, and it is seemingly worth exploring in greater depth than the other operators. Therefore, the guided mutation operator will select the 1st operator with probability 2/6, the 2nd with 3/6 and the 3rd with 1/6.

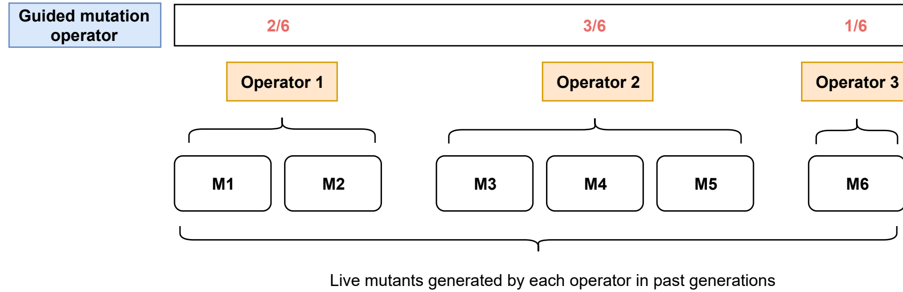

Fig. 5 Example of the probability of selection of operators with the guided mutation operator.

4) Pure random individual generator As previously mentioned, the GA generates N individuals in each generation in a random way (see step 2 in Section 2.2). The current individual generator in EMT first selects a random operator and then picks, also at random, one of the possible locations associated with that operator (Domínguez-Jiménez et al., 2011). However, as noted by Gutiérrez-Madroñal et al. (2019), since not all the operators produce the same number of mutants, this individual generator does not achieve a uniform distribution over the whole set of mutants. As an alternative, they proposed to shuffle the entire list of mutants and then select one of them randomly. As a result of this individual generator, all mutants are selected with equal probability.

The current non-uniform generator and the proposed uniform one will be hereinafter referred to as *partially random* and *pure random* generator, respectively. Figure 6 shows an example of the difference between both types of generators. Focusing on the partially random generator, mutant *M6* from *Operator 3* has a higher probability to be selected than the others because, once this operator is selected (with probability 1/3), it is the only candidate mutant; in contrast, mutant *M3* has a lower probability because, once *Operator 2* is selected (with probability 1/3), there are three candidates (*M3*, *M4* and *M5*) and, in turn, each one is selected with probability 1/3 (i.e., 1/9 in the end). Contrarily, the probability of each mutant to be selected is the same with the pure random option.

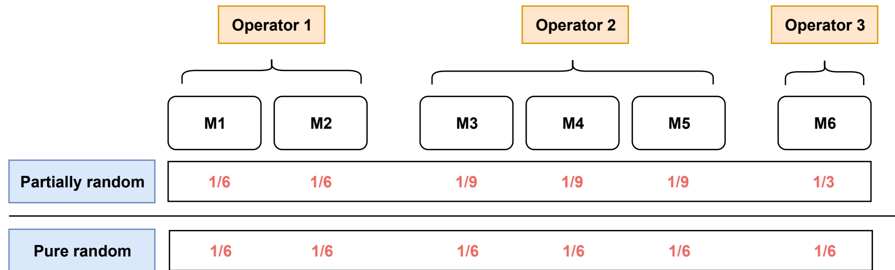

Fig. 6 Example of the difference between the partially random and pure random individual generator.

4 Automated Algorithm Configuration

4.1 Methodology

The four aforementioned design choices significantly increase the number of parameters to be configured. It also increases the number of combinations of values for all the parameters. As a consequence, performing manual tuning of parameters to reach an optimal combination becomes impractical. The *context* is also important when applying EMT because of the variability of several factors. With context, we refer, among others, to the following aspects:

- The number of mutants of each type (killed, equivalent,...).
- The distribution of mutants across the source code.
- The number of operators, how often they are applied and the percentage of all mutants that each operator generates.

As a consequence of all this, in this paper, we emphasize the need to use an automated approach to find good combinations of values for adjusting the GA to each context. Specifically, we propose the application of *iterated*

racing (López-Ibáñez et al., 2016), which has been used to configure similar algorithms in the past (Lima and Vergilio, 2017). An iterated racing procedure combines a search with a racing algorithm that allows discarding underperforming candidates in a set of instances given a cost measure. The process is iterative because the procedure undertakes a number of races with different sets of candidate configurations. *Candidates*, *instances* and the *cost measure* are defined in the scope of EMT as follows:

– **Candidates** are each of the possible configurations for EMT, including original and new design choices evaluated in this paper.
– **Instances** are a combination of (1) the source code of the programs, (2) the set of mutations that can be injected in their code and (3) the associated test suite. Section 5 will properly describe the form that all these elements take in our evaluation.
– **Cost or quality measure**: we say that a configuration candidate will perform better than another one when the application of EMT with that configuration would induce a higher degree of test suite improvement after selecting the same number of mutants with both configurations. The cost measure used in our study is detailed in the following subsection.

Therefore, in the execution of the iterated racing, the algorithm randomly selects an instance, and then executes EMT applying the candidate configurations with a new seed. The output is a measure of the quality of each of the assessed configurations with respect to that instance. After repeating the same process a number of times with different instances, the sum of the costs of each configuration helps the algorithm decide which configurations perform better and which ones can be discarded.

4.2 Cost measure

In a previous paper (Delgado-Pérez et al., 2017), we noted the importance of using a measure that reflects the degree of test suite refinement that would be achieved by the subset of mutants returned by EMT. A simple measure is counting the number of selected live mutants, but this option presents two issues: first, a single test case could suffice to kill some of those mutants that remained alive; second, equivalent mutants act as false positives.

Therefore, a more accurate measure for EMT should estimate the number of new test cases that would be added thanks to those mutants. Such a metric entails the following steps:

1. *Reach a mutant-adequate test suite* prior to the execution of EMT. This previous step implies adding as many new test cases as required to kill all non-equivalent mutants.
2. *Execute EMT*, to obtain the subset of mutants selected by the technique.
3. *Calculate the level of test suite refinement achieved*, that is, observe the proportion of the test cases in the mutant-adequate test suite that the

selected subset of mutants would induce. That proportion is represented by P.

As an example, let's say that we inspect all live mutants in a program previous to the execution of EMT, and we are able to reach a mutant-adequate test suite with size 10 test cases (t_1-t_{10}). Then, we run EMT and, after two generations, 6 mutants are selected (m_1-m_6). Now, we can analyze which test cases in the mutant-adequate test suite kill each of those mutants with the aid of an execution matrix (see Section 2.2), like the one shown below. By processing this matrix with a minimization algorithm, we can observe that 5 out of those 10 test cases suffice to kill all the mutants selected by the GA so far: t_1 (m_1), t_3 (m_2), t_4 (m_3 and m_5), t_8 (m_4) and t_{10} (m_6). Therefore, at that point, the measure of EMT's effectiveness is $P = 50\%$. Note the importance of calculating this measure as a percentage: the size of the mutant-adequate test suite can be different for each program and, therefore, reaching a certain value of P will require a different number of test cases in each program. The value of P is the cost measure used in our experiments in the next section.

	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
m_1	1	0	0	0	1	0	0	0	0	0
m_2	0	0	1	0	0	0	0	0	0	0
m_3	0	0	0	1	0	0	0	0	1	0
m_4	0	0	0	0	0	0	0	1	0	0
m_5	0	0	0	1	0	0	0	0	0	0
m_6	0	0	0	0	0	0	0	0	0	1

1: t kills m; 0: t does not kill m.

We should remark that our cost measure implicitly considers the effect of the equivalent mutants. If an equivalent mutant is selected by EMT, this mutant will not be able to increase the quality measure because equivalent mutants do not induce the design of the test cases in the mutant-adequate test suite. Therefore, the more equivalent mutants appear in the set of selected mutants, the less the test suite improvement that will be achieved.

5 Evaluation

5.1 Research questions

In the following items, we present the research questions that we want to answer in this work:

- **RQ1: Can we find a better configuration to improve EMT's performance?**
  This first question implies the application of iterated racing in combination with EMT to find a convenient configuration for the GA in a given context. To answer this question, we will compare the new values for the parameters with those applied in previous experiments.

– **RQ2: Which of the four design choices evaluated in this study are enabled in the best parameter settings found?**
  With this question, we seek to know whether the proposed additions to EMT take part in the best parameter settings found, which will mean that they actually improve EMT's performance. It will also be interesting to analyze how many times these options are applied over the iterated racing execution (i.e., whether they are mostly contained in configurations surviving the race), and whether the results agree in a cross-validation process.
– **RQ3: How much does the new combination of values for the parameters improve EMT's performance?**
  Finally, if a better configuration is found in the experiments, we want to know whether the improvement is generalized over the set of instances and how much we have undervalued the actual effectiveness of EMT up to now. We also want to know about the role of the termination condition in the process of parameter tuning.

Table 2 Data about the instances used in the experiments.

	Mutants	**Operators**	**Alive**	**TCE-Equivalent**
TCL Pro	137	6	45	9
XmlRPC++	151	18	76	4
Dolphin	219	14	103	19
MuParser	226	10	133	50
TinyXML2	614	14	159	30
MySQL	683	13	446	11
QtDOM	1,146	15	348	20

5.2 Context and set of instances

In this study, we apply EMT to a number of instances that belong to a specific context: real-world programs coded in C++ and the collection of mutants derived from them with a set of operators at the class level (i.e., operators focused on the object-oriented features provided by the language). The mutants are generated using the list of class operators included in the mutation tool *MuCPP* (Delgado-Pérez et al., 2017). Also, the four proposed refinements have been integrated into GAmera (Domínguez-Jiménez et al., 2009), which implements the GA described in Section 2. GAmera is a language-independent framework, so the new extensions of the algorithm can be applied to different contexts (programming languages and sets of operators).

Our set of instances consists of 7 open-source programs: Matrix TCL Pro (2019), XmlRPC++ (2019), Dolphin (2019), Tinyxml2 (2019), MySQL Server (2019), QtDOM (2019) and MuParser (2019). In the experiments, 6 out of the 7 instances are used for the parameter tuning and one is reserved for

validation. In the first part, *MuParser* is the instance reserved for testing; in the cross-validation part, each of the programs acts as the validation instance. Table 2 shows the number of mutants that can be derived from the source code of each of these programs, operators applied, mutants that remain alive after the execution of the initial test suite and equivalent mutants detected by TCE (TCE-Equivalent mutants). Recall that TCE is used to penalize known equivalent mutants in the second design choice described in Section 3.

As noted in Section 4.2, the cost measure requires of a mutant-adequate test suite that kills non-equivalent mutants. As initial test suites, we used the test suite that is distributed with each program under test, and we executed them on each of the generated mutants. The execution results allowed us to obtain a set of killed and live mutants. Then, we manually reviewed each of the mutants that remained alive: non-equivalent mutants led us to incorporate new test scenarios in order to kill them; the rest of the mutants were tagged as equivalent. We should note that some of the test cases designed for one of the live mutants, in turn, killed some other live mutants. Also, note that we checked that TCE-equivalent mutants were a subset of the manually-determined equivalent mutants.

5.3 Candidate configurations

Table 3 collects the set of configuration parameters to be optimized. This table includes both classical parameters, already shown in Table 1, and new parameters associated with the four alternative design choices:

- A **selection operator** (S_{op}), either *roulette wheel* or *tournament selection*. In the latter case, a *tournament size* (T_s) is also required.
- Whether the mechanism for **equivalent mutant detection and penalization** (EQ) is *enabled* or *disabled*.
- A **mutation operator** (M_{op}), either the *classical operator* (where the value of one of the fields is perturbed) or the *guided mutation operator* (where the value of the *Operator* field is selected based on the proportion of all the live mutants produced by that operator in previous generations). In the latter case, a *sampling rate* (G_{sr}) is also required to indicate the probability that the guided mutation operator is applied once the mutation operator is selected —in the rest of the cases, the classical mutation operator will be used as always.
- An **individual generator** (IG), either the current *partially random* generator (which depends on the number of mutants produced by each operator) or the new *pure random* generator (which achieves a uniform distribution over the set of all mutants).

The table provides the range of values for each parameter and, when applicable, the conditions under which a parameter is taken into account. The table also shows the initial values for those parameters, which correspond with the original configuration of EMT. In this way, the automated configuration algorithm will take these parameter values as the starting point of the search.

Table 3 Set of parameters and their values for tuning EMT (the line separates classical and alternative design choices). The parameters marked with '*' are not optimized directly, but they are assigned a value depending on other parameters (column conditions).

P.	Description	Values	Conditions	Initial
P_s	Population size	(3%, 7%)	-	5%
N	Individuals generated randomly	(10%, 25%)	-	10%
R*	Individuals generated by reproductive operators	(70%, 90%)	100% - N	90%
p_c	Crossover probability	(65%, 75%)	-	70%
p_m*	Mutation probability	(25%, 35%)	100% - p_c	30%
S_{op}	Selection operator	{Tournament, Roulette}	-	Roulette
T_s	Tournament size	(2, 4)	S_{op}=="Tournament"	-
EQ	Equivalent detection and penalization	{enabled, disabled}	-	disabled
M_{op}	Mutation operator	{Classical, Guided}	-	Classical
G_{sr}	Sampling rate (guided)	(70%, 90%)	M_{op}=="Guided"	-
IG	Inidividual generator	{Partial, Pure}	-	Partial

5.4 Iterated racing

The iterated racing is performed thanks to the application of the tool *irace* (López-Ibáñez et al., 2016). We configure *irace* to execute with a budget of 3,000 experiments and to perform a race (and therefore discard underperforming configurations) each time all candidate configurations are executed once in each of the 6 instances. As previously explained, the cost measure is the percentage of the mutant-adequate test suite achieved with the mutants selected in the execution.

Regarding the *stopping condition* of EMT's execution, we should note that we observed significant differences among the percentage of mutants required in each instance to reach a percentage of the mutant-adequate test suite. Therefore, we set a different stopping point for each instance; in this way, the cost measure remains in a controlled range of values that allows *irace* to discern between the performance of different configurations. Namely, the GA stops when reaching the percentage of mutants originally generated by EMT to achieve $P = 75\%$ in each particular instance (average of 30 executions).

6 Results and Discussion

In the following subsections, we show the results of the experiments and answer the three research questions. Finally, we discuss the threats to the validity of the results.

6.1 Iterated racing execution

Table 4 presents the best configuration found applying iterated racing with *irace*. It shows the best value for each of the parameters (see Table 3) when they are assessed together in the same execution of the automated configuration algorithm. As for Table 5, it shows the four next elite configurations obtained at the end of the run. As it can be seen, three of these elite configurations are quite similar to the best configuration, which reinforces the adequacy of the best values found. The other one, in contrast, is significantly different from the rest of the configurations. This fact shows that different combinations of values can work well together (that is, the parameters influence each other), and emphasizes the need for finding the combination that is best suited to each particular context.

Table 4 Best parameter settings found using the automated approach.

	P_s	N	p_c	S_{op}	T_s	EQ	M_{op}	G_{sr}	IG
Best	4%	20%	75%	Roulette	-	enabled	Guided	83%	Partial

Table 5 Next elite configurations found using the automated approach, ordered from best to worst.

	P_s	N	p_c	S_{op}	T_s	EQ	M_{op}	G_{sr}	IG
Elites	4%	17%	73%	Roulette	-	enabled	Guided	80%	Partial
	4%	14%	73%	Roulette	-	enabled	Guided	83%	Partial
	5%	12%	66%	Roulette	-	disable	Classical	-	Partial
	4%	10%	73%	Roulette	-	enabled	Guided	78%	Partial

Figure 7 allows us to observe the distribution of sampled values for these parameters in the execution of the race. The preferred values for the classical parameters in EMT are as follows:

- **Population size**. From the graph, it appears that 4% (instead of 5%) is the size favored by the algorithm, while the values 3% and 7% have been scarcely explored.
- **Individuals generated randomly**. It seems that values around 20% are a good option for this parameter (note that the algorithm sampled values for this parameter starting from 10%, which may have increased the number of candidate configurations with values around 10%). Interestingly, the value of this parameter decreases as the elite configurations worsen their performance (from 20% in the best configuration to 10% in the last elite configuration). This fact supports the trend observed in the graph to explore higher values for this parameter. However, the varying values for this parameter in the set of elite configurations suggest that it might not be as decisive in the performance of EMT as other parameters.

– **Crossover probability**. The frequency of this probability is quite spread across the range, with 65% as the most sampled percentage. Still, we can observe an increasing exploration tendency of values over 70%, which suggests that increasing the original probability (70%) provides better results.

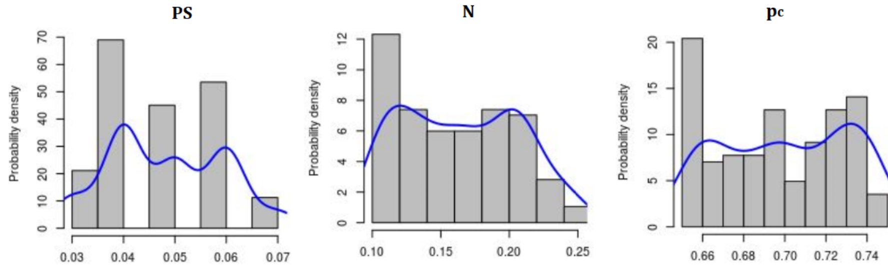

Fig. 7 Classical parameters sampling frequency: population size, individuals generated randomly and crossover probability.

Answering RQ1, the automated algorithm was able to find a different parameter configuration from the one used so far in experiments with C++ object-oriented mutants. This configuration modifies the value of the three classical parameters (population size, individuals generated randomly and crossover probability). The fact of having found a better configuration for EMT in this study does not necessarily mean that the parameter settings extrapolate to other contexts, but it underpins the need to perform parameter tuning in different contexts.

6.2 Evaluation of new design choices

Regarding the four alternative design choices, the following can be observed:

– **Selection operator**. The automated configuration algorithm confirms the usefulness of the roulette wheel as the selection operator in the case of EMT. Only one configuration enabling the tournament selection with size 3 appeared among the elite configurations of the three first iterations. The superiority of sampled configurations using the roulette wheel method can be seen in the frequency graph in Figure 8.
– **Equivalent mutant detection and penalization**. The option of detecting and punishing equivalent mutants is enabled in the best configuration as well as in three of the other elite configurations. The frequency graph in Figure 9 shows that the algorithm has evenly explored both options, but it is more inclined towards enabling this option in the end. As a conclusion, this option facilitates that the GA focuses on finding useful mutants for

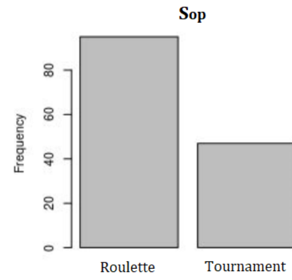

Fig. 8 Frequency graph of selection operators.

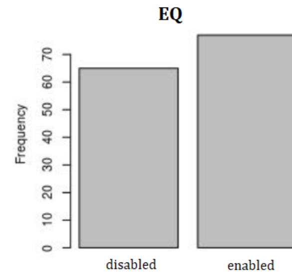

Fig. 9 Frequency graph of equivalent mutant detection and penalization option.

the improvement of the test suite, delaying the selection of some equivalent mutants.

– **Mutation operator**. The guided mutation operator appears as the preferred option over the classical mutation operator, as it can be graphically seen in Figure 10. The values for the sampling rate (G_{sr}) mainly concentrate around 80%. The most used sampling rate is 83%, which means that the classical mutation operator is still applied 17% of times. This rate is a bit lower than 90%, the percentage used in the experiments by Gutiérrez-Madroñal et al. (2019). In summary, we can conclude that learning from previous generations benefits the search.
– **Individual generator**. It is clear from the results that the *pure random individual generator* is not an advisable option in our context. This is supported by the graph in Figure 11. In fact, all elite configurations found in each iteration presented the partially random option. We believe that this outcome is connected with previous results using the same set of class operators (Delgado-Pérez et al., 2019); that study suggests that removing some of these operators may not be recommendable because each class operator targets a different object-oriented feature of the language (inheritance, polymorphism, method overloading, constructors, etc). We find that the partially random generator promotes the generation of mutants from different operators (regardless of the mutants that each operator produces) while the pure random one is more likely to select mutants from the most

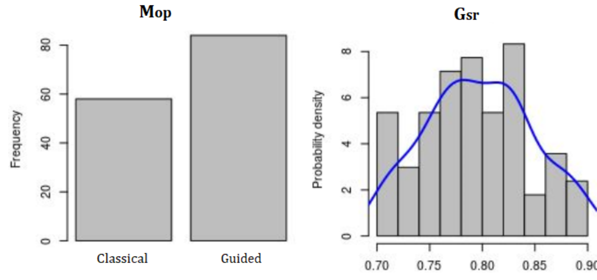

Fig. 10 Frequency graph of mutation operators (left) and sampling rates related to the guided mutation operator (right).

prolific operators. As a consequence, the pure random option may delay the selection of mutants from all operators and, therefore, that all the object-oriented features are properly covered.

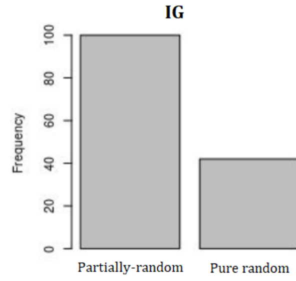

Fig. 11 Frequency graph of individual generators.

To analyze the consistency of these values, we carried out a process of leave-one-out cross-validation, that is, we executed the iterated racing algorithm seven times, leaving one of the instances out each time. The result of this process is a set of seven different configurations (those that were found as the best in each of the executions). Figure 12 shows the most selected value in these configurations for each parameter (the numerical parameters with more than five values have been divided in ranges). Additionally, this graph allows us to better understand which are the parameters with the greatest influence on EMT. The experiment reinforces $p_c = 72 - 75\%$, $S_{op} = Roulette$, $EQ = enabled$, $M_{op} = Guided$ and $IG = Partial$ as quite beneficial options, while the rest are apparently less decisive.

Answering RQ2, 2 out of 4 of the assessed design choices appear as convenient options for EMT in the targeted context. Namely, equivalent mutant detection and penalization and the application of the guided mutation operator. From the first, we can infer that equivalent mutants are often close to

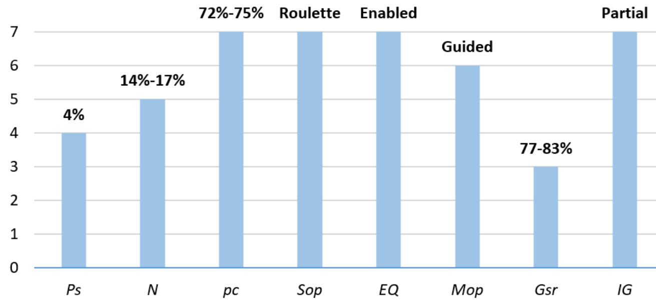

Fig. 12 Most selected value in each parameter after leave-one-out cross-validation. The most selected value or range is shown over each bar and the vertical axis represents the number of times that it appears in the seven configurations.

others (either in the code or with respect to the operators that produce them); the second one indicates that learning from previous generations is beneficial. Neither tournament selection nor pure random individual generation seems to be advisable options. Contrarily, the roulette wheel shows a quicker convergence, and the partially random generator fosters a greater diversity of mutants from different operators from the beginning of the search.

6.3 Assessment of EMT's performance

After analyzing the parameters in the best configuration found in detail, it is interesting to know its impact on EMT's performance. To this end, Table 6 presents the comparison of results between EMT with the default configuration and EMT with the best configuration automatically found. The table shows the results of 30 executions of EMT with both configurations in each instance, including the one reserved for validation. Note that we initially set 30 different seeds and we used them to execute EMT with both the state-of-the-art and the new configuration. Namely, we computed the mean and the standard deviation (SD) of the percentage of mutants that EMT selects before reaching the stopping condition ($P = 75\%$, the same condition set in the execution of the automated racing). Note that the lower the percentage of mutants, the more effective EMT is because it achieves the same level of test suite improvement with fewer mutants.

As it can be seen in the column *Difference*, the execution of EMT with the new configuration outperforms the original configuration in all the instances, both in the average and standard deviation. This means that the improved configuration allows reducing the percentage of mutants required as well as further stabilizes the GA. The greatest improvements are achieved in *TCL Pro* and *XmlRPC++* in both measures. Notably, the new configuration could also enhance the results in the programs *TinyXML2* and *QtDOM*, where the margin of improvement is lower (EMT already performed pretty well in them).

Table 6 Comparison of EMT with the *Initial* and *Best* configurations: mean and standard deviation (SD) of the percentage of mutants generated in 30 executions in each instance (including *MuParser*, the validation instance).

Instance	Measure	Initial	Best	Difference
TCL Pro	*Mean*	37.24	34.67	**-2.57%**
	SD	10.77	7.69	**-3.08%**
XmlRPC++	*Mean*	45.25	41.43	**-3.82%**
	SD	9.59	8.34	**-1.25%**
Dolphin	*Mean*	49.75	48.11	**-1.64%**
	SD	8.51	7.69	**-0.82%**
TinyXML2	*Mean*	19.26	18.42	**-0.84%**
	SD	4.38	3.94	**-0.44%**
MySQL	*Mean*	35.60	33.92	**-1.68%**
	SD	7.07	6.85	**-0.22%**
QtDOM	*Mean*	13.33	12.64	**-0.69%**
	SD	3.35	2.49	**-0.86%**
MuParser	*Mean*	42.71	41.19	**-1.52%**
	SD	8.58	7.74	**-0.84%**

Regarding the validation instance, we can observe a similar degree of enhancement to the one in the rest of the instances.

We additionally performed a statistical test to validate the significance of these differences. The null hypothesis in our test states that there is not a statistical difference between the percentage of mutants generated by EMT before and after the parameter tuning, while the alternative hypothesis states that the percentage of mutants with the new configuration is statistically lower. Namely, we run the Wilcoxon signed-rank test between the averages of the percentage of mutants generated in each case study using the original and the new configuration (values shown in Table 6). The result ($p\text{-}value = 0.0078$) leads us to accept the alternative hypothesis with 0.01 significance level.

Regarding the leave-one-out cross-validation, we analyzed the effect on EMT's performance of applying the different configurations obtained on average. The differences between the state-of-the-art and the obtained configurations remained reasonably stable in most of the programs, with the greatest average difference appearing in *XmlRPC++* (mean: -2.8; SD: -1.4). We observed greater variations among the results for *TCL Pro*, probably due to the few mutants in this program (mean: -1.1; SD: -1.5). On the contrary, the average result with the new configurations in *QtDOM* improved over the one shown in Table 6, both in mean (-1.0) and SD (-1.0). The rest of the results are: *Dolphin* (mean: -0.7; SD: -0.5); *TinyXML2* (mean: -0.6; SD: -0.1); *MySQL* (mean: -1.4; SD: -1.4) and *MuParser* (mean: -1.2; SD: -0.3).

We also wanted to know about the importance of the stopping condition in the process of parameter tuning, that is, what is the impact of a change in the termination point on EMT's performance when using the new configuration.

To achieve this, we repeated the experiments so that EMT stopped before ($P = 65\%$) and after ($P = 85\%$) the point for which the algorithm had been optimized (e.g., $P = 75\%$). Table 7 summarizes the results (Mean and SD) of the number of mutants generated when reaching both points with the original and the new configuration. As it can be observed, the difference between both configurations decreases in most of the programs (contrarily, it increases in *MySQL* and *QtDOM* with $P = 85\%$), and even the initial configuration performs better than the new one in some of them. This reveals the importance of optimizing the algorithm depending on the resources that will be dedicated to the testing process. Another interesting alternative would be to optimize the algorithm in the range between two stopping points. This would be achieved by considering an instance as a pair <*program*, *termination point*>; by analyzing each program with different termination points, *irace* could search for a more general configuration.

Table 7 Comparison of EMT with the *Initial* and *Best* configurations in the stopping points $P = 65\%$ and $P = 85\%$. Results with a negative value correspond to a difference in favor of the new configuration in the mean or the standard deviation (SD) of the percentage of mutants generated in 30 executions in each instance; otherwise, the difference is positive for the initial configuration.

Program	**P=65%**		**P=85%**	
	Mean	SD	Mean	SD
TCL Pro	–0.75	–0.17	+0.15	–1.28
XmlRPC++	+0.53	+0.88	–3.00	–0.57
Dolphin	+1.36	+1.22	–1.02	+0.50
TinyXML2	–0.32	–0.65	+1.70	+0.81
MySQL	–1.03	–0.33	–4.51	–1.53
QtDOM	–0.10	–0.41	–1.09	–1.56
MuParser	–0.58	+0.12	+0.85	+0.13

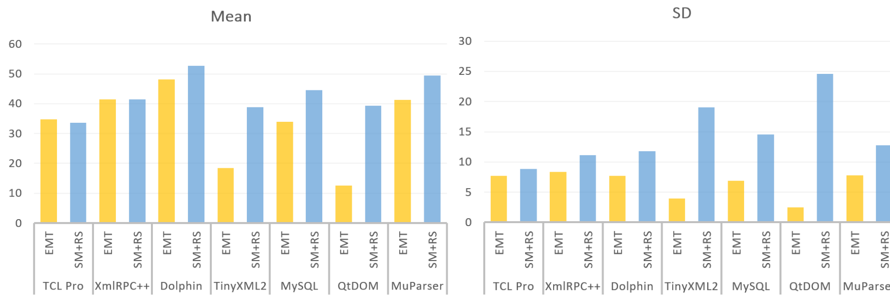

Fig. 13 Comparison between the percentage of mutants generated by EMT with the new automated configuration and random selection using a selective set of operators (SM+RS) with P=75%. Mean (left) and standard deviation (right) of 30 executions.

Finally, Figure 13 presents the results of a comparison between EMT and an improved version of selective mutation (i.e., selection of a subset of operators) combined with the random selection of mutants. In this improved version, first, a subset of operators is selected at random until reaching the stopping condition; second, in order to discard as many useless mutants as possible, mutants are selected at random from that subset of operators until reaching the stopping condition again. This version showed more competitive results than conventional selective mutation and random selection in previous experiments assessing EMT's effectiveness (Delgado-Pérez and Medina-Bulo, 2018). The figure shows the average and standard deviation of the percentage of all the mutants that are required by each technique in the programs under test. As it can be observed, the average results for both techniques are similar in the two programs with fewer mutants, but there is a considerable difference between them in favor of EMT in the rest of the cases, especially regarding the standard deviation.

Answering RQ3, the new configuration achieves improvements in all the instances, both in mean and standard deviation (3.8% and 3.1% in the best case, respectively), and these differences in favor of the new configuration also appear in the cross-validation process. This means that we can save a percentage of mutants with the new configuration and that the results of different executions are now less varied because they are spread out over a narrower range. In previous experiments, EMT outperformed both random selection and an improved version of selective mutation in most of these instances. The differences between EMT and these techniques now become more notable with the new configuration. Finally, the experiments suggest that the stopping point is an aspect that should not be set lightly when performing the parameter tuning.

6.4 Threats to Validity

The main threat to the validity of this study stems from the representativeness of the set of instances. We selected 7 widely used programs and libraries to train and test the system, which are of a different nature judging from the number of mutants, operators applied and the percentage of mutants that survive the execution of their respective test suites. The result in the program used for testing purposes makes us feel confident that the number of instances is appropriate and that the best configuration found does not suffer from the problem of overfitting to our set of instances.

Despite our best efforts, we cannot guarantee that this is the best possible configuration for the addressed context (simply because the mutants in each program are distributed differently). Therefore, this condition could only be determined with greater certainty at a larger scale with a larger set of programs. Also, note that computing an average of the values of each parameter in the different configurations is not convenient; as we have observed, the power of this approach is to find combinations of values for the parameters

that work well together. In any case, our main goal in this study was not to find the best possible configuration, but to give evidence on the benefits of performing parameter tuning in each context.

The study suggests that both equivalent mutant detection/penalization and the guided mutation operator improve the results of EMT in the context of C++ object-oriented systems. However, we cannot know whether they are actually recommendable options in general until the methodology of automated configuration presented in this paper is applied to other contexts. Regarding the first option, we should remark that mutants automatically flagged as equivalent by TCE can be safely considered as equivalent; as a result, those TCE-equivalent mutants selected during the execution of EMT with this option are guaranteed to be properly penalized. Nonetheless, the effectiveness of the choice to penalize equivalent mutants depends on the effectiveness of TCE in each instance.

7 Related Work

Search-based techniques have been extensively explored in the context of software testing (Khari and Kumar, 2019), with a special emphasis on their application for test case generation (McMinn, 2004; Rodrigues et al., 2018). Search-based approaches have also been developed in relation to mutation testing, these being collected by Silva et al. (2016) recently. Among other, metaheuristic search techniques have been used for the selection of operators (Banzi et al., 2012), the generation of mutation-based test cases (Papadakis and Malevris, 2011), the identification of subsuming higher order mutants (Jia and Harman, 2008) or the co-evolution of mutants and test cases (de Oliveira et al., 2013), being GAs the most used metaheuristic. In this study, we focus on the search-based problem of the selection of mutants with the potential to enhance a test suite at a reduced cost, and we also use a GA to this end. In the first studies, EMT was evaluated in the context of web service compositions (Domínguez-Jiménez et al., 2011). The assessment of this technique was later extended to other contexts, i.e., object-oriented programs (Delgado-Pérez et al., 2017; Delgado-Pérez and Medina-Bulo, 2018) and event processing queries (Gutiérrez-Madroñal et al., 2019), providing evidence that this approach could improve over other known selection techniques, such as mutant sampling (Budd, 1980) and selective mutation (Barbosa et al., 2001). In a related paper, Schwarz et al. (2011) also applied a GA to identify undetected mutants for the test suite improvement, where they considered the impact that mutations had on the code coverage and how these mutations were distributed over the code, thus increasing the number of useful mutants selected in their experiments.

Researchers making use of metaheuristic search algorithms have observed the need to perform an empirical analysis of the involved parameters, given the impact that the configuration can have on their outcome. Finding the best parameter values can be done by means of different methods, including

experimental design (Coy et al., 2001) and iterated local search (Hutter et al., 2009). Racing approaches, and especially the one implemented by the *irace* package (López-Ibáñez et al., 2016), constitute a well-established option that has been applied for similar purposes. As an example, Lima and Vergilio (2017) recently applied *irace* to avoid the manual configuration of a multi-objective algorithm; they used this algorithm to compare different strategies for the selection of second order mutants on the basis of different objectives, such as the number of generated mutants or their ability to reproduce more complex faults. They followed a hyper-heuristic approach, which is another alternative to address the uncertainty regarding different algorithm choices in search-based software testing (Balera and de Santiago Júnior, 2019).

However, the selection of mutants for the improvement of the test suite with EMT remained pending. Domínguez-Jiménez et al. (2011) initially performed an ad-hoc and manual process in order to obtain a value for the classical parameters shown in Section 2.6; in their experiments, they considered a manageable set of 3 or 4 values to configure each parameter in three WS-BPEL compositions. Their conclusions on the optimal values were based on the total times required to reach the whole set of live mutants. In our experiments, and thanks to the automated configuration approach, we analyze the best values for an extended set of parameters using a larger set of case studies. Additionally, we use a more accurate quality measure of the performance of EMT, which simulates the actual improvement that would be achieved with the selected mutants. Regarding the alternative design choices assessed in this paper, Gutiérrez-Madroñal et al. (2019) applied the guided mutation operator and the pure random individual generator in their experiments. They set the sampling rate for the guided operator to 90%, but no configuration process for this parameter was apparently carried out. In their experiments applying these refinements to EPL queries, they found that the changes improved the original version of EMT in most cases. The benefits associated with the guided approach hold in our study, but the pure random generator appears as detrimental to EMT's effectiveness.

The *context* has been remarked as a key factor within this study. In particular, the collection of operators is the aspect that has a greater impact on EMT's performance. Mutation testing has been successfully applied to many programming languages and different domains (Papadakis et al., 2019). However, experimental studies have revealed that each set of operators is characterized by some specific features. Remarkably, traditional operators are known to produce a large number of mutants, while class operators produce far fewer mutants but a higher percentage of equivalent ones (between 28% and 45% in the studies by Delgado-Pérez et al. (2017) and Segura et al. (2011), respectively). This means that class-level operators focus on features that are less common than those changed by traditional operators (e.g., relational operators). In our study, the number of class-level operators generating at least one mutant in the case studies ranged from 6 to 18, and the number of mutants from 137 to 1,146. Regarding WS-BPEL, Estero-Botaro et al. (2010) defined 26 operators for this programming language and analyzed them considering the number

of stillborn and equivalent mutants. Boubeta-Puig et al. (2011) carried out a comparison between these WS-BPEL operators and those for other languages. Based on that study, Papadakis et al. (2019) pointed out that many of those operators are different from traditional ones due to the particular constructs in BPEL compositions. The particularities of each context also affect the design options studied in this paper. With regard to equivalent mutant detection and penalization, TCE has shown a varied performance depending on whether it is applied to traditional C mutants (around 30% of all equivalent mutants are detected on average in the study by Papadakis et al. (2015)), class-level C++ mutants (13% in the study by Delgado-Pérez and Segura (2019)) or memory mutants (around 5% in the study by Wu et al. (2017)). As discussed in this paper, the results of the pure random individual generator may also be directly related to the set of operators, given that each operator focuses on particular features of the object-oriented paradigm (Delgado-Pérez et al., 2019).

8 Conclusions and Future Work

This paper presents an approach for the automated configuration of Evolutionary Mutation Testing, overcoming the impediments to its manual configuration. The best configuration found with the application of iterated racing was able to improve over the default configuration in all the studied programs. These results support the need to perform parameter tuning in different contexts and when new design choices come into play. The paper also provides an in-depth evaluation of a range of advanced design options that can help better tailor EMT to the particular features of each domain.

The fact that two of the alternative options assessed in the experiments were enabled in the best configuration found lead us to believe that there is still room for improvement in this search-based approach. For instance, the preference shown for the guided mutation operator suggests that learning from the past is a good idea. As such, the integration of data extracted by machine-learning techniques might bring further benefits. In this paper, we have evaluated the benefits of applying parameter tuning in terms of the size of the test suites. In the future, it would be interesting to assess the overlap between the test cases that would be achieved with each configuration and whether there is a substantial difference in the quality of these test suites (that would require to define a quantitative measure of test suite quality). Another line to explore in the future is the application of this approach to other contexts in order to observe if there are any changes in the best parameter settings. This could help us gain insight into the connection between the particular features of each context and the values acquired by the parameters in the best configuration found in them.

References

Balera JM, de Santiago Júnior VA (2019) A systematic mapping addressing hyper-heuristics within search-based software testing. Information and Software Technology 114:176 – 189, URL https://doi.org/10.1016/j.infsof.2019.06.012

Banzi AS, Nobre T, Pinheiro GB, Árias JCG, Pozo A, Vergilio SR (2012) Selecting mutation operators with a multiobjective approach. Expert Systems with Applications 39(15):12131–12142, URL http://dx.doi.org/10.1016/j.eswa.2012.04.041

Barbosa EF, Maldonado JC, Vincenzi AMR (2001) Toward the determination of sufficient mutant operators for C. Software Testing, Verification and Reliability 11(2):113–136, URL http://dx.doi.org/10.1002/stvr.226

Boubeta-Puig J, García-Domínguez A, Medina-Bulo I (2011) 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, p 398–407, URL http://dx.doi.org/10.1109/ICSTW.2011.52, print ISBN: 978-1-4577-0019-4

Budd TA (1980) Mutation analysis of program test data. PhD thesis, Yale University

Coy S, Golden B, Runger G, Wasil E (2001) Using experimental design to find effective parameter settings for heuristics. Journal of Heuristics 7(1):77–97, URL https://doi.org/10.1023/A:1026569813391

Delgado-Pérez P, Medina-Bulo I (2018) Search-based mutant selection for efficient test suite improvement: Evaluation and results. Information and Software Technology 104:130–143, URL https://doi.org/10.1016/j.infsof.2018.07.011

Delgado-Pérez P, Segura S (2019) Study of trivial compiler equivalence on C++ object-oriented mutation operators. In: Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, ACM, New York, NY, USA, SAC '19, pp 2224–2230, URL http://dx.doi.org/10.1145/3297280.3297499

Delgado-Pérez P, Medina-Bulo I, Núñez M (2017) Using evolutionary mutation testing to improve the quality of test suites. In: 2017 IEEE Congress on Evolutionary Computation (CEC), pp 596–603, URL https://doi.org/10.1109/CEC.2017.7969365

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

Delgado-Pérez P, Rose LM, Medina-Bulo I (2019) Coverage-based quality metric of mutation operators for test suite improvement. Software Quality Journal 27(2):823–859, URL https://doi.org/10.1007/s11219-018-9425-7

Dolphin (2019) Dolphin. https://www.kde.org/applications/system/dolphin

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

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

Eiben AE, Smith JE (2015) Introduction to Evolutionary Computing, 2nd edn. Springer Publishing Company, Incorporated

Estero-Botaro A, Palomo-Lozano F, Medina-Bulo I (2010) Quantitative evaluation of mutation operators for WS-BPEL compositions. In: Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW), 2010, pp 142–150, URL http://dx.doi.org/10.1109/ICSTW.2010.36

Goldberg DE (1989) Genetic Algorithms in Search, Optimization and Machine Learning, 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA

Gutiérrez-Madroñal L, García-Domínguez A, Medina-Bulo I (2019) Evolutionary mutation testing for IoT with recorded and generated events. Software: Practice and Experience 49(4):640–672, URL https://dx.doi.org/10.1002/spe.2629

Hutter F, Hoos HH, Leyton-Brown K, Stützle T (2009) ParamILS: An automatic algorithm configuration framework. Journal of Artificial Intelligence Research 36(1):267–306

Jia Y, Harman M (2008) Constructing subtle faults using higher order mutation testing. In: Eighth IEEE International Working Conference on Source Code Analysis and Manipulation, 2008, pp 249–258, URL http://dx.doi.org/10.1109/SCAM.2008.36

Khari M, Kumar P (2019) An extensive evaluation of search-based software testing: a review. Soft Computing 23:1933–1946, URL https://doi.org/10.1007/s00500-017-2906-y

Lima JAP, Vergilio SR (2017) 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, ACM, New York, NY, USA, SAST, pp 6:1–6:10, URL http://doi.org/10.1145/3128473.3128479

López-Ibáñez M, Dubois-Lacoste J, Cáceres LP, Birattari M, Stützle T (2016) The irace package: Iterated racing for automatic algorithm configuration. Operations Research Perspectives 3:43–58, URL https://doi.org/10.1016/j.orp.2016.09.002

Madeyski L, Orzeszyna W, Torkar R, Jozala M (2014) Overcoming the equivalent mutant problem: A systematic literature review and a comparative

experiment of second order mutation. IEEE Transactions on Software Engineering 40(1):23–42, URL http://dx.doi.org/10.1109/TSE.2013.44
Matrix TCL Pro (2019) TCL Pro. http://www.techsoftpl.com/matrix/download.php
McMinn P (2004) Search-based software test data generation: A survey. Software Testing, Verification and Reliability 14(2):105–156, URL https://doi.org/10.1002/stvr.v14:2
MuParser (2019) MuParser. https://beltoforion.de/article.php?a=muparser
MySQL Server (2019) MySQL Server. https://github.com/mysql/mysql-server
de Oliveira AAL, Camilo-Junior CG, Vincenzi AMR (2013) A coevolutionary algorithm to automatic test case selection and mutant in mutation testing. In: IEEE Congress on Evolutionary Computation, 2013, pp 829–836, URL http://dx.doi.org/10.1109/CEC.2013.6557654
Papadakis M, Malevris N (2011) Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. Software Quality Journal 19(4):691–723, URL https://doi.org/10.1007/s11219-011-9142-y
Papadakis M, Jia Y, Harman M, Le Traon Y (2015) 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, IEEE Press, Piscataway, NJ, USA, ICSE'15, pp 936–946, URL http://dx.doi.org/10.1109/ICSE.2015.103
Papadakis M, Kintis M, Zhang J, Jia Y, Traon YL, Harman M (2019) Chapter six - mutation testing advances: An analysis and survey. Advances in Computers, vol 112, Elsevier, pp 275–378, URL https://doi.org/10.1016/bs.adcom.2018.03.015
Qingfu Zhang, Jianyong Sun, Tsang E (2005) An evolutionary algorithm with guided mutation for the maximum clique problem. IEEE Transactions on Evolutionary Computation 9(2):192–200, URL https://dx.doi.org/10.1109/TEVC.2004.840835
QtDOM (2019) QtDOM. https://github.com/qtproject/qtbase/tree/dev/src/xml/dom
Rodrigues DS, Delamaro ME, Corrêa CG, Nunes FLS (2018) Using genetic algorithms in test data generation: A critical systematic mapping. ACM Computing Surveys 51(2):41:1–41:23, URL http://doi.org/10.1145/3182659
Schwarz B, Schuler D, Zeller A (2011) Breeding high-impact mutations. In: Proceedings of the 4th IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2011, pp 382–387, URL http://dx.doi.org/10.1109/ICSTW.2011.56
Segura S, Hierons RM, Benavides D, Ruiz-Cortés A (2011) Mutation testing on an object-oriented framework: An experience report. Information and Software Technology 53(10):1124–1136, URL http://dx.doi.org/10.1016/j.infsof.2011.03.006, special Section on Mutation Testing

Silva RA, do Rocio Senger de Souza S, de Souza PSL (2016) A systematic review on search based mutation testing. Information and Software Technology URL http://dx.doi.org/10.1016/j.infsof.2016.01.017

Tinyxml2 (2019) Tinyxml2. https://github.com/leethomason/tinyxml2

Wu F, Nanavati J, Harman M, Jia Y, Krinke J (2017) Memory mutation testing. Information and Software Technology 81:97–111, URL https://doi.org/10.1016/j.infsof.2016.03.002

XmlRPC++ (2019) XmlRPC++. http://xmlrpcpp.sourceforge.net/