GiGAn: Evolutionary Mutation Testing for C++ Object-Oriented Systems

Pedro Delgado-Pérez
Escuela Superior Ingeniería
University of Cádiz, Spain
pedro.delgado@uca.es

Inmaculada Medina-Bulo
Escuela Superior Ingeniería
University of Cádiz, Spain
inmaculada.medina@uca.es

Sergio Segura
Escuela Superior Ingeniería
University of Sevilla, Spain
sergiosegura@us.es

Antonio García-Domínguez
School of Engineering and Applied Science
Aston University, UK
a.garcia-dominguez@aston.ac.uk

Juan José Domínguez-Jiménez
Escuela Superior Ingeniería
University of Cádiz, Spain
juanjose.dominguez@uca.es

ABSTRACT

The reduction of the expenses of mutation testing should be based on well-studied cost reduction techniques to avoid biased results. Evolutionary Mutation Testing (EMT) aims at generating a reduced set of mutants by means of an evolutionary algorithm, which searches for potentially equivalent and difficult to kill mutants to help improve the test suite. However, there is little evidence of its applicability to other contexts beyond WS-BPEL compositions. This study explores its performance when applied to C++ object-oriented programs thanks to a newly developed system, *GiGAn*. The conducted experiments reveal that EMT shows stable behavior in all the case studies, where the best results are obtained when a low percentage of the mutants is generated. They also support previous studies of EMT when compared to random mutant selection, reinforcing its use for the goal of improving the fault detection capability of the test suite.

CCS Concepts

•**Software and its engineering** → **Software testing and debugging;** *Search-based software engineering;*

Keywords

mutation testing; evolutionary computation; genetic algorithm; object orientation; C++.

1. INTRODUCTION

A test suite is developed in order to reveal possible faults in a system under test. Mutation testing provides a means for measuring its ability to detect coding errors. In this technique, new versions of the code with injected faults (*mutants*) are used to stress the test suite. The outputs after their execution against the test suite should be different from the expected ones in all the cases (mutants should be *killed*). Undetected or *alive* mutants reflect weaknesses in the test suite. The goals when a system undergoes a mutation testing process are: (1) evaluate to what extent the test suite is able to identify faults and (2) improve the test suite with new test cases based on the inspection of alive mutants.

Several techniques have been suggested in the past to ease the cost of applying mutation testing [11]. While most of them are useful for (1), *Evolutionary Mutation Testing* (EMT) [8] was recently presented with a focus on (2). EMT proposes the generation of a subset of mutants through an evolutionary algorithm. That subset should contain a high proportion of the mutants that can guide on the creation of new test cases (called *strong mutants*): *potentially equivalent* (currently not detected) and *difficult to kill* mutants (detected by one test case only killing that mutant).

EMT was successfully put into practice regarding WS-BPEL compositions [8], but it has not been assessed in other domains after that. As a result, its applicability to other contexts is an open question. This paper analyzes the performance of this technique in object-oriented (OO) systems. To that end, we developed *GiGAn* to make use of a genetic algorithm (GA) in connection with *MuCPP* [6], a C++ mutation tool implementing operators at the class level. The evaluation in this paper replicates existing studies but in an OO context, revealing that the technique effectively outperforms the random selection of mutants, as a smaller percentage of mutants is needed to achieve the same percentage of strong mutants. Another interesting finding of the performance of EMT is that it only varies slightly among case studies and percentage of mutants produced.

The paper is structured as follows. Section 2 describes the fundamental aspects of EMT. Section 3 explores the details of its application to C++ OO applications through *GiGAn*. Section 4 is about the empirical evaluation carried out and discussion about results. The last section presents conclusion and future research lines.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SAC 2017,April 03-07, 2017, Marrakech, Morocco

Copyright 2017 ACM 978-1-4503-4486-9/17/04…$15.00

http://dx.doi.org/10.1145/3019612.3019828

2. EVOLUTIONARY MUTATION TESTING

2.1 Definition

Evolutionary Mutation Testing [8] proposes the use of an evolutionary algorithm to produce only a subset of the full set of mutants in order to reduce the cost. This algorithm works under the assumption that there are some mutants with greater potential than others in guiding the tester to the design of new test cases with high fault detection capability, which are referred to as strong mutants. The generation of strong mutants is favored by the evolutionary algorithm search, thereby reducing the number of mutants while retaining the power to refine the test suite.

These two kinds of mutants are regarded as strong mutants:

- *Potentially equivalent:* mutants not detected by the initial test suite. These mutants either lead to the generation of new test cases or result in equivalent mutants once they are inspected.
- *Difficult to kill:* mutants detected by only one test case that detects no other mutants. This test case might only be created by inspecting this mutant.

Ideally, all potentially equivalent mutants help improve the test suite with new test cases. However, some of those mutants may turn out to be equivalent as this is an undecidable problem and they cannot be discarded automatically.

2.2 Fitness function

Although the algorithm favors the generation of strong mutants, each of the mutants receives a fitness. The fitness of a mutant decreases as a) the number of test cases detecting the mutant increases and b) the number of mutants killed by those test cases increases. Therefore, to calculate the fitness function every mutant has to be executed against every test case. Equation 1 shows how the fitness of mutant I is computed with respect to test suite S, where M is the number of mutants, T is the number of test cases in S and m_{ij} is 1 when mutant i is detected by test case j, and 0 otherwise.

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

According to this fitness function, if the mutant I is:

- Potentially equivalent, it receives the maximum value ($M \times T$) because $m_{Ij} = 0$ for all j.
- Difficult to kill, it receives a fitness of $M \times T - 1$ because $m_{Ij} = 0$ for all j except for one (z), which kills no other mutants ($m_{Iz} = 1$ and $\sum_{i=1}^{M} m_{iz} = 1$).
- Weak, it receives a fitness lower than $M \times T - 1$. The more test cases kill I, the lower the fitness; also, the more mutants those test cases kill, the lower the fitness.

As a final remark, invalid mutants (i.e., they cannot be executed) neither are assigned a fitness nor affect the fitness computation of the rest of valid mutants.

```
int f(int x, int y){
  if(x > 0){
    if(y > 1){
      return y;
    }
  }
  return x;
}
```

(a) Original

```
int f(int x, int y){
  if(x > 0){
    if(y < 1){
      return y;
    }
  }
  return x;
}
```

(b) Mutant

Operators	Attributes
1.Relational operator replacement	<=, >=, <, >, ==, !=
2.Arithmetic operator replacement	+, -, *, /, %
...	...

(c) List of operators

Figure 1: Information for mutant encoding: a) original code, b) mutation inserted (second appearance of $>$ by $<$) and c) predefined positions of operators and their attributes

2.3 Individuals

In EMT, the mutants are the individuals for the GA and, as such, they must be uniquely identified. To this end, each mutant is encoded with 3 fields: *operator* (identifier of the mutation operator), *location* (order in the code of the mutants of an operator) and *attribute* (variant inserted in a location). As an illustration, consider the information in Figure 1. The mutant depicted in the figure is identified as:

- **Operator = 1**: The first operator is applied.
- **Location = 2**: The second relational operator in the code is mutated.
- **Attribute = 3**: The relational operator is changed by the third variant in the predefined set of attributes.

2.4 Genetic algorithm

The GA produces several generations of mutants during its execution, which is directed by the search of strong mutants through the fitness function. The algorithm performs two main steps in each generation:

1. Generation of mutants:
   - *First generation*: mutants are generated randomly.
   - *Next generations*: mutants are generated both randomly and with reproductive operators.
2. Execution of the mutants generated. The fitness assigned to the mutants generated is computed with respect to:
   - *First generation*: the mutants in that generation.
   - *Next generations*: all the mutants generated so far. This is achieved by storing a second population with the mutants created in previous generations, which helps the fitness function to produce better estimations (see [8] for an example).

Regarding the reproductive operators, the GA can apply *mutation operators* and *crossover operators* to individuals

from the previous generation to create new ones (the *roulette wheel method* is used to select the mutants):

- *Mutation operators:* One of the three fields (*operator*, *location* or *attribute*) is mutated.
- *Crossover operators:* Starting from two parents (*operator*$_1$, *location*$_1$, *attribute*$_1$) and (*operator*$_2$, *location*$_2$, *attribute*$_2$), one of these crossover points is selected:
  - *Point 1:* generates (*operator*$_1$, *location*$_2$, *attribute*$_2$) and (*operator*$_2$, *location*$_1$, *attribute*$_1$).
  - *Point 2:* generates (*operator*$_1$, *location*$_1$, *attribute*$_2$) and (*operator*$_2$, *location*$_2$, *attribute*$_1$).

We should note that a process of normalization of the fields avoids invalid representations of mutants. Further details on the essentials of the technique and computation overheads are included in the paper by Domínguez-Jiménez et al [8].

3. GIGAN: EMT IN OO SYSTEMS FOR C++

3.1 Class mutation operators for C++

Most of the studies in the literature covering issues related to the cost of mutation testing have been carried out with traditional operators [2] (used for structured programming). However, it remains unclear whether the same benefits apply to operators at the class level. Studies on class operators [15] consistently show that they exhibit different properties when compared to traditional operators: they generate fewer mutants but a higher equivalence percentage. In particular, EMT has only been applied to WS-BPEL in the past. However, it is unknown to what extent the reduction achieved in that study holds in other contexts.

The GA described in Section 2.4 is implemented in *GAmera* [7], and this tool makes use of *MuBPEL* to analyze, generate and execute mutants for these compositions. Recently, the mutation tool *MuCPP* [6] has been developed including a set of class operators for C++ programs. In order to reuse the same GA, we developed a new tool, *GiGAn*, to connect the algorithm in *GAmera* and the mutation tool *MuCPP*. In the experiments in this paper, the same list of 31 class operators defined by Delgado-Pérez et. al [6] is applied.

3.2 GiGAn

Figure 2 displays how *GiGAn* connects *MuCPP* and *GAmera* to apply EMT to C++ OO systems. As it can be seen, *GiGAn* acts as a bridge between both tools, translating the commands that each of the tools uses and mapping mutant identifiers so that *MuCPP* and *GAmera* can work together. *GiGAn* presents two main changes regarding the original description of the technique, which can impact the results:

- **Attribute:** Some class operators in *MuCPP* [6] produce multiple mutations from a single location. The available mutations depend on the context, so the range of the *attribute* field is unknown in advance. As a result, the tool treats each of these mutations as its own location and all mutants have *attribute* = 1 as a consequence. Thus, we limit reproductive operators to mutation of *operator* and *location* fields and *point 1* crossover (see Section 2.4), as the rest of operators would result in the same mutant being created.
- **Mutants in different source files:** *MuCPP* allows several source files of a project to be analyzed in the same execution. As a result, mutants from different files can be generated when the location field is changed to produce new individuals from previous ones (notice that in *GiGAn* the *location* of each mutation in each file is known because files are sorted beforehand). Even though classes in a project often use a similar design pattern, it is possible that the behavior of a mutation operator varies for different classes, especially when they belong to different source files.

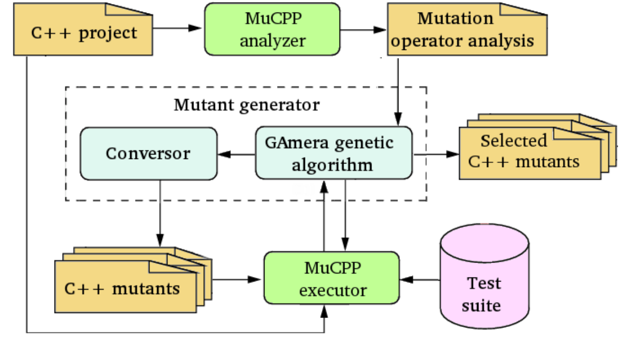


Figure 2: GiGAn diagram

4. EMPIRICAL EVALUATION

4.1 Research Questions

This empirical study investigates the relative effectiveness of the use of EMT in real C++ programs using object orientation. In particular, this experiment aims to know 1) how many mutants EMT needs to generate to find different percentages of strong mutants and 2) whether this technique produces better results when compared to random mutant selection. Thus, these are the research questions to answer:

RQ1: How does EMT behave when searching different percentages of strong mutants in C++ OO systems?

RQ2: Does EMT outperform random mutant selection?

4.2 Case Studies

This study analyzes four open-source programs, which were chosen because they make use of C++ OO facilities and are distributed with a test suite (see information in Table 1). *MuCPP* produces a different number of mutants for these applications. Thanks to a previous execution of all the mutants, we know how many of them are strong with the current test suite (used as a ground truth to compute our results).

4.3 Experiment design

EMT needs to be configured with several parameters:

- *Population size*: individuals in each generation. It is a percentage of the number of mutants in each program.
- *Individuals generated randomly and by reproductive operators*: since all the mutants in a generation are produced through these two ways (see Section 2.4), the sum of both percentages has to be 100%.
- *Mutation and crossover probability*: the probability that mutation or crossover operators are used when a mutant is generated through reproductive operators. As in the previous item, they have to sum 100%.

Table 1: Information about the subject programs

	TCL	DPH	TXM	DOM	Total
Classes	9	13	20	11	53
Lines of code	3,228	3,667	2,620	2,117	11,632
Test cases	17	61	57	46	181
Total mutants	137	219	614	1,146	2,116
Valid mutants	135	208	433	681	1,457
% Strong mut.	33.3%	49.5%	36.7%	51.1%	45.0%

TCL=Matrix TCL, DPH=Dolphin, TXM=Tinyxml2, DOM=QtDOM

The values selected for these parameters (see Table 2) are the ones found as optimal for the execution of this algorithm in the experiments where the technique was presented [8].

In these experiments, we want to know the ability of EMT to find strong mutants. Thus, all mutants were generated and executed against all test cases in a previous execution to maintain a record of strong mutants in the analyzed programs. We established several stopping conditions for the algorithm: finding 30%, 45%, 60%, 75% and 90% of the set of strong mutants. Then EMT was run 30 times with different seeds for each of the five conditions. Therefore, the data shown are obtained from the results of these 30 executions.

In order to answer RQ2, we make use of a random strategy where mutants are selected one by one until reaching the stopping condition. Again, this random technique was executed 30 times and several statistics were calculated.

Table 2: Genetic algorithm configuration

Parameter configuration	Value
Population size	5%
Individuals generated randomly	10%
Individuals generated by reproductive operators	90%
- *Mutation probability*	30%
- *Crossover probability*	70%

4.4 Results and discussion

Table 3 contains, individually for each program, the average, median, minimum, maximum and standard deviation of the results of the 30 executions for each of the 5 stopping conditions. Thus, the numbers shown in this table represent the percentage of mutants that EMT needs to generate before finding 30%, 45%, 60%, 75% and 90% of the set of strong mutants in these applications. We can observe that the percentage of necessary mutants increases as the stopping condition becomes more demanding in all the programs.

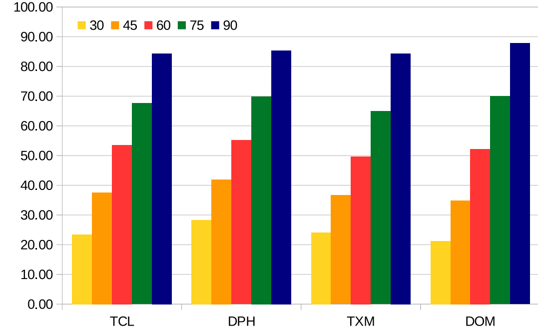


Figure 3: Average percentage of mutants generated in the programs to reach the stopping conditions.

Table 3: Percentage of the number of mutants generated to achieve 30%, 45%, 60%, 75% and 90% of the strong mutants (SD: standard deviation)

Program	*30%*	*45%*	*60%*	*75%*	*90%*
TCL					
Average	23.45	37.59	53.55	67.59	84.33
Median	24.08	39.05	54.74	67.52	83.94
Min.	13.13	25.54	40.14	51.09	70.07
Max.	37.22	51.09	65.69	79.56	92.70
SD	5.44	6.67	6.10	6.98	5.21
DPH					
Average	28.35	41.94	55.19	69.87	85.35
Median	28.54	42.23	54.79	70.09	85.38
Min.	24.65	38.35	50.68	62.55	78.99
Max.	33.33	47.03	59.81	76.71	90.41
SD	2.11	2.28	2.10	3.57	2.67
TXM					
Average	24.09	36.62	49.74	64.91	84.32
Median	24.18	36.07	49.67	64.74	84.12
Min.	20.52	31.92	44.78	60.58	77.85
Max.	27.85	41.04	56.35	71.49	89.73
SD	1.61	2.34	3.05	2.59	3.34
DOM					
Average	21.20	34.86	52.21	69.96	87.84
Median	21.16	34.86	52.31	70.15	88.09
Min.	19.02	32.28	46.59	66.05	83.33
Max.	23.03	37.26	57.06	73.38	90.13
SD	1.01	1.26	2.39	1.98	1.60

Figure 3 focuses on the average, allowing us to know the tendency of this increase in each application. Given that the stopping conditions have been selected in 15% increments, this graphic reflects that the upward tendency is quite stable, not only between conditions but also among applications. Still, there is often a small increment in the percentage of mutants generated as the stopping condition increases. Taking *TXM* to illustrate this fact, on average EMT needs to produce 12.5% more mutants to find 45% of the strong mutants than to find 30%. However, this difference increases when considering the conditions 45%-60% (13.1%) 60%-75% (15.2%) and 75%-90% (19.4%). The standard deviation does not follow a pattern and is quite low, except for *TCL* where it might be affected by the few mutants in this program.

Figure 4 shows the average results of the random strategy focused on the two more demanding stopping conditions. In the light of the results, EMT produces a better outcome than the random mutant selection in all cases. Similar re-

sults are obtained for the other three stopping conditions and the rest of statistics, except for the standard deviation (where we can observe varying results). We run a statistical test to know about the significance of the results. We used *STATService*[1], which selects an appropriate statistical test depending on the data (smart statistical test). The results (collected in Table 4) lead us to accept that the median percentage of mutants that EMT needs to generate to find a subset of strong mutants is significantly lower than with random selection within a 99.9% confidence interval.

In order to evaluate the effect size, we also computed the non-parametric *Vargha and Delaney's* A_{12} statistic. In all cases, the difference between the results for both algorithms can be described as large, especially in *TXM* where the best results were achieved (with a difference over 10% for the 75% stopping condition) Still, the gap between both strategies in the experiments by Domínguez-Jiménez et al. [8] is greater than in this study in the better case (16% on average for the more complex WS-BPEL composition when trying to find all strong mutants); each of the class operators addresses different OO features in general, which may limit the benefits of using this GA. Overall, EMT should work better as the probability of randomly selecting a strong mutant lowers. Given that finding a strong mutant in *TXM* is harder than in *DOM* (36.7% and 51.1% of strong mutants respectively), the good results in *TXM* when compared to *DOM* might be related to this fact, but this merits further investigation.

RQ1: *How does EMT behave when searching different percentages of strong mutants in C++ OO systems?* The GA behaves in a stable way for all the tested programs (the relation between mutants generated and strong mutants does not vary much with the program). Additionally, the proportion of strong mutants found by EMT slightly decreases with the number of mutants generated in general.

RQ2: *Does EMT outperform random mutant selection?* Yes. EMT yields better results than the random strategy with high confidence. The difference between both selection strategies is on average 6.17% and 3.80% to find 75% and 90% of strong mutants respectively for the analyzed programs.

4.5 Threats to validity

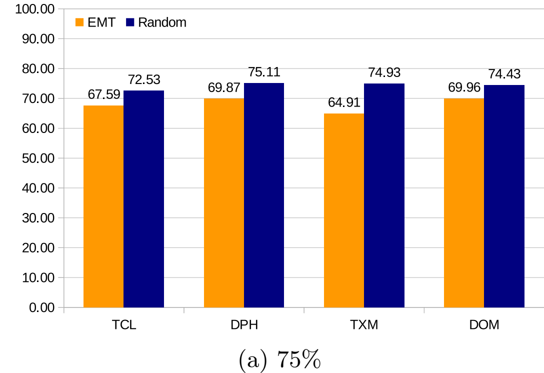
Representativeness of the programs under study presents a threat to the validity of the results. To counter this threat, we have selected four applications in which (a) different mutation operators were applied and (b) those operators generated a different number of mutants. Moreover, the number of strong mutants varies among those programs.

Table 4: Results of the smart and *Vargha and Delaney's* A_{12} statistical tests

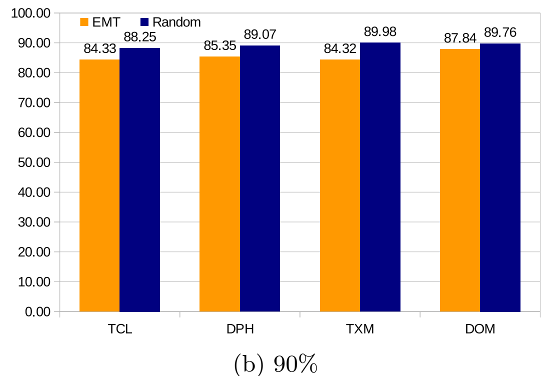
	75%		90%	
Program	p-value	A_{12}	p-value	A_{12}
TCL	2.26×10^{-03}	0.711	1.24×10^{-03}	0.734
DPH	4.55×10^{-07}	0.848	2.72×10^{-06}	0.829
TXM	7.14×10^{-21}	0.996	1.65×10^{-06}	0.937
DOM	4.31×10^{-12}	0.962	1.71×10^{-05}	0.816

[1]http://moses.us.es/statservice

Figure 4: Average percentage of the number of mutants generated with EMT and random selection to achieve 75% (a) and 90% (b) of the strong mutants.



(a) 75%



(b) 90%

The GA can show a different behavior depending on the configuration. Since the best configuration for a particular program is unknown in advance for the user, we have set the same parameters for all the programs. Namely, we have used the configuration found as optimal in previous studies, but other parameters could yield a worse or better performance. EMT selects and generates new individuals on a random basis. As such, we have executed the technique 30 times in order to avoid biased results because of a single execution.

5. RELATED WORK

There exist several techniques to alleviate the cost of mutation testing by reducing the number of mutants generated [11], such as *mutant sampling* [4] (selects a subset of the mutants randomly), *selective mutation* [2] (selects a subset of the mutation operators) or *high order mutation* (HOM) [10] (combines more than a single fault into a mutant). EMT [8] was proposed recently for test suite improvement and assessed with 3 WS-BPEL compositions. The technique was later extended to generate HOMs [3]. In this work, EMT has been shown to be better than mutant sampling at finding strong mutants in 4 different C++ OO programs.

Silva et al. [16] surveyed the studies applying search based techniques in the scope of mutation testing. However, most of these works are devoted to test data generation, even for OO software [9], and only a few to mutant generation

(where EMT is classified). Adamopoulos et al. [1] were the first in using a GA for the co-evolution of mutant and test suite population, where difficult to kill mutants are favored and equivalent mutants are penalized (unlike EMT), while Oliveira et al. [5] also studied this approach but describing a new representation with new genetic operators. Other studies in the literature have focused on using a GA to generate interesting HOMs [10, 12]. Finally, Schwarz et al. [14] leveraged a GA to find mutations not detected by the test suite, which have a high impact and are also spread throughout the tested code.

6. CONCLUSIONS

The experiments in an OO context using *GiGAn* confirm the promising results yielded by EMT in previous research, thereby supporting this cost reduction technique as a useful mechanism for the selection of mutants with the goal of improving the test suite. The evaluation reveals that there is high stability among the results for the tested programs, and little variation in the percentage of strong mutants found as the number of mutants increases (best results with low percentages of mutants generated). Additionally, this study has shown EMT to be different from random testing, with better results in all case studies with high confidence. The gap between both strategies was however greater in the experiments with WS-BPEL.

Future work can be divided into two different lines. Firstly, we would like to simulate a real process where the test suite is improved with new test cases. Instead of stopping when finding a percentage of strong mutants, in that experiment the algorithm would stop when a percentage of new test cases is reached. In this way, we could evaluate how EMT really help us improve the test suite. Secondly, studying the impact of mutations in the code coverage [12] or how to integrate trivial compiler equivalence [13] can help isolate equivalent mutants. This information could assist in lowering the probability of selecting equivalent mutants.

7. ACKNOWLEDGMENTS

This paper was partially funded by the research scholarship PU-EPIF-FPI-PPI-BC 2012-037 (University of Cádiz), by the Spanish Government projects DArDOS (TIN2015-65845-C3-3-R), CICYT BELI (TIN2015-70560-R) and the Excellence Network SEBASENet (TIN2015-71841-REDT), and Andalusian Government project COPAS (P12-TIC-1867).

8. REFERENCES

[1] K. Adamopoulos, M. Harman, and R. M. Hierons. How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference, 2004*, pages 1338–1349.

[2] 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*, 11(2):113–136, 2001.

[3] E. Blanco-Muñoz, A. García-Domínguez, J. J. Domínguez-Jiménez, and I. Medina-Bulo. Towards higher-order mutant generation for WS-BPEL. In *Proceedings of the International Conference on e-Business (ICE-B), 2011*, pages 1–6.

[4] T. A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, 1980.

[5] 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*, pages 829–836.

[6] 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*, 2016.

[7] J. J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, and I. Medina-Bulo. GAmera: an automatic mutant generation system for WS-BPEL compositions. *Proceedings of the 7th IEEE European Conference on Web Services, 2009*, pages 97–106.

[8] J. J. Domínguez-Jiménez, A. Estero-Botaro, A. García-Domínguez, and I. Medina-Bulo. Evolutionary mutation testing. *Information and Software Technology*, 53(10):1108–1123, Oct. 2011.

[9] 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, 2011* pages 416–419.

[10] 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*, pages 249–258.

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

[12] W. B. Langdon, M. Harman, and Y. Jia. Efficient multi-objective higher order mutation testing with genetic programming. *Journal of Systems and Software*, 83(12):2416 – 2430, 2010. TAIC PART 2009.

[13] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon. Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, pages 936–946.

[14] 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, 2011*, pages 382–387.

[15] S. Segura, R. M. Hierons, D. Benavides, and A. Ruiz-Cortés. Mutation testing on an object-oriented framework: An experience report. *Information and Software Technology*, 53(10):1124–1136, 2011. Special Section on Mutation Testing.

[16] 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.