Combining Evolutionary Mutation Testing with Random Selection

Lorena Gutiérrez-Madroñal
UCASE Research Group
University of Cadiz
Puerto Real, Spain
lorena.gutierrez@uca.es

Antonio García-Domínguez
SEA Research Group
Aston University
Birmingham, UK
a.garcia-dominguez@aston.ac.uk

Inmaculada Medina-Bulo
UCASE Research Group
University of Cadiz
Puerto Real, Spain
inmaculada.medina@uca.es

***Abstract*—Mutation testing is a well-known fault-based technique that has been applied to different domains as new technologies have appeared. Evolutionary Mutation Testing (EMT) finds mutants that are useful to produce new test cases. It uses evolutionary algorithms to reduce the number of mutants that are generated, keeping as many difficult to kill and stubborn mutants (strong mutants) as possible in the reduced set. Given the popularity of real-time systems, the MuEPL mutation system was developed for the Esper Event Processing Language (EPL), a query language aimed at the Internet of Things (IoT). In past work, EMT was integrated into MuEPL, and it reduced the cost of finding strong mutants in some EPL queries but not in others. This study takes a step forward by proposing and evaluating two metaheuristics for EMT that combine EMT and random selection: one which bootstraps the hall of fame with a random subset (Bootstrapped EMT), and one which falls back to random selection after a certain point (Inverse EMT). While BEMT is shown to outperform IEMT in most cases, BEMT has not managed to outperform EMT. An additional experiment studies the impact of low-quality mutation operators in the relative performance of BEMT, IEMT and plain EMT. Results suggest that the MuEPL RRO operator was the reason for the poor performance of EMT in some scenarios.**

I. Introduction

The *Internet of Things* (IoT) is becoming increasingly important all over the world. Areas such as education, government, industry, or agriculture are examples where IoT is being widely applied. One of the main challenges for IoT systems is the amount of information they have to handle. Information that, in the majority of the cases, is crucial to make correct decisions.

In the literature, depending on the author and even the company, the challenges for testing IoT applications are categorised in different ways. We have identified the following challenges:

- Environment: IoT applications work in very dynamic environments that change in unpredictable ways.
- Complexity: IoT applications may have multiple real-time scenarios and their use cases require the coordination of many entities.
- Scalability: creating a test environment to assess functionality that can scale to many devices reliably is difficult.
- Availability: the majority of software to test IoT is proprietary, costly, and not publicly available.

If IoT systems are thoroughly tested before launch, we will be able to detect errors and prevent possible failures on the field. Heath [15] mentioned that testing real-time systems is one of the most complex and time-consuming activities. The author states that testing these systems typically consumes 50% of the overall development effort and budget, given that testing embedded systems is more difficult than testing conventional systems.

Mutation testing is a widely applied technique that consists of evaluating how well the test suite can tell apart the original program from one seeded with a common defect (a “mutant”). Mutation testing can simulate different errors that programmers make, and reduce the risk that applications make incorrect decisions. The multiple scenarios that mutation testing simulates represent one of the previous challenges: complexity.

While effective, mutation testing has one main drawback [24]: it can take a long time to run the large number of mutants produced for some programs against their test suites. Several well-studied cost reduction techniques designed to avoid biased results can be found. *Evolutionary Mutation Testing* (*EMT* [6]) consists of generating a reduced set of mutants by means of an evolutionary algorithm, which searches for stubborn and difficult to kill mutants to help improve the test suite. In [13], *EMT* has been applied to a specific language which processes the events that IoT systems have to monitor: the Event Processing Language (EPL) from EsperTech [9]. The results showed *EMT* to have different performance depending on the case study: sometimes *EMT* was better than random selection, but not always. This suggested the need to revise *EMT* to make it more consistent across systems.

Hypothesis: a higher-level strategy (metaheuristic) that combines *EMT* and random selection can be more consistent than plain *EMT*.

Contribution: the main contribution of this paper is testing the above hypothesis by evaluating two metaheuristics on top of *EMT*. *Bootstrapped EMT* (*BEMT*) expands the “hall of fame” that informs the fitness function through random

Paper partially funded by The Ministry of Economy and Competitiveness (Spain) and the FEDER Fund, under the National Program for Research, Development and Innovation, Societal Challenges Oriented, Project DArDOS TIN2015-65845-C3-3-R, and Project FAME RTI2018-093608-B-C33

selection before proper *EMT* is used. *Inverse EMT* (*IEMT*) "reverts" to random selection after *EMT* has run a certain proportion of all mutants. The paper repeats experiments in the prior work with these two metaheuristics and provides an evaluation of the results.

Structure: the rest of this paper is organised as follows. Section II presents the key background concepts. Section III describes the two metaheuristics, states the design of the experiments, and analyses the results obtained. Section IV reviews related work. The last section presents the conclusions and future lines of research.

II. Background

Before the two metaheuristics can be presented, some background knowledge on mutation testing and *EMT* is required. This section will present the key concepts from these fields needed to understand the rest of the work, and justify our decision to define and test these metaheuristics.

A. Mutation Testing

Mutation testing is a well-known fault-based technique that has been used to evaluate and improve the quality of test suites in software systems. This technique introduces simple syntax-level changes in the original program by applying *mutation operators*. Unlike other fault-based strategies that directly inject artificial faults into the program, the mutation method generates syntactic variations (*mutants*) of the original program by applying mutation operators. Each mutation operator represents "typical" programming errors that developers make.

Mutation testing has been applied to different domains as new technologies have appeared. Programming real-time systems requires languages with features beyond those of traditional general-purpose programming languages (GPLs). Most of the work on mutation testing is on traditional GPLs [19, 1, 23]. A survey by Jia in 2011 [18] showed many mutation systems, but none related to event-processing languages such as the Esper Event Processing Language (EPL).

In order to apply mutation testing to EPL, the MuEPL mutation system has been developed [12, 14]. Given that mutation testing is expensive, it was decided to reduce the cost of mutation testing by using well-studied cost reduction techniques to avoid biased results. *Evolutionary Mutation Testing* (*EMT* [6]) works by generating a reduced set of mutants through an evolutionary algorithm which searches for stubborn and difficult to kill mutants that help improve the test suite.

B. Evolutionary Mutation Testing

EMT [6] uses an evolutionary algorithm (implemented in the *GAmera* [7] tool) to guide mutant selection towards a subset that inspires the design of new test cases. The algorithm favours *strong mutants*, of which there are two kinds:

- *Stubborn* mutants survive the existing test suite. These need to be inspected to see if they are semantically equivalent to the original program. If they are not, they will lead to a new test case: either because the mutation was not reached, or because the test suite could not tell the mutant apart.
- *Difficult to kill* mutants are killed by a specific test case, which kills no other. They model a subtle, hard-to-find bug, and they are useful as places where the algorithm should focus.

1) Individual encoding: The generated mutants in the original program are the individuals for the evolutionary algorithm in *EMT*. *MuEPL* encodes each mutant with three fields (see Figure 1):

1) Operator: index within the operator list (1 is the first operator).
2) Location: index within the location list for that operator (1 is the first mutant of that operator).
3) Attribute: identifier of the variant of the operator for the location (1 is the first variant).

Operator	Location	Attribute

Figure 1. Individual encoding in GAmera for a program mutant

For the sake of clarity, consider the information in Example 1 and Table I. The mutant is identified by *MuEPL* as:

1) Operator = 2: the ROR operator is applied.
2) Location = 1: the first relational operator is mutated.
3) Attribute = 1: the relational operator is changed by the first variant in the predefined set of attributes.

Table I
PREDEFINED POSITIONS OF OPERATORS AND THEIR ATTRIBUTES

OPERATORS	ATTRIBUTES
Arithmetic op. replacement (AOR)	+, *, -, /
Relational op. replacement (ROR)	<, >, =, ! =, <=, >=

Example 1. Original and mutated queries (first appearance of a relational operator, >, is changed by the first variant, <).

```
select A as temp1, B as temp2 from
  pattern [every temp1.temperature > 400 ->
  temp2.temperature > 400]

select A as temp1, B as temp2 from
  pattern [every temp1.temperature < 400 ->
  temp2.temperature > 400]
```

The *MuEPL* encoding has a flaw: the ranges of the location and attribute fields change according to the operator. This is a problem when applying the *EMT* genetic operators (gene mutation and crossover), so *EMT* uses a *normalised* representation where the range of the Location and Attribute fields is the same regardless of the value of the Operator field:

- The normalised Location is an integer between 1 and the least common multiple of the number of locations for each operator. For instance, if we had two operators, one with 2 locations and one with 3, the range of Location would be between 1 and 6.

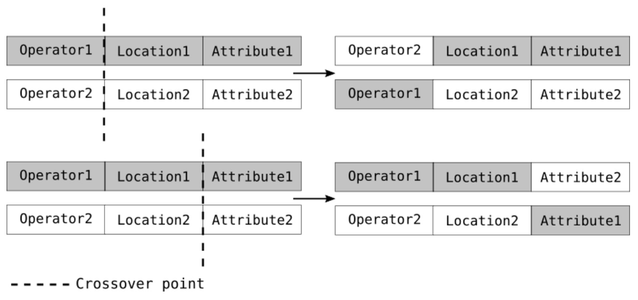

Figure 2. Crossover operator

When executing a mutant, a normalised Location L_i of the i-th operator is mapped back to the $\lceil (L_i \times l_i)/L \rceil$-th location, where l_i is the number of locations available for that i-th operator. In the example above, an individual for the first operator with a normalised Location of 4 would be mapped to the denormalised Location $\lceil (4 \times 2)/6 \rceil = 2$.

- The normalised Attribute is similar to the Location, being an integer between 1 and the least common multiple of the maximum values of the Attribute field for the various operators.

2) Main steps: The evolutionary algorithm starts by generating a random set of individuals as its first generation. Later generators are a combination of:

1) *Random generation*: a configuration parameter (percentage of new individuals, N) determines how many mutants will be randomly generated. Mutants are generated according to uniform distributions over the valid ranges of the three fields (Operator, Location, and Attribute), following the normalised ranges as mentioned above.
2) *Mutations and crossover*: the other new individuals are produced by modifying and combining pairs of parent individuals in the previous generation. The parents are picked through roulette selection [11], with a probability proportional to their fitness[1]. The parents will then go randomly through one or two of these processes, according to further probabilities set in the configuration:
   - *Mutation*: one of the three fields (operator, location or attribute) is perturbed, while honouring the range of valid values for that field. If the original value was α, the maximum value for that field is U, and the probability of mutation is p_m, the new value is computed as:

$$\begin{aligned}\beta = \quad & \alpha + \text{uniform\_random}(-10(1-p_m), \\ & 10(1-p_m)) \pmod{U}\end{aligned}$$

   - *Crossover*: a random crossover point is selected between the fields of the individuals. This may result in either the operator or the attribute field being swapped across the individuals (see Figure 2).

[1]Using a method with quick convergence such as roulette selection is intentional: the aim is to find the strong mutants as quickly as possible.

The algorithm assigns fitness values to the mutants, favouring strong mutants. The fitness function counts the test cases which kill the mutant, and the mutants killed so far by those test cases. The algorithm keeps track of all the mutants executed so far in a "hall of fame". The hall of fame maps the normalised (Operator, Location, Attribute) triple to its row of the execution matrix, which indicates for each test case whether the mutant survived it or not. This means that the fitness function is refined over time as more information about the program is revealed.

The fitness function is designed so that mutants which are killed by very common test cases will have low fitness. Stubborn mutants will have the maximum fitness, as they have not been killed by any test case. Difficult to kill mutants will have the second best value, since they were killed by only one test case which killed no other mutant.

C. Metaheuristics

EMT would be expected to generally behave better than random selection, but it is subject to the limitations of its fitness function. The fitness function is more nuanced for larger programs that have more mutants and require more test cases, as it was observed by Domínguez-Jiménez et al. [6], who noted that *EMT* worked best with their largest BPEL programs.

In our prior work in [13], it was observed that many mutants were killed on exactly the same test cases, producing only a few distinct fitness values. Another problem was that for some of the case studies, most of the mutants were actually strong, which went against the assumption in *EMT* that the strong mutants would be a small, hard-to-find subset.

Therefore, it was decided to evaluate two alternatives:

- Providing more information to *EMT* before it started. Rather than generating only the small initial population, a percentage of the mutants would be picked by random selection, further "bootstrapping" the hall of fame in the fitness function. This can be considered to be *Bootstrapped EMT* or *BEMT*.
- Falling back to random selection after a certain ratio of all individuals have been generated, assuming that the program has many strong mutants and that the genetic algorithm may be stuck in local optimal values of the fitness function. This approach can be considered to be *Inverse EMT* or *IEMT*.

III. Experimental process and results

Answering the hypothesis that plain *EMT* can be improved with a metaheuristic that integrates random selection can be broken down into two questions:

- **RQ1**: Which of *IEMT* and *BEMT* gives better results?
- **RQ2**: Is the best metaheuristic better than plain *EMT*?

Subsection III-A is dedicated to the experimental design. RQ1 is answered in Subsection III-B and RQ2 in Subsection III-C.

Table II
EVOLUTIONARY ALGORITHM PARAMETERS (TABLE 11 OF [6])

PARAMETER	VALUE
Population size	5%
Randomly generated individuals	10%
Crossover probability	70%
Mutation probability	30%

A. Experiment

EMT is a complex algorithm with many different parameters. In [13], the experiments use the values recommended in the initial study on *EMT* [6], which are listed in Table II. The population size specifies how many mutants will be produced in each generation, and it is a percentage of the total number of mutants for each application. As there are two ways of generating mutants (randomly and by crossover/mutation), the sum of both approaches is 100%.

The experiments followed these steps:

1) Execute all mutants against the test suite, recording test results and test execution times into "simulation record" YAML files. This is mostly for saving time when evaluating many different configurations of *BEMT* and *IEMT*: rather than having to re-run the same mutants again and again, we can refer back to our records. Real execution times can always be recovered from the recorded times.
2) Extract the list of the strong mutants from the recorded test results. These are the mutants we want to find through *BEMT* and *IEMT*.
3) From now on we are going to consider $S\%$ as the percentage of strong mutants, and $T\%$ as the percentage of mutants that will be generated with the first selected method before switching to the second one.
   For each target percentage S in $\{30\%, 45\%, 60\%, 75\%, 90\%\}$ and each switch-over threshold T in $\{5\%, 10\%, 15\%, 20\%, 25\%, 30\%, 35\%, 40\%\}$:
   a) Run *BEMT* with 30 different seeds, changing from random selection to *EMT* when $T\%$ is reached and stopping when we find $S\%$ of the strong mutants. Count how many mutants were executed.
   b) Run *IEMT* with 30 different seeds, changing from *EMT* to random selection when $T\%$ is reached and stopping when we find $S\%$ of the strong mutants. Count how many mutants were executed.

Ideally, we want to stop before executing $S\%$ of all the mutants: the fewer, the better. In order to evaluate the stability of the algorithm across seeds, each configuration was tested over 30 runs. To analyse if these executions stop before the mentioned percentage, the means and standard deviations are calculated. We have included four tables, one per case study, in a repository[2]. These tables contain the means and standard deviations of the percentages of all the mutants needed by *BEMT* and *IEMT* to find S% of the strong mutants.

[2]https://gitlab.com/lorgut/bemt-and-iemt/blob/master/Tables.pdf

Statistical tests have been used to check the significance of the results. As the results from *BEMT* and *IEMT* were not normally distributed, non-parametric Mann-Whitney U tests have been conducted using $\alpha = 0.05$ as the threshold for significance.

B. Case studies

We have selected three case studies with different purposes and from different sources: *Ecological Island* [25], *Terminal Self-Service* [8] and *DENMEvaP* [10].

Table III collates the results for each case study. The table contains, for each value of S%, how many times *BEMT* performs the same as *IEMT*, how many times *BEMT* is better than *IEMT*, and how many times *IEMT* is better than *BEMT*. These results are obtained from *Mann-Whitney U* test p-values. An initial test with a "*IEMT* performs differently than *BEMT*" alternative hypothesis is used. If the null hypothesis (i.e. they perform the same) is rejected, additional Mann-Whitney U tests with different alternative hypotheses (e.g. "*IEMT* performs better than *BEMT*") are run.

1) Ecological Island: This program was implemented at the University of Cadiz [25]. Its goal is to promote the development of *Smart Cities*. The ecological islands are gifted with "intelligence" in order to reduce business costs, reduce environmental pollution, and increase energy efficiency. The program contains four EPL queries, and obtains the JSON events from five channels of the *ThingSpeak* platform [26]. The total number of mutants is 181, where 61 are strong (33.7%).

Table III contains the results for this case study. Judging from the calculated values, it can be generally concluded that *BEMT* outperforms *IEMT*. The more percentage of strong mutants S% has to be found, the better *BEMT* is.

2) Terminal Self-Service: This case study is about a J2EE-based self-service terminal managing system in an airport that gets a large number of events (around 500 events per second) from connected terminals. This example was provided by EsperTech Inc. [9], and contains six EPL queries. The system receives from an integrated event generator a set of events. The events provide information about the source terminal. Each event carries similar information: the terminal identifier and a timestamp. The total number of mutants is 118, where 79 are strong (66.9%). Once again, after analysing the values of the table III, it can be said that *BEMT* outperforms *IEMT*, especially when the S% percentage is high.

3) DENMEvaP: DENMEvaP (Distributed Event-driven Network Monitoring Evaluation Prototype) is a program developed by Gad [10] which contains 37 EPL queries that processes events in PCAP (Packet CAPturer) format. The program is divided into five modules which are monitoring different tasks. In our study we are going to analyse two modules: SnifferCongestion and BruteForce, the mutants from the rest of modules were easily killed, so the modules were discarded. The test suite of the DENMEvaP program was obtained from the repository[3]. The author of DENMEvaP

[3]https://github.com/fg-netzwerksicherheit/DENMEvaP

S%	*BEMT* wins	Same	*IEMT* wins	*BEMT* wins	Same	*IEMT* wins
	Ecological Island			Terminal Self-Service		
0.30	2	2	4	1	4	3
0.45	4	3	1	3	5	-
0.60	5	3	-	4	4	-
0.75	6	2	-	8	-	-
0.90	8	-	-	8	-	-
	BruteForce			SnifferCongestion		
0.30	1	7	-	4	4	-
0.45	3	5	-	2	6	-
0.60	1	7	-	5	3	-
0.75	4	4	-	4	4	-
0.90	5	3	-	6	2	-

Table III
COMPARISON BETWEEN *BEMT* AND *IEMT*, MEASURED IN NUMBER OF VALUES OF $T\%$ WHERE MANN-WHITNEY U TESTS CONFIRMED SIGNIFICANT DIFFERENCES.

briefly explains the origin of each one. The BruteForce total number of mutants is 286, where 248 are strong (86.7%), and the SnifferCongestion total number of mutants is 61, where 34 are strong (55.7%).

For both modules, the values that are shown in Table III indicate that there is not a case when *IEMT* outperforms *BEMT*. In a 56% of the cases *IEMT* and *BEMT* have similar performance; in the rest of cases *BEMT* is better than *IEMT*.

Table III contains the results for each case study. The table shows in the "Same" column the number of values of T for that value of S where the Mann-Whitney U tests confirmed that *BEMT* and *IEMT* showed no significant differences. "*BEMT* wins" counts the cases when *BEMT* was shown to be significantly better than *IEMT*, and "*IEMT* wins" counts the cases when *IEMT* was significantly better than *BEMT*.

4) Overall results: After reviewing the results, we can conclude that in all of the case studies, *BEMT* is better than *IEMT* when at least 75% of the strong mutants are required. The answer to RQ1 is that *BEMT* is the best metaheuristic of the two.

C. BEMT and EMT comparison

In order to answer RQ2, the previous *BEMT* results are going to be compared with the results obtained for *EMT* in [13]. Table IV shows, for each value of S%, how many times *EMT* performs as well as *BEMT*, better than *BEMT* or worse than *BEMT*. The same process described in Subsection III-B has been followed: *Mann-Whitney U* tests are executed to check if there are any significant differences between *EMT* and *BEMT* for that case study and value of S%. *EMT* runs without considering a threshold T%, so *EMT* values are the same for each T%.

After comparing the behaviour of *EMT* against *BEMT*, we can provide an answer for RQ2: the *BEMT* metaheuristic behaviour is quite similar to basic *EMT*, so the implemented metaheuristic is not better than the original execution. However, it can be seen that in the Ecological Island and Terminal Self-Service cases study, *EMT* outperforms *BEMT*, in some of the cases when S% is low. This means that, when we are comparing *EMT* and *BEMT*, with the exception of a few cases when S% is low, their behaviour is quite similar. In conclusion, *BEMT* is not an improvement over *EMT*.

D. Checking mutation operators

Looking at Tables III and IV, it was clear that for the DENMEvaP case studies, EMT and BEMT were not doing as well as for the Ecological Island and Terminal Self-Service cases. This suggested there was something different about those case studies. To clarify the reason, it was decided to study the worst performing DENMEvaP program among them: BruteForce.

Particularly, the distribution of strong mutants across all operators was studied. In general, EMT operates under the assumption that strong mutants are a restricted subset of the set of all mutants: under this assumption, a directed search should work better than a random sampling of the set of all mutants. In contrast, in BruteForce 94 of all 286 mutants were strong (surviving all tests), with 64 of them from the same operator (RRO). RRO produced 165 of all mutants, largely dominating the search space. This large proportion of mutants went against the assumptions made by EMT, and appeared as a strong factor for its poor performance.

To confirm this, a new set of executions of EMT and BEMT was performed on BruteForce, while excluding the 165 mutants from RRO. This left only 29 strong mutants among the 121 remaining ones. The Mann-Whitney U test comparison results for this case are shown on Table V. The left half is dedicated to comparing the two new variants (*BEMT* and *IEMT*) in this situation, and the right half compares the best option so far (*BEMT*) with the original (EMT).

The left half of the table shows a change in the relative performance of *BEMT* and *IEMT*: for small values of S (small subsets of the strong mutants) and large values of T (large thresholds for switchover), *IEMT* compares favourably. *IEMT* does better in these situations because doing a large sample right before switching to a directed search is already enough to catch the requested small subset. However, as soon as a larger subset is required, the directed search performed by *BEMT* becomes important.

S%	*EMT* wins	Same	*BEMT* wins	*EMT* wins	Same	*BEMT* wins
	Ecological Island			Terminal Self-Service		
0.30	6	2	-	6	2	-
0.45	5	3	-	3	5	-
0.60	1	7	-	8	-	-
0.75	1	7	-	2	6	-
0.90	0	8	-	4	4	-
	BruteForce			SnifferCongestion		
0.30	-	8	-	-	8	-
0.45	-	8	-	-	8	-
0.60	-	8	-	-	8	-
0.75	-	8	-	-	8	-
0.90	-	8	-	-	8	-

Table IV
COMPARISON BETWEEN *EMT* AND *BEMT*, MEASURED IN NUMBER OF VALUES OF $T\%$ WHERE MANN-WHITNEY U TESTS CONFIRMED SIGNIFICANT DIFFERENCES.

S%	*BEMT* wins	Same	*IEMT* wins	*BEMT* wins	Same	EMT wins
	BruteForce No RRO			BruteForce No RRO		
0.30	-	4	4	-	2	6
0.45	3	3	2	-	4	4
0.60	5	3	-	-	6	2
0.75	6	2	-	-	6	2
0.90	8	-	-	-	8	-

Table V
COMPARISONS BETWEEN *BEMT*, EMT, AND *IEMT*, MEASURED IN NUMBER OF VALUES OF $T\%$ WHERE MANN-WHITNEY U TESTS CONFIRMED SIGNIFICANT DIFFERENCES.

The right half of the table shows how avoiding RRO (which produces far too many surviving mutants) clearly favored EMT over *BEMT*, showing that it is worth going for a directed search in all generations rather than starting with a random sample. This suggests that the definition of the RRO operator should be refined, making it produce fewer and more useful mutants.

IV. RELATED WORK

Speaking about real-time systems, mutation testing has been applied to many traditional programming languages that have been growing and now can be applied to real-time systems: Java [19], C [1], or Ada [23].

There are also mutation testing studies which are focused on real-time systems. R. Nilsson et al. have published works where they propose a model-based approach for mutation testing, which analyses models of real-time systems to generate test cases for the automated testing of timeliness [21]. Nilsson followed up this work with heuristics-based simulations for test generation [22], and with a mutation-based framework for the automated testing of timeliness [20].

If testing a real-time system is a time-consuming activity, the main drawback of mutation testing has to be taken into account [24]: the high computational cost involved in the execution of the large number of mutants produced for some programs against their test suites. Several cost reduction techniques which attempt to run fewer mutants exist [17], such as *selective mutation* [2] which selects a subset of the mutation operators, *mutant sampling* [4] which selects a subset of the mutants randomly, or *high order mutation (HOM)* [16] which combines several faults into a mutant. *EMT* [6] was proposed to improve the test suite, and was applied to 3 WS-BPEL compositions. The technique was later extended to generate HOMs [3].

EMT was implemented in the *GAmera* tool [7], and the experiments with WS-BPEL showed a reduction in the number of mutants compared to pure random selection. The study also determined the optimal values for the configuration parameters of *EMT*. *GAmera* had an architecture which allowed its reuse for other programming languages: Delgado et al. extended *EMT* to the C++ language, and observed a significant reduction in the number of mutants when compared to random selection as well [5].

V. CONCLUSIONS AND FUTURE WORK

In this work we have proposed two metaheuristics that build upon *EMT*: Bootstrapped *EMT*, which bootstraps the hall of fame of *EMT* with a larger set of randomly selected individuals, and Inverse *EMT*, which uses *EMT* until it stabilises and then switches over to random selection. This switch is done based on a threshold on the ratio of mutants generated so far. The hypothesis has been tested through two research questions. RQ1 asked which metaheuristic was better: *BEMT* was the winner. RQ2 asked if the best metaheuristic was better than plain *EMT*: the general answer to RQ2 is that there were no cases where *BEMT* surpassed *EMT*. However, it was observed that the definition of the mutation operators played a large role: excluding the RRO operator in BruteForce showed a noticeable improvement in performance for *BEMT* and EMT over the random search favored by *IEMT*. The definition of RRO should be refined to reduce the number of mutants it generates and improve their usefulness in designing test cases.

This suggests that, in order to improve the performance of *EMT*, more information about the executions is required. One option to make the execution matrix more nuanced is to add a new value that differentiates a surviving mutant where the mutation was not reached, from a surviving mutant where the mutation was reached. Another option would be to consider the proportion of the query that has been exercised (coverage ratio) for the fitness of the individuals. It may also be useful to "profile" the likelihood a mutation operator produces high-quality mutants (e.g. across a suite of programs in the same language), and to have the fitness function take that into account.

References

[1] Hiralal Agrawal et al. *Design of Mutant Operators for the C Programming Language*. Technical report SERC-TR-41-P. Software Engineering Research Center, Purdue University. West Lafayette, Indiana: Purdue University, 1989.

[2] Ellen Francine Barbosa, Jose Carlos Maldonado, and Auri Marcelo Rizzo Vincenzi. "Toward the determination of sufficient mutant operators for C." In: *Software: Testing Verification and Reliability* 11 (2001).

[3] Emma Blanco-Muñoz et al. "Towards higher-order mutant generation for WS-BPEL". In: *Proceedings of the International Conference on e-Business (ICE-B), 2011*. IEEE. 2011, pp. 1–6.

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

[5] P. Delgado-Pérez et al. "GiGAn: Evolutionary Mutation Testing for C++ Object-Oriented Systems". In: *Software Verification and Testing (SAC-SVT 2017), Marrakech*. 2017.

[6] J. J. Domínguez-Jiménez et al. "Evolutionary Mutation Testing". In: *Information and Software Technology* 53.10 (2011), pp. 1108–1123. ISSN: 0950-5849. DOI: 10.1016/j.infsof.2011.03.008.

[7] J. J. Domínguez-Jiménez et al. "GAmera: An Automatic Mutant Generation System for WS-BPEL Compositions". In: *Proceedings of the 7th IEEE European Conference on Web Services*. Ed. by Rik Eshuis, Paul Grefen, and George A. Papadopoulos. Eindhoven, The Netherlands: IEEE Computer Society Press, 2009, pp. 97–106. ISBN: 978-0-7695-3854-9.

[8] EsperTech. *EsperTech Self-Service Terminal*. http://archive.is/UzhaN. last checked: 7 October 2019. 2017.

[9] EsperTech. *EsperTech website*. http://www.espertech.com. last checked: 7 October 2019.

[10] Ruediger Gad. "Event-driven Principles and Complex Event Processing for Self-adaptive Network Analysis and Surveillance Systems". PhD thesis. University of Applied Sciences Frankfurt am Main, 2015.

[11] David E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. 1989.

[12] Lorena Gutiérrez-Madroñal. "Generación Automática de Casos en Procesamiento de Eventos con EPL - Authomatic Generation of Cases in Event Processing using EPL". PhD thesis. University of Cadiz, 2017.

[13] Lorena Gutiérrez-Madroñal, Antonio García-Domínguez, and Inmaculada Medina-Bulo. "Evolutionary mutation testing for IoT with recorded and generated events". In: *Software: Practice and Experience* 49.4 (2019), pp. 640–672. DOI: 10.1002/spe.2629.

[14] Lorena Gutiérrez-Madroñal, Inmaculada Medina-Bulo, and Juan José Domínguez-Jiménez. "Evaluation of EPL Mutation Operators with the MuEPL Mutation System". In: *Expert Systems with Applications* 116 (2019), pp. 78–95. ISSN: 0957-4174. DOI: 10.1016/j.eswa.2018.09.003.

[15] Walter S. Heath. *Real-time Software Techniques*. New York, NY, USA: Van Nostrand Reinhold Co., 1991. ISBN: 0-442-00305-6.

[16] Y. Jia and M. Harman. "Constructing Subtle Faults Using Higher Order Mutation Testing". In: *Source Code Analysis and Manipulation, 2008 Eighth IEEE International Working Conference on*. 2008, pp. 249–258.

[17] Yue Jia and M. Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *Software Engineering, IEEE Transactions on* 37.5 (2011), pp. 649–678. ISSN: 0098-5589. DOI: 10.1109/TSE.2010.62.

[18] Yue Jia and Mark Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *Software Engineering, IEEE Transactions on* 37.5 (2011), pp. 649–678. ISSN: 0098-5589. DOI: 10.1109/TSE.2010.62.

[19] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. "MuJava: a mutation system for Java". In: *Proceedings of the 28th international conference on Software engineering*. ACM. 2006, pp. 827–830.

[20] Robert Nilsson and Jeff Offutt. "Automated testing of timeliness: A case study". In: *Proceedings of the Second International Workshop on Automation of Software Test*. IEEE Computer Society. 2007, p. 11.

[21] Robert Nilsson, Jeff Offutt, and Sten F Andler. "Mutation-based testing criteria for timeliness". In: *Computer Software and Applications Conference, 2004. COMPSAC 2004. Proceedings of the 28th Annual International*. IEEE. 2004, pp. 306–311.

[22] Robert Nilsson, Jeff Offutt, and Jonas Mellin. "Test case generation for mutation-based testing of timeliness". In: *Electronic Notes in Theoretical Computer Science* 164.4 (2006), pp. 97–114.

[23] A Jefferson Offutt, Jeff Voas, and Jeff Payne. *Mutation operators for ada*. Tech. rep. Technical Report ISSE-TR-96-09, Information and Software Systems Engineering, George Mason University, 1996.

[24] A. Jefferson Offutt and Roland H. Untch. “Mutation 2000: Uniting the Orthogonal”. In: *Mutation Testing for the New Century*. Boston, MA: Springer US, 2001, pp. 34–44. ISBN: 978-1-4757-5939-6.

[25] Daniel J. Rosa-Gallardo et al. “Sustainable WAsTe Collection (SWAT): One Step Towards Smart and Spotless Cities”. In: *Service-Oriented Computing – ICSOC 2017 Workshops*. Ed. by Lars Braubach et al. Cham: Springer International Publishing, 2018, pp. 228–239. ISBN: 978-3-319-91764-1.

[26] ThingSpeak. *ThingSpeak website*. https://thingspeak.com. last checked: 7 October 2019.