Using a swarm to detect hard-to-kill mutants

Alfredo Ibias
Universidad Complutense de Madrid
Madrid, Spain
aibias@ucm.es

Manuel Núñez
Universidad Complutense de Madrid
Madrid, Spain
manuelnu@ucm.es

***Abstract*—Mutation Testing is an effective testing technique that relies in the generation of mutants from the system under test. The main limitation of this technique is that the potential number of mutants is usually huge. Therefore, it is important to classify and select mutants in order to avoid repetitive, useless or excessive computations, and biased results. In this paper we focus on avoiding too many executions and/or biased results by classifying mutants into two categories: hard-to-kill and easy-to-kill mutants. We propose a new swarm intelligence algorithm to classify a set of mutants between those two classes and we show how our algorithm compares to other approaches.**

***Index Terms*—Mutation Testing, Swarm Intelligence, Mutant Selection**

I. Introduction

Software Testing [2], [31] is the most widely used technique to detect faults in software systems. Software testing includes different approaches and methodologies that target specific categories of faults. Most approaches try to increase *code coverage*, that is, try to build test suites that *traverse* all the paths of the system that are relevant with respect to a certain criterion. In this paper we focus on *mutation testing*, an approach that does not only focus on showing *where* to test, but also on helping to identify *what* should be checked for. Experimental evidence has showed that tests suites produced by mutation testing approaches were significantly better than the (high quality) manually written ones [17]. Intuitively, mutation testing considers a software system, that we would like to evaluate, and variants of this system, called *mutants*, that represent potential faults of the system. The goal is to find good test suites that *kill* all the mutants: a test case kills a mutant if the application of the test to the original system and to the mutant produces different results. If a mutant is *alive* after the application of all the test cases, then we have to analyse whether our test suite was not good enough or the mutant is equivalent to the original system.

Mutant selection is critical because it ameliorates the scalability problem associated with mutation testing: usually, we have huge amounts of potential mutants. Producing and working with a large number of mutants is impractical, as they need to be analysed, compiled, executed and killed by test cases. This is an important problem and, actually, makes difficult the wide applicability and large adoption of mutation testing. Classically, mutant selection tried to solve this problem through the reduction of the size of the mutants sets used in the process, defining mutant reduction strategies such as selective mutation [36], [43] and random mutant selection [1].

Previous work has focused on classifying mutants based on different characteristics. For example, it is usually assumed that *harder* to kill mutants are more useful than the easy to kill ones, but there are many categories. Hard-to-kill mutants are usually considered to be the ones killed by a small fraction of the considered test cases, but there is also work identifying hard-to-kill mutants based on the internal structure of the code of the given system under test [42]. A recent work [37] identifies different criteria to classify mutants (hard to kill, subsuming, hard to propagate and fault revealing) and show that each of them classifies different mutants as the preferable ones for mutation testing. Moreover, the authors found that there is a weak connection between these classifications and fault revelation. However, they conclude that hard-to-kill mutants are the ones more related to fault revelation.

Our goal in this paper is to classify a set of mutants between hard-to-kill and easy-to-kill mutants, with the idea that this kind of mutants gives a compromise between easy to classify and fault detection ability. We propose an approach based in the Swarm Intelligence Algorithms theory [6], [40], a family of algorithms that naturally falls into the Computational Collective Intelligence [10], [32]–[34] research area, to address this problem. In previous work other authors have proposed different approaches for mutant classification, including using machine learning methods [13], [25], [28], but we are not aware of a hard-to-kill mutant classification algorithm based on Swarm Intelligence Algorithms theory. In our work, we use a swarm of agents to apply tests to small sets of mutants with the goal of building a hard-to-kill set of mutants from the results obtained by the agents.

The paper is organised as follows. First, in Section II, we present some basic concepts needed to understand our work. Then, we explain our algorithm (Section III) and we present the experiments that we performed to assess its performance (Section IV). Finally, we present the threats to the validity (SectionV) and the conclusions of our work (Section VI).

II. Preliminaries

In this section we present basic information about the two main fields that we consider in this paper: mutation testing and

Research partially supported by the Spanish MINECO/FEDER project FAME (RTI2018-093608-B-C31), the Comunidad de Madrid project FORTE-CM (S2018/TCS-4314) co-funded by EIE Funds of the European Union and the Region of Madrid - Complutense University of Madrid (grant number PR65/19-22452).

swarm intelligence algorithms.

A. Mutation Testing

Software Testing is a broad field of techniques with the goal of detecting faults in software systems. Essentially, testing consists in applying a set of inputs to the System Under Test (SUT), observe the generated output, and decide whether this output is consistent with the expected output. In order to perform this decision, it is necessary some kind of oracle that determines which one is the correct output. In some cases, there is no oracle available and the testers should resort to other kind of techniques to determine what constitutes a valid output [3]. One of the techniques to overcome this oracle problem is *mutation testing*. Mutation testing is not just an *academic methodology* because it has been successfully used in real software systems [38].

A *mutant* is a modified version of the SUT that includes a fault. These faults can be randomly seeded or following some guide and can be generated either automatically or manually. Then, when applying a test to a mutant, we will check whether the produced output is different to the output obtained after applying the test to the original SUT. If the outputs are different, then we say that the mutant has been killed. If the mutant is not killed then we have three different possibilities: the fault has not been executed, the fault has been executed but it has not propagated to the output, or the mutant is equivalent to the original SUT. Intuitively, mutation testing uses the SUT as a kind of oracle and generates mutants from it with the goal of having faulty versions of the SUT to assess the quality of the generated tests. Then, the quality of a test is assessed with a metric called *mutation score*, which represents the percentage of mutants that the test has killed.

Just as we can classify the tests using their mutation score, we can classify the mutants depending on how many tests kill those mutants. This idea crystallises in the concept of *hard-to-kill* mutants, that is, those mutants that are killed by a small amount of tests (as opposed to easy-to-kill mutants, which are those that are killed by most of the tests). In previous work, there have been different definitions of hard-to-kill mutants, all of them with the idea that hard-to-kill mutants are the hardest to find with a test. It is possible to provide a certain bound, for example, those killed by 5% or 2.5% of the tests [37]. Another possibility [42] is to mark a mutant as hard-to-kill if it presents a specific internal structure. In this paper we will consider as hard-to-kill mutants those that are the least killed, using a variable measure instead of a fixed cap.

B. Swarm Intelligence Algorithms

Swarm Intelligence Algorithms [6], [40] are a family of algorithms that base its *intelligent* behaviour in their swarm interactions. In this kind of algorithms, there is a *swarm* of agents, where each agent has little to no intelligence. Usually, these agents perform basic and repetitive tasks. The *intelligent* behaviour from the Swarm Intelligence Algorithms comes from the fact that joining all the information obtained by the individual agents allows the swarm to perform intelligent decisions. This characteristic of the Swarm Intelligence Algorithms is the so called *emergence property*. This property is the basic key for the good results obtained by Swarm Intelligence Algorithms, like the widely known Particle Swarm Optimisation (PSO) algorithm [26] and its most recent variation the Tree Swarm Optimisation [20]. This last algorithm has been recently applied to the Software Testing field [19].

```
Set parameters;
Initialise kill matrix (all zeros);
Initialise iteration-hard-to-kill list (all mutants);
Initialise final-hard-to-kill list (empty);
while iteration-hard-to-kill list is not empty do
    Assign a mutant and a set of tests to each agent;
    Each agent applies its set of tests to its mutant;
    Each agent updates the kill matrix;
    Update iteration-hard-to-kill list;
end
Return final-hard-to-kill list;
```

Algorithm 1: Heuristic: general scheme

III. Swarm Mutant Classification

We work with a scenario where we have m mutants and t tests. Our goal is to detect those mutants that are hard to kill by this set of tests, having in mind that our approach should represent a good balance between the computing time and the quality, in terms of the proportion of (un-)detected interesting mutants, of the obtained solution. We have considered a swarm heuristic that avoids the application of the full set of tests to all the mutants and that, at the same time, gives more flexibility than setting a fixed cap to decide when a mutant is hard-to-kill [37].

Our heuristic uses three important elements:

- *Agents* conform the swarm that performs the evaluation. We will have a agents.
- The *Kill Matrix* is the matrix where the agents store the information. It will encode which tests kill which mutants, which tests fail to kill which mutants, and which tests have not been applied to which mutants.
- The *hard-to-kill mutants lists* store the *promising* hard-to-kill mutants. These lists are updated after each iteration of our algorithm. We will have two hard-to-kill mutants lists: one for storing the considered hard-to-kill mutants in the current iteration (this one guides the algorithm), and one for storing the final solution.

In Algorithm 1 we present a high-level view of our heuristic. For each iteration, our heuristic has four steps:

- For each of the agents, we choose one mutant from the iteration-hard-to-kill list and choose $n << t$ tests[1] that will be applied to that mutant (and that have not been applied previously) and assign them to an agent. If we detect that a mutant cannot leave the iteration-hard-to-kill list with the remaining tests, we add this mutant to the final-hard-to-kill list and take another mutant.
- Each agent applies its set of n tests to its mutant.

[1]Note that n is chosen by the user. For our experiments, we used $n = 5$.

- Each agent updates the kill matrix. We use the following convention:
  - 1: the mutant has been killed by that test.
  - 0: the test has not been applied to the mutant.
  - -1: the mutant has not been killed by that test.
- The iteration-hard-to-kill list is updated.

The update of the iteration-hard-to-kill list is the most critical step of the algorithm, because it is where the *emergence property* arises. In this step, the list of mutants considered to be hard-to-kill is updated. The selection is performed after each iteration as a way to guide the development of the algorithm. After each iteration we compute how many tests killed each mutant, storing the highest ($\max$) and the lowest values ($\min$). Then, we mark as hard-to-kill mutants the ones whose value is less than or equal to:

$$\min + \frac{\max - \min}{4}$$

Note that this bound is selected by us but other different bounds could be used. For instance, using this bound we choose all the mutants that after this iteration are in the the lowest quarter of the obtained values. Finally, we remove from the iteration-hard-to-kill list the mutants that are already in the final-hard-to-kill list. We also remove those that have already been tested with all the tests, and we add them to the final-hard-to-kill list.

An interesting property of our heuristic is that the $\max$ value does not have to be equal to the maximum number of tests that kill a mutant. In other words, even if a mutant is killed by all the tests, it is possible that the $\max$ value is lower than t (the total number of tests). This happens because the difference between $\max$ and $\min$ will be lower or equal to $\frac{4}{3} \cdot n$ and, therefore, we can have $\max < \min + \frac{4}{3} \cdot n < t$ (and that will be usually the case).

Our heuristic is able to overcome two problems when deciding which mutants are hard-to-kill and which ones are not. The first one is that, in general, it is not necessary to apply all the tests to all the mutants. This will avoid the associated costs of other algorithms based on brute force. The second one is that our heuristic is more flexible than an approach based on fixed percentages. For example, if we define hard-to-kill mutants as the mutants that are killed by at most 5% of the tests, but we have a situation where we have 100 tests and all the mutants are killed by at least 6 tests, then the set of hard-to-kill mutants will be empty. If we use our algorithm in the same situation, our set of hard-to-kill mutants will not be empty because those 6 tests will provide the $\min$ value of our range. Therefore, we will always have mutants that can be considered, under the circumstances, the *hardest-to-kill* of all the generated mutants. This implies that our heuristic will be more consistent than other algorithms like random selection.

IV. Experiments

In this section we present the experiments that we performed to measure the usefulness of our approach. We wanted to compare our algorithm with three different alternatives: a brute force approach that consists in executing all the tests over all the mutants and afterwards determining which mutants we consider hard-to-kill; a cap approach where we execute only the necessary tests to know if a mutant overcomes a previously fixed cap; and a random algorithm that executes randomly tests on mutants and then computes the solution. Therefore, our main goal was to answer two questions:

Research Question 1: How many test applications does our approach save when compared to a brute force approach? How different is our approach from a *cap-based* approach?

Research Question 2: What is the quality of the solution of our approach and how *good* it is compared with the solution obtained by a *cap-based* approach and a random approach?

Our heuristic strongly depends on the application of tests to mutants. However, we only use whether a given mutant was killed by a certain test; we do not use any information concerning the actual application of the test (e.g. which parts of the code of the mutant were traversed). Therefore, we only need a *kill matrix* encoding the result of the application of tests to mutants. Each position i, j of the matrix says whether the ith test kills the jth mutant. We have used the matrices provided by a recent work [13], available at https://mutationtesting.uni.lu/farm/. These matrices where build from a set of mutants generated from the CodeFlaws [41] and CoREBench [7] program sets, and a set of tests generated by using KLEE [9] for each program. This combination arises $1,737$ matrices, with a total count of $4,778,157$ mutants and $144,738$ tests, what needed $8,463$ CPU days of computation. Breaking down by benchmark, the CodeFlaws program set brings a total of $3,213,543$ mutants and $122,261$ test cases from $1,692$ programs with between 1 to 322 lines of code (mean of 36 lines of code). The CoREBench program set brings a total of $1,564,614$ mutants and $22,477$ test cases from 45 programs with between $9,000$ and $83,000$ lines of code. In computation terms, the CodeFlaws benchmark needed $8,009$ CPU days of computation and the CoREBench benchmark used 454 CPU days of computation to generate all their mutants.

We applied our heuristic to the kill matrix of each program, computing the number of total operations (that is, the number of tests that are applied) and computing the resulting hard-to-kill mutants. As an additional step we computed average values and some quality indicators. These last values will be useful to compare our solutions with respect to previous work [37] where hard-to-kill mutants are those killed by less than 5% of tests. Specifically, we would like to know how different are our hard-to-kill mutants sets from the ones generated using a *fixed cap* approach and how many extra operations we save. Therefore, we also implemented an algorithm to compute those mutants that are killed by 5% of the the tests or less. The algorithm is very simple: it traverses each row of the matrix until more than 5% of the tests have killed the mutant (so it is not a hard-to-kill mutant). We store the quality indicators of these sets of mutants and the number of operations (that is, the number of elements of the matrix that have been accessed) needed to compute them.

In order to have an easier visualisation of the results of our experiments, we combine all these values and plot them. In Figure 1 we have the relative number of operations for each algorithm with respect to the total number of operations obtained by all three algorithms: Brute Force, our Swarm

Mutant Classification (SMC) and the cap algorithm. We can observe that our SMC always needs less operations than the Brute Force algorithm and, depending on the program, may need less operations than the cap algorithm. On average, the Brute Force algorithm needed $70,041$ operations while our SMC needed $25,255$ operations and the cap algorithm needed $25,109$ operations. That is, our SMC needs, on average, 61.97% less operations (that would be applications of tests if we do not have the killing matrix) while the cap algorithm saves, on average, 68.99%.

We performed a statistical hypothesis test over the results concerning operations. The null hypothesis was that the cap algorithm and our SMC algorithm give similar results, that is, both need a similar number of operations. We applied a one-way ANOVA test[2] where we tested whether the values of both algorithms are, on average, similar. Then, we computed the p-value for the experiment, obtaining a p-value of 0.9044. Therefore, we can confirm the null hypothesis for this experiment because its p-value is much higher than 0.05. In order to double-check our results, we performed a t-test and obtained the same p-value. Thus, the conclusion is that the needed number of operations of our SMC is equivalent to the number needed for the cap algorithm.

In Figure 2 we present the relative operations percentage saving with respect to the saving of both the cap algorithm and our algorithm. We can see how they save some operations with respect to the Brute Force algorithm and how, sometimes, our algorithm outperforms the cap algorithm.

Next, we wanted to perform a quality assessment. In order to do so, we compared our algorithm and the cap algorithm with another new algorithm: the random algorithm. This algorithm applies random tests to random mutants, filling a kill matrix, and then chose the mutants that are killed by at most a fixed number of tests. In our case, in order to compare with the cap algorithm, we decided to take the mutants killed by less than 5% of the tests. Also, in order to compare with our SMC algorithm, we decided that the number of tests that the random algorithm will apply will be equal to the number of tests applied by our algorithm.

We assessed the quality with three indicators: how many mutants that are killed by less than the 5% of tests are not included in the solution; how many mutants that are killed by more than the 25% of tests are included in the solution; and how many mutants, that are killed by less tests than the ones that kill the mutant of the solution that is killed by more tests, are included in the solution. Using these three indicators, we assess three different qualities of the hard-to-kill mutants sets, respectively: how good is the algorithm obtaining the most hard to kill mutants; how good is the algorithm avoiding mutants that cannot be considered hard-to-kill, and how good is the algorithm obtaining dense hard-to-kill mutants sets.

The results of the three algorithms are positive. For the cap algorithm, as it is pretty consistent, the values for the three indicators are equal to 0. For the random algorithm, the mean for the first indicator is 0, for the second indicator is 31.61 and for the third indicator is 303.63. Finally, for our algorithm, that is more flexible than the cap algorithm and it is *less* random than the purely random algorithm, the mean for the first indicator is 24.02, the mean for the second indicator is 1.12, and the mean for the third indicator is 51.16. Analysing these results, we can conclude that the cap algorithm gives what it says: the mutants that are killed by less than the 5% of tests; that the random algorithm gives hard-to-kill mutants sets that are more hollow, what indicates that their choice criteria is less uniform; and that our algorithm gives hard-to-kill mutants sets that are a middle point between the fixed criteria and the random criteria, with more flexibility than the cap algorithm, and with a huge improvement in the choice criteria over the random algorithm.

In Figure 3 we summarise the results concerning the quality of the obtained mutants. We show the cumulative values of the first and second indicator for the two algorithms that obtained values different from 0 and their relative values with respect to the ones obtained by the other algorithm. White lines correspond to cases were all the algorithms obtained a value of 0 for both indicators.

As a recap, answering the first research question, our SMC saves 61.97% of the operations of the brute force approach and needs a similar number of operations as the cap algorithm. In fact, regarding number of operations, both algorithms are statistically equivalent. Concerning the second research question, our approach obtains hard-to-kill mutants sets of good quality. In fact, their quality is better than the quality of the solutions of a random approach, and not so far to the quality of the solutions of the cap approach. However, an advantage of our SMC over the cap approach is that it avoids some extreme cases (from both sides) that appear with it, what makes it a more reliable tool to be used.

V. Threats to validity

Concerning the threats to the validity of our results, most of them have been already addressed. Starting with the internal validity threats, the main concern is whether our results can be the product of internal faults in our experiments code. We addressed this concern by thoroughly testing our code with carefully constructed examples for which we could manually check the results. Another important internal validity threat is whether our results are valid in terms of time computation while using kill matrices instead of properly apply the tests to the mutants. In order to address this threat we compared the number of performed operations instead of execution times, under the assumption that the difference in execution time between tests will not be so critical as the difference in execution time between applying different number of tests. A final internal validity threat is how the randomness associated with the random algorithm affects the obtained results. In order to overcome this threat we repeated the same experiment different times and see that the mean results where similar enough to be considered representative.

Concerning threats to external validity, here arises the question of whether the kill matrices used in our work can be generalised to other families of kill matrices. Although this

[2]Note that we could use the ANOVA test because we performed an homogeneity of variance check and it raised a positive result.

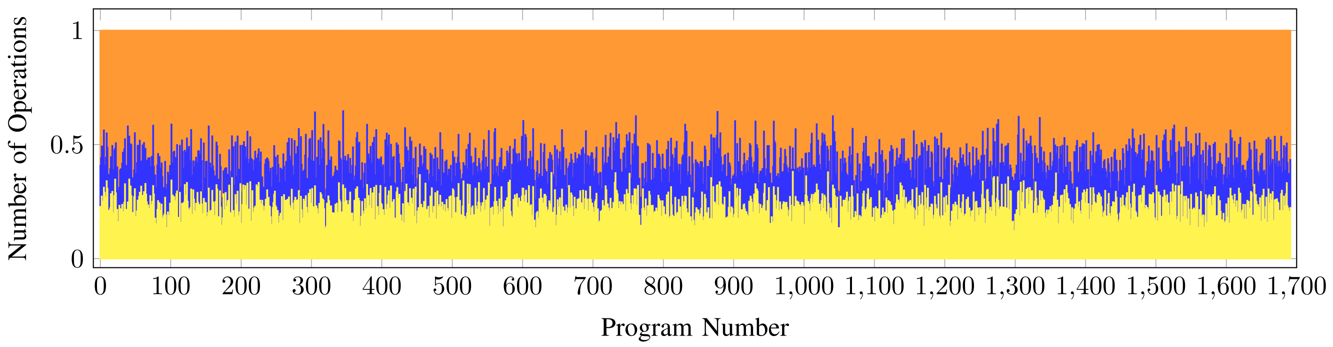

Fig. 1. Relative number of operations (orange = Brute Force, yellow = SMC, blue = cap (5%)).

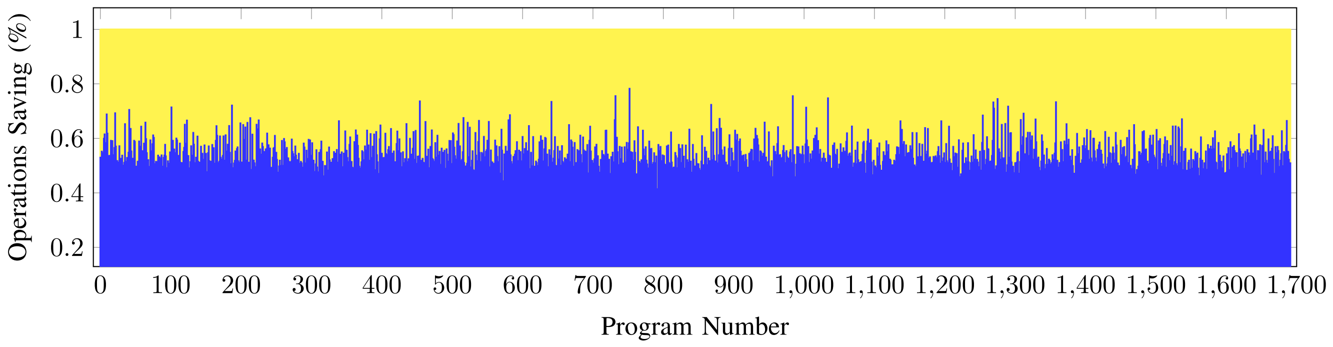

Fig. 2. Relative number of operations savings (yellow = SMC vs Brute Force, blue = cap (5%) vs Brute Force).

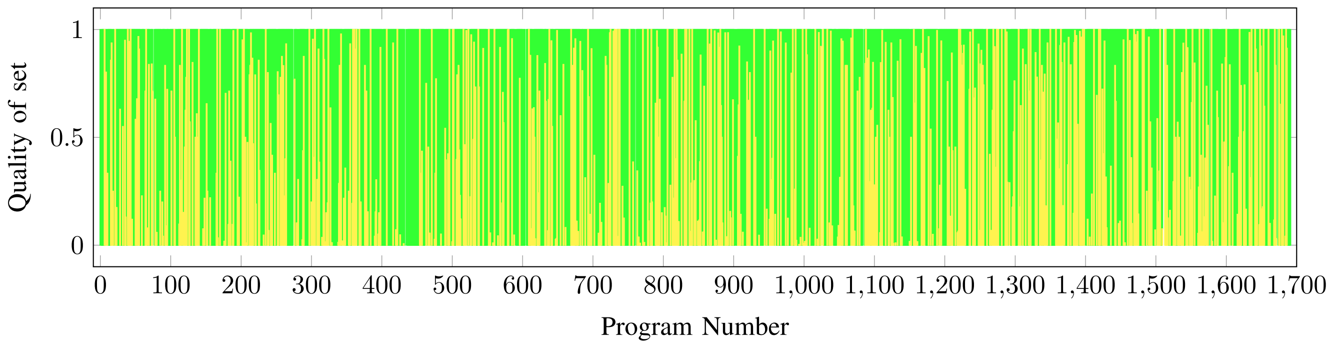

Fig. 3. Relative quality of sets of hard-to-kill mutants (yellow = SMC, green = random).

threat cannot be completely addressed, we worked with kill matrices generated from real programs and with mutants and tests generated by state-of-the-art techniques.

Finally, concerning threats to construct validity, the main concern is whether the kill matrices used in our work can be representative of real programs. Fortunately, our kill matrices where generated from real programs, so they are indeed representative of *real* kill matrices.

VI. Conclusions

Mutation testing is one of the main techniques in Software Testing. In order to perform a good and efficient mutation process, it is necessary to *filtrate* the mutants. In this work, we focused on detecting hard-to-kill mutants. As the concept of hard-to-kill mutants is too abstract, we developed a Swarm Intelligence Algorithm in order to choose a set of hard-to-kill mutants from the set of all the mutants of a program. We performed several experiments to prove the efficiency of our algorithm when compared to other approaches to the concept of hard-to-kill mutants. We showed that our SMC is a preferable option to detect those hard-to-kill mutants.

For future work we have identified several lines. First, we would like to assess how the hard-to-kill mutants set changes when modifying the bound for choosing the hard-to-kill mutants in the main loop of our algorithm. Second, we would like to compare the efficiency of our algorithm in a weak mutation scenario, compared to the current strong mutation scenario we presented here. Third, we would like to compare our approach to other Swarm Intelligence Algorithms like Particle Swarm Optimisation [26] and its variations. Fourth, we would like to assess how related are the hard-to-kill mutants determined by our algorithm and the set of fault revealing mutants [37]. Fifth, we would like to deal with bigger sets of mutants by including recent approaches to mutation testing [11], [15], [18] and our recent work on heuristics based on Information Theory [24]. Finally, we would like to take previous research as initial step to generalise the framework and measures to deal with asynchronous [21], [27], [29], [30], distributed [8], [16], [22], [23], IoT [14], [39] and cloud [4], [5], [12], [35] systems.

References

[1] A. T. Acree, A. T. Budd, R. Demillo, R. J. Lipton, and F. G. Sayward. Mutation analysis. Technical Report GIT-ICS-79/08, Georgia Institute of Technology, 1979.

[2] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2nd edition, 2017.

[3] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[4] A. Bernal, M. E. Cambronero, A. Núñez, P. C. Cañizares, and V. Valero. Improving cloud architectures using UML profiles and M2T transformation techniques. *The Journal of Supercomputing*, 75(12):8012–8058, 2019.

[5] A. Bernal, M. E. Cambronero, V. Valero, A. Núñez, and P. C. Cañizares. A framework for modeling cloud infrastructures and user interactions. *IEEE Access*, 7:43269–43285, 2019.

[6] C. Blum and D. Merkle, editors. *Swarm Intelligence: Introduction and Applications*. Springer, 2008.

[7] M. Böhme and A. Roychoudhury. CoREBench: studying complexity of regression errors. In *23rd Int. Symposium on Software Testing and Analysis, ISSTA'14*, pages 105–115. ACM Press, 2014.

[8] J. Boubeta-Puig, G. Díaz, H. Macià, V. Valero, and G. Ortiz. MEdit4CEP-CPN: An approach for complex event processing modeling by prioritized colored Petri nets. *Information Systems*, 81:267–289, 2019.

[9] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI'08*, pages 209–224. USENIX Association, 2008.

[10] A. Camacho, M. G. Merayo, and M. Núñez. Collective intelligence and databases in eHealth: A survey. *Journal of Intelligent & Fuzzy Systems*, 32(2):1485–1496, 2017.

[11] P. C. Cañizares, A. Núñez, and M. G. Merayo. Mutomvo: Mutation testing framework for simulated cloud and HPC environments. *Journal of Systems and Software*, 143:187–207, 2018.

[12] P. C. Cañizares, A. Núñez, J. de Lara, and L. Llana. MT-EA4Cloud: A methodology for testing and optimising energy-aware cloud systems. *Journal of Systems and Software*, 163:110522:1–110522:25, 2020.

[13] T. T. Chekam, M. Papadakis, T. F. Bissyandé, Y. Le Traon, and K. Sen. Selecting fault revealing mutants. *Empirical Software Engineering*, 25(1):434–487, 2020.

[14] D. Corral-Plaza, I. Medina-Bulo, G. Ortiz, and J. Boubeta-Puig. A stream processing architecture for heterogeneous data sources in the internet of things. *Computer Standards & Interfaces*, 70:103426:1–103426:13, 2020.

[15] P. Delgado-Pérez, Louis M. Rose, and I. Medina-Bulo. Coverage-based quality metric of mutation operators for test suite improvement. *Software Quality Journal*, 27(2):823–859, 2019.

[16] G. Díaz, H. Macià, V. Valero, J. Boubeta-Puig, and F. Cuartero. An intelligent transportation system to control air pollution and road traffic in cities integrating CEP and colored Petri nets. *Neural Computing and Applications*, 32(2):405–426, 2020.

[17] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 38(2):278–292, 2012.

[18] P. Gómez-Abajo, E. Guerra, J. de Lara, and M. G. Merayo. A tool for domain-independent model mutation. *Science of Computer Programming*, 163:85–92, 2018.

[19] D. Griñán and A. Ibias. Generating tree inputs for testing using evolutionary computation techniques. In *22nd IEEE Congress on Evolutionary Computation, CEC'20*, pages E–24267: 1–8. IEEE Computer Society, 2020.

[20] D. Griñán, A. Ibias, and M. Núñez. Grammar-based tree swarm optimization. In *2019 IEEE Int. Conf. on Systems, Man and Cybernetics, SMC'19*, pages 76–81. IEEE Press, 2019.

[21] R. M. Hierons, M. G. Merayo, and M. Núñez. An extended framework for passive asynchronous testing. *Journal of Logical and Algebraic Methods in Programming*, 86(1):408–424, 2017.

[22] R. M. Hierons, M. G. Merayo, and M. Núñez. Bounded reordering in the distributed test architecture. *IEEE Transactions on Reliability*, 67(2):522–537, 2018.

[23] R. M. Hierons and M. Núñez. Implementation relations and probabilistic schedulers in the distributed test architecture. *Journal of Systems and Software*, 132:319–335, 2017.

[24] A. Ibias, R. M. Hierons, and M. Núñez. Using Squeeziness to test component-based systems defined as Finite State Machines. *Information & Software Technology*, 112:132–147, 2019.

[25] Y. Kamei and E. Shihab. Defect prediction: Accomplishments and future challenges. In *Leaders of Tomorrow Symposium: Future of Software Engineering, FOSE@SANER'16*, pages 33–45. IEEE Computer Society, 2016.

[26] J. Kennedy and R. Eberhart. Particle swarm optimization. In *3rd Int. Conf. on Neural Networks, ICNN'95*, pages 1942–1948. IEEE Computer Society, 1995.

[27] R. Lefticaru, R. M. Hierons, and M. Núñez. Implementation relations and testing for cyclic systems with refusals and discrete time. *Journal of Systems and Software*, 170:110738:1–110738:20, 2020.

[28] T. Menzies, J. Greenwald, and A. Frank. Data mining static code attributes to learn defect predictors. *IEEE Transactions on Software Engineering*, 33(1):2–13, 2007.

[29] M. G. Merayo, R. M. Hierons, and M. Núñez. Passive testing with asynchronous communications and timestamps. *Distributed Computing*, 31(5):327–342, 2018.

[30] M. G. Merayo, R. M. Hierons, and M. Núñez. A tool supported methodology to passively test asynchronous systems with multiple users. *Information & Software Technology*, 104:162–178, 2018.

[31] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. John Wiley & Sons, 3rd edition, 2011.

[32] N. T. Nguyen, D. Hwang, and E. Szczerbicki. Computational collective intelligence for enterprise information systems. *Enterprise IS*, 13(7-8):933–934, 2019.

[33] N. T. Nguyen, E. Szczerbicki, B. Trawinski, and V. D. Nguyen. Collective intelligence in information systems. *Journal of Intelligent and Fuzzy Systems*, 37(6):7113–7115, 2019.

[34] V. D. Nguyen and N. T. Nguyen. Intelligent collectives: Theory, applications, and research challenges. *Cybernetics and Systems*, 49(5-6):261–279, 2018.

[35] A. Núñez, P. C. Cañizares, M. Núñez, and R. M. Hierons. TEA-Cloud: A formal framework for testing cloud computing systems. *IEEE Transactions on Reliability (in press)*, 2020.

[36] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In *15th Int. Conf. on Software Engineering, ICSE'93*, pages 100–107. IEEE Computer Society / ACM Press, 1993.

[37] M. Papadakis, T. T. Chekam, and Y. Le Traon. Mutant quality indicators. In *13th Int. Workshop on Mutation Analysis, MUTATION'18, ICST Workshops*, pages 32–39. IEEE Computer Society, 2018.

[38] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman. Mutation testing advances: An analysis and survey. volume 112 of *Advances in Computers*, pages 275 – 378. Elsevier, 2019.

[39] J. Roldán, J. Boubeta-Puig, J. L. Martínez, and G. Ortiz. Integrating complex event processing and machine learning: An intelligent architecture for detecting iot security attacks. *Expert Systems with Applications*, 149:113251:1–113251:22, 2020.

[40] S. Selvaraj and E. Choi. Survey of swarm intelligence algorithms. In *3rd Int. Conf. on Software Engineering and Information Management, ICSIM'20*, pages 69–73. ACM Press, 2020.

[41] S. H. Tan, J. Yi, Y., S. Mechtaev, and A. Roychoudhury. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *39th Int. Conf. on Software Engineering, ICSE'17 Companion Volume*, pages 180–182. IEEE Computer Society, 2017.

[42] W. Visser. What makes killing a mutant hard. In *31st IEEE/ACM Int. Conf. on Automated Software Engineering, ASE'16*, pages 39–44. ACM Press, 2016.

[43] W. E. Wong and A. P. Mathur. Reducing the cost of mutation testing: An empirical study. *Journal of Systems and Software*, 31(3):185–196, 1995.