PIT-HOM: an Extension of Pitest for Higher Order Mutation Analysis

Thomas LAURENT
Complex Software Lab
University College Dublin
Belfield, Ireland
& Lero, The Irish Software Research Centre
thomas.laurent@ucdconnect.ie

Anthony VENTRESQUE
Complex Software Lab
University College Dublin
Belfield, Ireland
& Lero, The Irish Software Research Centre
anthony.ventresque@ucd.ie

***Abstract*—Mutation testing is a well-known, effective, fault-based testing criterion. First order mutation introduces defects in the form of a single small syntactic change. While the technique has been shown to be effective, it has some limits. Higher order mutation, where the faults introduced include multiple changes, has been proposed as a way to address some of these limits. Although the technique has shown promising results, there is no practical tool available for the application and study of higher order mutation on Java programs.**

In this paper we present PIT-HOM, an extension of Pitest (PIT) for higher order mutation. Pitest is a practical mutation analysis tool for Java, applicable on real-world codebases. PIT-HOM combines mutants in a same class to create higher order mutants of user-defined orders, it runs the mutants and reports the results in an easy to process format. We validate PIT-HOM using two small Java programs and report its performance as well as some characteristics of the mutants it creates.

***Index Terms*—Mutation analysis, Tool, Higher order mutation, Pitest**

I. INTRODUCTION

Mutation analysis is a well known-fault-based testing criterion. Mutation analysis creates mutant versions of the system under test (SUT), often by introducing a single syntactic change, and evaluates the capacity of a test suite to detect the difference between the SUT and the mutants. It has been the focus of much work in the last few decades [1] and has shown its effectiveness [2]. In particular, mutation analysis has been applied to the Java language, and many tools have been developed and made available for the mutation of Java programs [3]–[6]. The availability of these tools encouraged the development of the technique in both research and industry.

Despite all the work done on it, and the effectiveness of the technique, mutation analysis still suffers from several problems. The two main setbacks for the adoption of mutation analysis are its cost and the equivalent mutant problem. As mutation analysis creates a potentially large number of mutants against which tests must be executed, it is a computationally expensive testing criterion and often provides delayed feedback. The equivalent mutant problem refers to mutants that, although syntactically different from the original system, are semantically equivalent, and thus not detectable by any test. These equivalent mutants are not detectable automatically and are left to the tester to manually analyse, creating noise in the results of the mutation analysis and adding to the human cost of using the technique.

Higher order mutation analysis, where the injected faults are composed of several syntactic changes, has been proposed to alleviate these problems [7]. Although the technique has seen much interest in recent years [8], no practical tool for higher order mutation of Java programs has been made readily available. This absence of open tools hinders the progress of research and adoption of the technique in industry.

In this paper, we propose PIT-HOM, an extension for higher order mutation of the well-known Java mutation tool PIT [6]. PIT is an efficient tool for first order Java bytecode mutation that integrates with many modern build systems. PIT is used both in research [9], [10] and industry [11]. We extend PIT with higher order mutation capabilities, allowing the combination of single order mutants located in a same class. We validate using two small programs and make it available for further research and extension at https://github.com/ucd-csl/pitest.

The remainder of this paper is structured as follows. First we introduce background and related work on higher order mutation in Section II, then we describe PIT and our extension of the tool in Section III. We describe our validation of PIT-HOM and report our results in Section IV, and discuss the results in Section V. Finally, we conclude in Section VI.

II. BACKGROUND AND RELATED WORK

In this section, we introduce background and related work on higher order mutation. We first review the history of the technique, then its benefits, before mentioning previous tools developed for higher order mutation.

Higher order mutation analysis is an extension of classic, first order, mutation analysis. In classic mutation analysis, we only consider mutants consisting of one syntactic change, hence the term first order. Higher order mutation analysis considers mutants that are composed of multiple changes.

Historically, the field of mutation analysis was focused on First Order Mutants (FOMs). As the number of Higher Order Mutants (HOMs) is combinatorial to the number of first order mutants, it was considered that there were too many HOMs for them to be usable. Additionally, two firmly held beliefs worked against higher order mutation in the

common wisdom: the competent programmer hypothesis and the coupling effect. The competent programmer hypothesis states that, as programmers can be considered to be competent, they will not make big mistakes. This means that, even if a fault is introduced by a programmer, the faulty version of the program will be syntactically very close to the correct program. The coupling effect [12], [13] is the observation that tests designed to detect simple faults will detect many complex faults, i.e. that complex faults are coupled with simple ones. According to the competent programmer hypothesis and the coupling effect, single order mutation was enough: simple mutants would resemble the small faults made by developers and any complex faults would likely be coupled with the simple mutants anyway.

In [7], Harman et al. rehabilitate HOMs by tackling the "myths" of higher order mutation testing. First they argue that the competent programmer hypothesis is misinterpreted. Although they agree that programmers are competent, they say this means that programmers produce software that is close in *behaviour* to a correct program, and not software that is "within a few keystrokes of correctness". They also refer to studies [14]–[16] showing that many faults come from misunderstanding of requirements, and are larger than the hypothesis suggests. Then they say that, although the coupling effect stands, it is not a valid argument against using HOMs. In [13], they agree that "Complex faults are coupled to simple ones in such a way that test data which find all simple ones will detect a high percentage of complex faults.", but suggest that, as there are very many HOMs, having only a small percentage of uncoupled HOMs still makes them valuable in absolute numbers. The final myth against higher order mutation that the authors tackle is that the number of HOMs makes them unusable. They show that, using Search Based Software Engineering (SBSE), it is possible to navigate the very large search space that is the space of HOMs, and to effectively use higher order mutation analysis.

Harman et al. also explore an interesting property of HOMs in [7]: subsumption. They define a HOM as subsuming its component FOMs when the set of tests killing the HOM is smaller than the union of the sets of tests killing its component FOMs. A HOM strongly subsumes its component FOMs if the set of tests killing it is included in the intersection of the sets of tests killing its component FOMs. Based on these definitions, the authors propose a classification of HOMs as non- weakly- or strongly-subsuming. They further explore strongly subsuming higher order mutants and their effect on the mutation analysis process in [17]. They show that using subsuming HOMs instead of FOMs reduces the number of mutants to analyse by 35 to 45% while improving the test efficiency.

The improved efficiency provided by HOMs in [17] can be explained by HOMs in which constituent FOMs interact to create new behaviours. These new behaviours in turn exert the tests in new ways and improve the efficiency of a mutation-adequate test suite. Omar et al. explore the construction of such mutants in [18] and coin the phrase "subtle mutants" to refer to them. They find a large number of subtle HOMs for 10 projects, with subtle HOMs of order more than 8, showing how complex interactions between multiple mutants can generate new behaviours.

Although higher order mutation has been the focus of much work, only few tools are available that implement the technique. MILU [19] is a higher order mutation tool for the C language proposed by Jia et. al. Omar et. al. introduce HOMAJ [20], a tool for higher order mutation of Java code that integrates several search based mutant selection techniques. However, the tool is not publicly available. More recently, LittleDarwin [21], a higher order mutation tool for Java, was released, although the authors report longer mutation analysis run-times than with PIT. We were unable to find precise documentation regarding its HOM features on its website and to compare it to our tool.

III. PIT AND PIT-HOM

In this section, we describe how PIT and PIT-HOM carry out the mutation analysis process. We first review how PIT generates, manages and analyses mutants, then how this process was modified in PIT-HOM, and finally how to use PIT-HOM.

A. Mutation analysis process in PIT

PIT creates mutants at the bytecode level, instead of from the Java sources. PIT performs bytecode manipulation using the ASM library [22], an all purpose Java bytecode manipulation and analysis framework. This allows for significant performance advantages, as each mutant does not need to be recompiled, and bytecode manipulation is computationally inexpensive. Manipulating bytecode also allows for the mutants to never be stored to disk and to be fully stored and processed in main memory. In order to reduce memory cost, the full bytecode of a mutant is stored in memory only when the mutant is run, otherwise only metadata about the mutant is kept.

PIT carries out the mutation process in three stages, which are represented in figure 1. The first stage is mutant generation. All classes under analysis are scanned in a single pass to discover the available mutation points. During this pass, all possible mutants are created and stored in the form of a MutationIdentifier object, that uniquely identifies the mutant. This first pass actually generates the mutated bytecode, but it is immediately discarded, and only the mutants' metadata, the MutationIdentifier objects, are stored.

A MutationIdentifier object records the location in the SUT of a mutant and the mutation operator used to create the mutant. The location consists of the class, method and bytecode instruction number where the mutation happens. As PIT's mutation operators uniquely map a bytecode instruction to a mutated instruction, the position and operator are enough information to uniquely identify each mutant. Additional information, such as the tests to be run against the mutant are stored in a MutationDetails objects, linked to the MutationIdentifier object. Storing the mutants in this way makes them take very

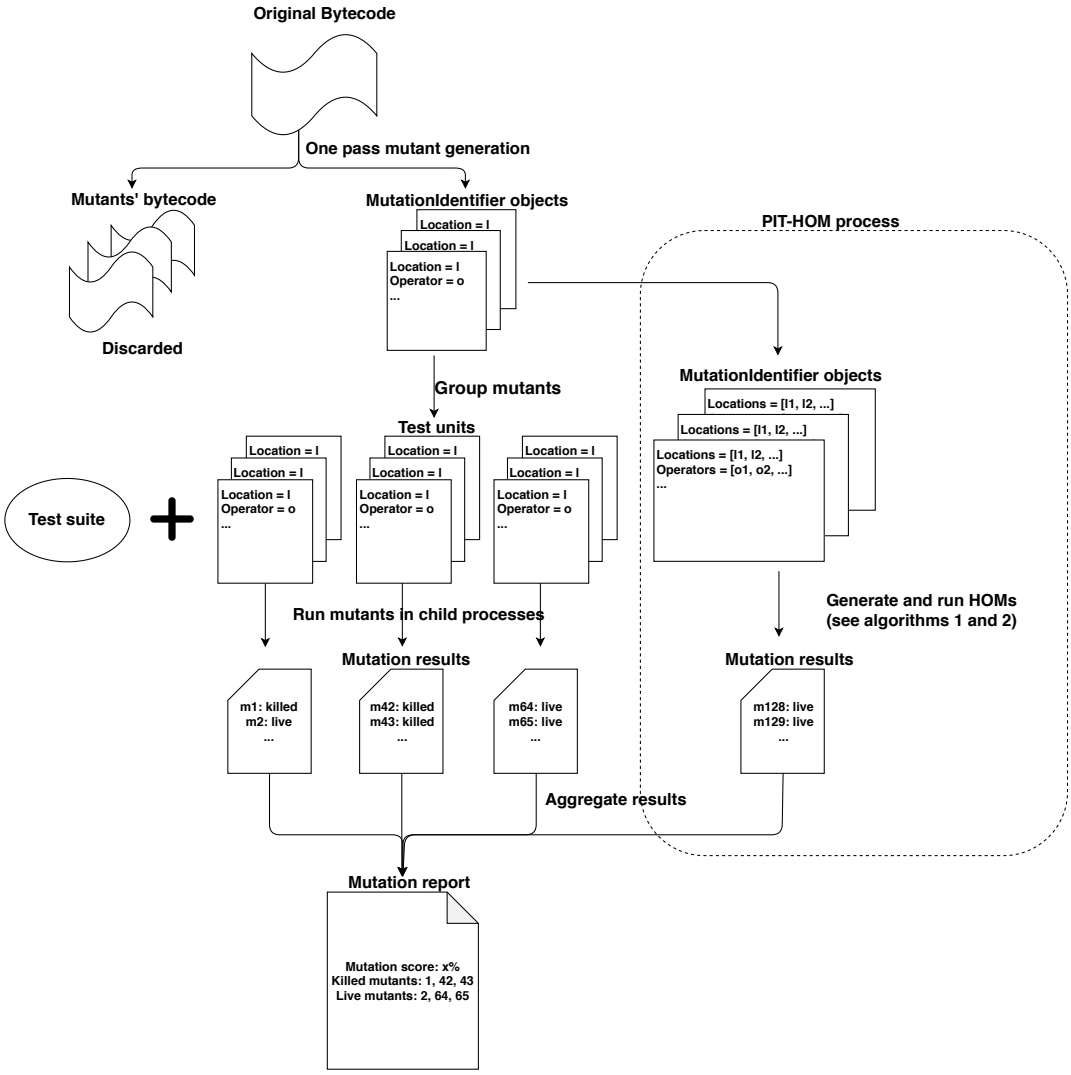

Fig. 1. Flowchart of the mutation process in PIT. Circled, the added steps in PIT-HOM

little memory and allows PIT to store the many mutants that can be generated, even on large projects.

Once all mutants have been generated, the second stage, mutant evaluation, can start. This process is performed in child JVM processes. The main process passes the Mutation-Identifier to the child process and it generates the bytecode corresponding to the mutant and injects it into the SUT. The tests targeting the mutant are then run against it to evaluate whether it is killed or not. This status is reported to the main process. By default, PIT will run tests against a mutant only until it is killed and will ignore further tests, but PIT provides an option to the user to run all tests that cover a mutant against this mutant, collecting the result of each test. This option, which creates a full mutation matrix, is particularly interesting in a research context.

As starting a new JVM child process is a very expensive operation, a natural approach would be to evaluate all mutants using the same JVM process. The problem with this approach is that executing tests against a mutant can modify the state of the JVM, i.e. "poison" the JVM, through modifying static variables for example. Such a poisoning of the JVM could modify the results of the analysis of other mutants, which is undesirable. A solution would be to run each mutant in a new JVM child process, but this proves extremely slow. PIT adopts a tradeoff where mutants are grouped in test units, and mutants of a same unit are run in the same JVM. By default, PIT groups mutants of a same class together, isolating the analysis of each class but allowing intra-class contamination of results. This behaviour can be changed by the user by specifying a maximum size of the test units. The user can then choose to follow the default behaviour, or take a safer approach with smaller test units, or even run each mutant in their separate JVM to get the most reliable results possible.

The final stage is reporting. The results collected in the evaluation stage are written to disk in a human-readable (HTML), or an easily processed (XML, CSV) report. These

reports contain information about the different mutants, as stored in the MutationIdentifier and MutationDetails objects, and on their status (killed or survived), as well as the potential killing test(s). The HTML report offers a visual and interactive view of the information, as well as information on basic code coverage of the tests.

B. Mutation analysis process in PIT-HOM

PIT-HOM extends PIT and thus also mutates bytecode, maintaining the same performance benefits. As PIT-HOM allows higher order mutation, where multiple mutations are performed in a single mutant, the overall mutation process had to be modified.

The first modification was to change the way mutants are represented in PIT. As a mutant can now consist of multiple mutations, the combination of a location and a mutation operators is no longer enough to identify it uniquely. The MutationIdentifier object was modified to hold a list of mutation locations and of operators. Similarly to the FOMs, this is enough information to uniquely identify any HOM, and is more effective than storing the bytecode of the mutant.

The MutationDetails object and its creation were also adapted. In particular, when considering a HOM, any test that covers any of the component FOMs is considered to cover the HOM. This approach considers every test that can potentially kill a HOM, although it includes many tests that only partially execute the HOM (i.e. do not execute all mutations points when run against the HOM). Considering only tests that fully execute a HOM might be more interesting but is non-trivial, as introducing mutations changes the coverage of the tests.

The process of mutant generation was also adapted for higher order mutation. The single pass that PIT performs reveals all mutation points and creates all possible FOMs. The FOMs then need to be combined to create HOMs. PIT-HOM creates all possible HOMs of the desired order in a class by combining any FOMs not on the same bytecode instruction, as combining two FOMs on the same instruction would mean the second FOM "overriding" the first.

As the number of HOMs is combinatorial to that of FOMs, there are very many possible HOMs, and holding them all in main memory at the same time becomes problematic. PIT-HOM thus breaks with PIT's process of creating all mutants. We propose two ways to carry out the mutation analysis process: streaming and batch-streaming. These two processes are summarised in Algorithms 1 and 2 and we compare them in Section IV.

In the streaming method (Algorithm 1), for each class under analysis PIT's classic pass is performed (line 2), creating all the FOMs. If first order is to be performed, the FOMs are processed, each in a separate JVM (lines 3-7). The mutants of the class are then combined to create the HOMs of each targeted order. As soon as a correct combination of FOMs is found, the HOM is created and processed in a new JVM (lines 8-13). Combinations of FOMs are found using the findNextCombination function, which takes the list of FOMs and a target order as input. This function enumerates all combinations of FOMs of the correct size and checks if the FOMs in the combination do not affect the same instruction. If each FOM in the combination affects a different instruction, they form a valid HOM and this HOM is created and returned. As the mutants are processed as soon as they are created they do not need to be all kept in memory at the same time. Furthermore, as the process of creating the FOMs and combining them into HOMs is often faster than running the tests, and as the run function is run in another thread and is thus non-blocking here, a guard is put in place to ensure that not too many HOMs are created and await processing (not shown in the algorithms). The streaming method allows the HOMs to be processed without running out of memory, as all mutants do not need to be held in main memory at the same time, but also requires each mutant to be processed in a new JVM, creating a significant overhead.

Algorithm 1 Mutation analysis process with streaming method

Input:
classesToAnalyse: List of classes that should be mutated
ordersToRun: List of mutation orders that should be run

```
1:  for class ∈ classesToAnalyse do
2:      foms ← MutationSoure.findMutants(class)
3:      if 1 ∈ ordersToRun then
4:          for mutant ∈ foms do
5:              run(new TestUnit(mutant))
6:          end for
7:      end if
8:      for order ∈ ordersToRun do
9:          hom ← findNextCombination(foms, order)
10:         while hom ≠ null do
11:             run(new TestUnit(mutant))
12:             hom ← findNextCombination(foms)
13:         end while
14:     end for
15: end for
```

The batch-streaming process (Algorithm 2) provides a balance between PIT's normal process of creating all mutants before starting evaluating them and the streaming method of evaluating each mutant separately. For each class, the FOMs are created, made into testing units, and evaluated as in PIT's normal process (lines 2-5). The HOMs are then created and stored until a set number (10,000 in our tests) have been created. The batch of HOMs is then turned into a testing unit (multiple testing units if a maximum unit size less than the batch size has been specified by the user) and processed in a new JVM (line 12). This process greatly reduces the number of JVMs created compared to the streaming method, thus reducing the overhead of JVM creation, while ensuring that the number of mutants to be kept in memory at once is reasonable.

Algorithm 2 Mutation analysis process with batch-streaming method

Input:
classesToAnalyse: List of classes that should be mutated
ordersToRun: List of mutation orders that should be run

```
1: for class ∈ classesToAnalyse do
2:     foms ← MutationSoure.findMutants(class)
3:     if 1 ∈ orderToRun then
4:         run(makeTestUnits(mutants, maxTestUnitSize))
5:     end if
6:     for order ∈ ordersToRun do
7:         hom ← findNextCombination(foms, order)
8:         homsToRun ← {}
9:         while hom ≠ null do
10:            homsToRun.add(hom)
11:            if homsToRun.size() == 10000 then
12:                run(makeTestUnit(homsToRun, maxTestUnitSize))
13:                homsToRun ← {}
14:            end if
15:            hom ← findNextCombination(foms)
16:        end while
17:        if homsToRun.size() > 0 then
18:            run(makeTestUnits(homsToRun, maxTestUnitSize))
19:        end if
20:    end for
21: end for
```

C. Using PIT-HOM

PIT-HOM follows the same workflow as PIT and is fully integrated to a variety of build tools, IDEs, and static code analysis tools. Thus there is no need for additional effort when one of these common tools is used.

PIT-HOM can be configured in the same way as PIT and accepts an additional option to specify the orders of mutation to be run which the user can pass as a list. An order can be run without running lower orders first. As of this writing PIT-HOM supports the CSV and XML report formats.

IV. Validation

In this section we describe how we validate PIT-HOM to show its capacity to perform higher order mutation, and to compare the different methods of performing the mutation process. We also briefly explore some of the properties of the HOMs created by the tool and the influence of the mutation operators used. We first describe the experiment we performed before reporting the results.

A. Experiment Setup

The experiment was performed using two small sized projects. The first project is Triangle [23], an example project used to demonstrate how to set up and use PIT and provided with PIT. The second project used is the Bisect project used in [24] that computes square roots. We use all the tests provided by Kintis et. al. in our evaluation.

Characteristics of the test subjects used in the validation are summarised in table I.

TABLE I
Name, number of source lines of code (SLOC) and number of tests of the projects used in the validation experiment

Project	SLOC	#tests
Triangle	69	12
Bisect	37	64

For each project, we run PIT-HOM, based on PIT-1.4.3, at different orders in order to evaluate the number of HOMs generated and the properties they have. We run PIT-HOM both using all operators available in PIT-1.4.3, and using them plus the extended set of operators proposed in [25]. When using the extended set of operators, overlapping operators between PIT's original operators and the extended ones are removed in order to avoid creating the same mutant multiple times.

We also run 3 configurations of PIT-HOM that implement the 3 mutation analysis processes described before: generating all mutants before running them, streaming and batch-streaming. This allows us to compare the performance of each method.

Finally we run the triangle project using PIT's full mutation matrix mode, where all tests covering a mutant are run against said mutant. This allows us to compute metrics such as the easiness of a mutant to be killed, i.e. the proportion of tests covering a mutant that kill it.

Computing the full mutation matrix could not be performed for both projects because of time constraints. Both projects were thus also run using PIT's usual stop condition of killing the mutant in order to collect comparable run times.

Our experiments were performed on a quad-core Intel Xeon processor (3.1GHz) with 8GB RAM running Debian 9.6 "stretch". PIT-HOM was run with the "threads" option set to 3.

B. Results

Table II reports the metrics concerning the mutants generated. For each project, we report the number of mutants generated at each order of mutation and the obtained mutation score, i.e. the proportion of mutants killed by the tests. The results are given for the two sets of mutation operators: the operators available in PIT 1.4.3 and the ones introduced in [25]. Mutation analysis at order 3 using the extended set of mutants was not run for the Bisect project because of time constraints.

For the triangle project we also report the average easiness of killing a mutant, i.e. the proportion of tests covering a mutant that kill it. This shows whether the created HOMs are trivial or not. We also report the size of the disjoint mutants set which, when compared to the size of the set of created mutants, shows how redundant the created mutants are. Disjoint mutants [26], or minimal mutants [27] are the set of mutants that subsume the full set of generated mutants.

TABLE II
METRICS ABOUT MUTANTS GENERATED AT EACH ORDER FOR EACH PROJECT WITH EACH SET OF OPERATORS

		Order 1		Order 2		Order 3	
		Original	Extended	Original	Extended	Original	Extended
Triangle	#mutants	94	401	4,282	70,007	125,950	10,302,692
	Mutation score	0.86	0.77	0.98	0.95	0.99	0.98
	Easiness to kill	0.23	0.18	0.41	0.35	0.47	0.42
	#disjoint mutants	10	16	66	66	66	66
Bisect	#mutants	31	209	445	21,913	3,939	Not evaluated
	Mutation score	0.90	0.89	0.99	0.99	0.76	Not evaluated

As in [25], we observe on both projects that the extended mutation operator set generates many more mutants than PIT's original set and that the mutants are harder to kill. This effect is also observed at higher orders of mutation. Results also indicate that the HOMs are in average harder to kill than the FOMs. This result is tied to our definition of a test covering a HOM.

The results also show for the triangle project that the mutants generated by the extended set are not subsumed by the ones generated by the original set of operators. At higher orders of mutation, as the triangle project is very small and only offers a few test, we see that the many HOMs created are mostly redundant and that there is some sort of saturation.

Table III reports the time taken, in seconds, to run the mutation analysis process using the different methods described above at different mutation orders and for both the original and extended sets of mutation operators. In this table MemErr means that we were not able to run the mutation analysis as PIT-HOM ran out of memory. Either the tool crashed or the children processes running the mutants did, polluting the results. The times in italic are approximations based on partial runs, the full runs were not performed because of time constraints.

On both projects, we observe that the extended set of operators, producing more mutants, takes longer to run than the original one. We also observe that the time taken to run the mutation analysis process rapidly increases when considering higher order mutants. In the triangle project we observe similar results for the generate all and batch-stream strategies when the number of mutants is low, as they will then both only generate a single child JVM process. Both methods perform better than the stream method. On the bisect project, generate all and batch-stream perform again very similarly, but this time are outperformed by stream. This is explained by the fact that a single class is under analysis and that a low number of mutants are generated. This means that the generate all and the batch stream methods only generate one testing unit, and thus execute all tests on the same thread, sequentially. On the other hand, the stream method creates a testing unit for each mutant, and can thus take advantage of the 3 threads allowed for PIT to use. As Bisect's tests are rather long to run (14 tests timed out out of 164 test executions at order one), the overhead introduced by creating the children JVM processes is compensated by the gain of running 3 mutants in parallel.

V. DISCUSSION

We validated PIT-HOM by running it against two well-known, small Java programs and have compared the different ways of performing the mutation process we proposed. As initial results seem to show that no one method is always preferable, a more in-depth evaluation is needed, using larger and more diverse subject programs.

PIT is a well engineered, mature tool, that allows for much extendability, but the changes made in this work were quite profound and not trivial. It is probable that opportunities for optimisation were missed and that PIT-HOM can be made more efficient. This is why we make our tool's code available at https://github.com/ucd-csl/pitest in hopes the community can both use and improve it.

Our results confirm the previous intuition that for higher order mutation to be usable and profitable, some kind of mutant selection process has to be used. Although we only considered two very small classes for our validation, the run time for second order mutation was already in the scale of hours for the Bisect project, and order 3 could not be run as it would take days.

The results of this small-scale experiment indicate that HOMs are largely redundant. This confirms the previous conclusion that careful selection is needed to capitalise on the benefits that HOMs bring, such as subsuming and subtle HOMs. Again, further investigation into the characteristics of the generated HOMs is needed, using larger and more diverse subjects.

VI. CONCLUSION

We presented PIT-HOM, an extension of PIT for higher order mutation of Java programs. PIT-HOM automates the generation and analysis of higher order mutants in Java bytecode. We make PIT-HOM and its source code freely available for the community to use it and expand on it.

We are extending PIT-HOM to use different mutant selection techniques and to allow for HOMs combining mutants of different classes. This will improve the performance of PIT-HOM, as less mutants will be considered, and will allow for more complex mutants to be created.

ACKNOWLEDGEMENT

This work was supported, in part, by Science Foundation Ireland grant 13/RC/2094 and co-funded under the European Regional Development Fund through the Southern & Eastern Regional Operational Programme to Lero - the Irish Software

TABLE III
TIME (IN S) TAKEN TO RUN EACH PROJECT WITH EACH MUTATION PROCESS METHOD AND WITH EACH SET OF OPERATORS

		Generate all		Stream		Batch-Stream	
	Order	Original	Extended	Original	Extended	Original	Extended
Triangle	1	2	6	23	99	2	6
	2	51	1,084	1,081	19,708	52	406
	3	1,880	MemErr	32,577	*35 days*	577	47,280
Bisect	1	75	397	42	191	73	372
	2	983	32,947	410	16,392	805	20, 665
	3	7,023	MemErr	3,086	*13 days*	6,467	N/A

Research Centre (www.lero.ie). Thomas Laurent is supported by an Irish Research Council grant (GOIPG/2017/1829).

REFERENCES

[1] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Transactions on Software Engineering*, vol. 37, no. 5, pp. 649–678, Sep. 2011.

[2] N. Li, U. Praphamontripong, and J. Offutt, "An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-Uses and Prime Path Coverage," in *2009 International Conference on Software Testing, Verification, and Validation Workshops*, Apr. 2009, pp. 220–229.

[3] Y. Ma, J. Offutt, and Y. Kwon, "MuJava: an automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, Jun. 2005.

[4] D. Schuler and A. Zeller, "Javalanche: Efficient Mutation Testing for Java," in *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE '09. New York, NY, USA: ACM, 2009, pp. 297–298.

[5] R. Just, F. Schweiggert, and G. M. Kapfhammer, "MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, Nov. 2011, pp. 612–615.

[6] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: A Practical Mutation Testing Tool for Java (Demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 449–452.

[7] M. Harman, Y. Jia, and W. B. Langdon, "A Manifesto for Higher Order Mutation Testing," in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, Apr. 2010, pp. 80–89.

[8] A. S. Ghiduk, M. R. Girgis, and M. H. Shehata, "Higher order mutation testing: A Systematic Literature Review," *Computer Science Review*, vol. 25, pp. 29–48, Aug. 2017.

[9] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, "Predictive Mutation Testing," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[10] L. Chen and L. Zhang, "Speeding up Mutation Testing via Regression Test Selection: An Extensive Study," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, Apr. 2018, pp. 58–69.

[11] M. Kirk, "Experience of introducing mutation testing to a team," accessed 21 January 2019. [Online]. Available: http://pitest.org/sky_experience/

[12] J. Offutt, "The Coupling Effect: Fact or Fiction," in *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification*, ser. TAV3. New York, NY, USA: ACM, 1989, pp. 131–140.

[13] ——, "Investigations of the Software Testing Coupling Effect," *ACM Trans. Softw. Eng. Methodol.*, vol. 1, no. 1, pp. 5–20, Jan. 1992.

[14] B. Cheng and J. Atlee, "From state of the art to the future of requirements engineering," *Future of Software Engineering*, 2007.

[15] R. S. Pressman, *Software engineering: a practitioner's approach*. Palgrave Macmillan, 2005.

[16] I. Sommerville, *Software Engineering (6th Ed.)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[17] M. Harman, Y. Jia, P. Reales Mateo, and M. so, "Angels and Monsters: An Empirical Investigation of Potential Test Effectiveness and Efficiency Improvement from Strongly Subsuming Higher Order Mutation," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14. New York, NY, USA: ACM, 2014, pp. 397–408.

[18] E. Omar, S. Ghosh, and D. Whitley, "Subtle higher order mutants," *Information and Software Technology*, vol. 81, pp. 3–18, Jan. 2017.

[19] Y. Jia and M. Harman, "MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language," in *Testing: Academic Industrial Conference - Practice and Research Techniques (taic part 2008)*, Aug. 2008, pp. 94–98.

[20] E. Omar, S. Ghosh, and D. Whitley, "HOMAJ: A Tool for Higher Order Mutation Testing in AspectJ and Java," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, Mar. 2014, pp. 165–170.

[21] A. Parsai, A. Murgia, and S. Demeyer, "LittleDarwin: A Feature-Rich and Extensible Mutation Testing Framework for Large and Complex Java Systems," in *Fundamentals of Software Engineering*, ser. Lecture Notes in Computer Science, M. Dastani and M. Sirjani, Eds. Springer International Publishing, 2017, pp. 148–163.

[22] E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: A code manipulation tool to implement adaptable systems," in *In Adaptable and extensible component systems*, 2002.

[23] H. Coles, "Triangle example project," accessed 21 January 2019. [Online]. Available: https://github.com/hcoles/triangle-example

[24] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, and N. Malevris, "Analysing and Comparing the Effectiveness of Mutation Testing Tools: A Manual Study," in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, Oct. 2016, pp. 147–156.

[25] T. Laurent, M. Papadakis, M. Kintis, C. Henard, Y. L. Traon, and A. Ventresque, "Assessing and Improving the Mutation Testing Practice of PIT," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Mar. 2017, pp. 430–435.

[26] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating mutation testing alternatives: A collateral experiment," in *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*. IEEE, 2010, pp. 300–309.

[27] P. Ammann, M. E. Delamaro, J. Offutt *et al.*, "Establishing theoretical minimal sets of mutants," in *IEEE International Conference on Software Testing, Verification, and Validation, 7*. IEEE Computer Society, 2014.