Assessing and Improving the Mutation Testing Practice of PIT

Thomas Laurent*[†], Mike Papadakis[‡], Marinos Kintis[‡], Christopher Henard[‡], Yves Le Traon[‡], Anthony Ventresque*

*Lero@UCD, School of Computer Science, University College Dublin, Ireland
[†]Ecole Centrale de Nantes, France
[‡]Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg
[†]thomas.laurent@eleves.ec-nantes.fr, *anthony.ventresque@ucd.ie, [‡]{firstname.lastname@uni.lu}

Abstract—Mutation testing is extensively used in software testing studies. However, popular mutation testing tools use a restrictive set of mutants which does not conform to the community standards and mutation testing literature. This can be problematic since the effectiveness of mutation strongly depends on the used mutants. To investigate this issue we form an extended set of mutants and implement it on a popular mutation testing tool named PIT. We then show that in real-world projects the original mutants of PIT are easier to kill and lead to tests that score statistically lower than those of the extended set of mutants for a range of 35% to 70% of the studied classes. These results raise serious concerns regarding the validity of mutation-based experiments that use PIT. To further show the strengths of the extended mutants we also performed an analysis using a benchmark with mutation-adequate test cases and identified equivalent mutants. Our results confirmed that the extended mutants are more effective than a) the original version of PIT and b) two other popular mutation testing tools (major and muJava). In particular, our results demonstrate that the extended mutants are more effective by 23%, 12% and 7% than the mutants of the original PIT, major and muJava. They also show that the extended mutants are at least as strong as the mutants of all the other three tools together. To support future research, we make the new version of PIT, which is equipped with the extended mutants, publicly available.

I. INTRODUCTION

Mutation testing is an established criterion [1] that promises to rigorously examine the programs under test. It operates by determining whether the candidate test cases can distinguish the behavior of the original program from the behavior of some altered program versions, which are called *mutants*. Mutants represent program defects and are used to assess test thoroughness (by computing the ratio of mutants that exhibit a different behavior from the original program). The technique is powerful because mutants simulate well the behavior of real faults [2], [3] and they lead to test cases that subsume almost all the other test criteria [4], [5], [1].

Because of its remarkable power, mutation testing is widely-used to support software testing experiments. This widespread use of the technique is due to the fact that it has "entered the mainstream" of practice [4] and the existing tool support. Indeed in recent years several robust mutation testing tools have been developed [6], [7], integrated with the most popular build systems and development tools [6], [7], which make mutation testing application easy and fast.

Unfortunately, modern mutation testing tools employ a restrictive set of mutants that does not fully conform to the recommendations made by the mutation testing literature. This fact indicates potential issues with the effectiveness of the tools given that mutation is sensitive to its mutants [1]. Since these tools are extensively used in Software Engineering studies (as shown by our literature review), it is mandatory to validate the strength of their mutant sets. Furthermore, previous research has demonstrated that existing mutation testing tools are incomparable and provide inconsistent results [7], [8]. This inconsistency highlights the need for a tool that subsumes the others and can reliably support experimentation.

This paper presents a thorough study investigating the above-mentioned issue using PIT [9]. We use PIT because it is one of the most popular mutation testing tool in the recent literature. PIT is open source and the most robust and easy to use tool [6]. A recent study indicates significant limitations of the mutants supported by PIT [7] and, thus, motivates the need for a more comprehensive set of mutants that we develop (referred to as *extended mutants*) and assess. Overall, our study shows that the extended mutants are stronger than those in the original version of PIT and that they provide better results (statistically significant) for a range of 35% to 70% of the classes of the projects we study. These results raise concerns regarding the validity of the studies that use PIT.

To further show the strengths of the extended mutant set, we also performed an additional analysis using a benchmark set with hand-analysed data; composed of mutation-adequate test suites and identified equivalent mutants. Our results demonstrate that the extended mutants are more effective than a) the original version of PIT and b) two other popular mutation testing tools (major and muJava). In particular our results demonstrate that 23%, 12% and 7% of the killable extended mutants are respectively not killed by the adequate test suites that were designed to kill the mutants of the original PIT, major and muJava mutation testing tools. At the same time tests killing the extended mutants also kill all the mutants generated by the other tools. Our results also show that the extended mutants are stronger even when all the mutants of all the other three tools are combined, i.e., merging their mutant sets. Therefore, we equipped PIT with an extended set of mutant operators.

II. TERMINOLOGY & BACKGROUND

A. Mutation Testing

Mutation analysis operates by injecting defects in the software under investigation. Thus, given a program, several variants of this program are produced, each one containing a defect. These are called *mutants* and they are made by altering (mutating) the code, either source code or executable binary code, of the program under test. The creation of mutants is based on syntactic rules, called *mutant operators*, that transform the syntax of the program. For example, an arithmetic mutant operator changes an instance of an arithmetic operator such as '+' to another one, such as '−'.

Mutants are used to measure test thoroughness by comparing the runtime behavior of the original, non-mutated, and the mutated programs. Thus, when some differences are found, we exhibit behavior discrepancies that are attributed to the ability of the used tests to project the syntactic changes of the mutants to its behavior, i.e., to show a semantic difference. When mutants exhibit differences in their behavior they are called *killed*. Those that do not exhibit differences are called *live*. Mutants might be live either because the employed test cases are not capable of killing them or because they are functionally equivalent to the original program. Mutants belonging to the latter case are called *equivalent* [10].

Mutation testing measures the number of mutants that are killed and calculates the ratio of those over the total number of mutants. This ratio represents the adequacy metric and is called *mutation score*.

B. Mutant Selection

Performing mutation testing using a specific set of operators was studied with programs written in Fortran [11] and in C [12]. As a result, Java mutation tools were built based on the findings of these studies. To address scalability, tool developers made further reductions. Thus, popular tools, like PIT [9], support a very small and restrictive set of mutants that does not follow any previous studies or practical experience.

The mutants supported by the current release of PIT have several shortcomings. One such example is the relational operator for which PIT replaces one instance of the operator by only another one, i.e., it mutates $<$ only to $<=$, or $<=$ only to $<$, or $>$ only to $>=$, or $>=$ only to $>$. However, this practice is not sufficient. Indeed, previous studies have shown that in this case, three mutants are needed to avoid a reduced effectiveness of the method [13], [14].

Although practical, mutant selection should not come at the expense of the method's effectiveness. Therefore, when using mutation for research purposes, it is mandatory to make sure that a representative mutant set is employed.

C. Disjoint Mutants

In literature, mutation testing is extensively used to support experimentation [15], [16] by quantifying the level of test thoroughness achieved by various testing methods. Thus, mutation score serves as a comparison basis between testing techniques and hence, to judge the more effective one. While this practice is quite popular in research [17], there is evidence suggesting that it can introduce severe problems, which can threaten the validity of the conducted research [17].

The problem is that not all mutants are of equal power [18], which means that some are useful and some are not. Indeed, mutants can be trivial, easy to kill, duplicated, equivalent or hard to kill. Those of the last category are of particular interest since they lead to strong tests [3], [18]. Hard to kill, trivial and easy to kill mutants are defined with respect to the employed test suite [18]. Thus, mutants killed by a small percentage of tests that exercise them are hard to kill, while those killed by a large one are easy to kill.

When using mutation as a basis for comparing testing methods, a filtering process that sweeps out the duplicated and equivalent mutants is needed [10]. However, this process is not adequate since in most cases many mutants tend to be killed jointly [17], [19]. Thus, they do not contribute to the test process despite being considered. This has an inflation effect on the mutation score computation since only a very small fraction of mutants contribute to the test process[1].

This issue was initially raised by Kintis *et al.* [19] who introduced the concept of *disjoint mutants*, i.e., a representative subset of mutants that is free of redundant ones. Their use is motivated by the same study which demonstrated that mutant sets suffer from the inflation problem caused by redundant mutants. Later Ammann *et al.* [20] introduced the concept of "minimum mutants", which forms the smallest possible representative subset of mutants, based on the notion of mutant subsumption and suggested it as a way to bypass the mutation score inflation problem. Papadakis *et al.* [17] demonstrated that there is a very good chance ($> 60\%$ for an arbitrary experiment) to come to a wrong conclusion when using all mutants (rather than only the disjoint mutants). Along these lines, Kurtz *et al.* [21] found that selective mutation performs well when evaluated with all the mutants but performs poorly when evaluated with the subsuming ones. These results imply that we need to take into account mutant redundancy when using mutation testing.

III. EXPERIMENTAL STUDY

A. Definition of the Experiment and Research Questions

Mutation analysis is typically used as supported by the existing mutation testing tools. However, a central role in mutation testing is played by the mutants that are used; meaning that the effectiveness of the method is sensitive to the mutants employed. Therefore, it is important to know whether the mutants used by popular mutation testing tools are suitable for test assessment.

To this end, we seek to assess the completeness of PIT mutants. This leads us to our first question:

> **RQ1 (Effectiveness).** Is there any effectiveness difference between the PIT mutants and the extended ones?

---

[1]Kintis *et al.* [19] report that this is 9% of mutants, for Java programs using the muJava mutation testing tool, Amman *et al.* [20] report 10% for the Java mutants of the muJava tool and 1% for the C mutants of the Proteum tool, Papadakis *et al.* [17] report a range from 0.3% to 1.7%, of C mutants.

We perform our experiments on real-world projects using their developer test suites augmented with automatically generated ones, using a random test generation tool called Randoop. To further strengthen our study, we also perform an additional experiment by repeating our analysis on a benchmark set with hand-analysed data of the PIT mutants, i.e., mutation-adequate test cases and identified equivalent mutants.

Mutation score is affected by the number of mutants, the number of equivalent mutants, the number of trivial and hard to kill mutants, all of which differ when comparing two sets of mutants. Thus, we perform an objective comparison, i.e., we measure the extent to which test cases selected based on one set of mutants (using the test selection process that is detailed in Section III-E) kill the killable mutants of the other set. Objective comparisons are common practice in software testing research, e.g., [3], [11], and they can assess the relative strengths of the two mutant sets by avoiding the influence of the above factors.

Recent research has shown that mutation score suffers from an inflation effect caused by trivial mutants [17], [19]. To circumvent this problem and to make a more robust empirical study, we measure the ratios of the disjoint mutants that are killed, [17], [20], [22] and report results based on the objective comparison scores using them.

Our study has focussed on PIT, however, a natural question to ask is how extended mutants compare with the mutants supported by other mutation testing tools. Mutation testing tools like muJava [23] and major [24] have been developed by researchers and thus, it is likely that their mutants are much stronger than those of PIT. Hence, we examine:

RQ2 (Other tools). Is there any effectiveness difference between the extended mutants and those supported by the muJava and the major mutation testing tools?

To answer RQ2 we repeat the analysis of RQ1 using a publicly available benchmark set that contains hand-analysed mutants of the muJava and major tools, i.e., mutation-adequate test cases and identified equivalent mutants (of both muJava and major) [7]. We also constructed a super-set of mutants based on those generated by all three used mutation testing tools (the original version of PIT, the muJava and major).

B. Subject Programs

Our experiments involve two program sets. The first set contains 4 Java projects, presented in Table I. We selected these programs since they have been used extensively in the recent mutation testing literature, e.g., [25], [26].

Joda-time is a date and time manipulation library. Jfreechart is a popular library for creating charts and plots. Jaxen is an engine for evaluating XPath expressions and Commons-lang is a set of utility methods for the commons classes of Java.

The second set (benchmark [7]) is composed of selected methods from 4 real-world programs (Commons-Math, Commons, Pamvotis and XStream) and two small ones (Triangle and Bisect). The benchmark is accompanied by a set of hand-analysed mutants of PIT, muJava and major, together with mutation-adequate test suites. We selected this benchmark

TABLE I
TEST SUBJECTS. THE LINES OF CODE (LoC) AND CLASSES ARE ONLY THOSE CORRESPONDING TO CLASSES HAVING TEST CASES.

Subjects	Version	LoC	Classes	Tests
joda-time	2.8.1	18,611	212	5,440
jfreechart	1.0.19	46,986	346	8,639
jaxen	1.1.6	6,790	160	12,490
commons-lang	3.3.4	16,286	200	10,839

since it provides a realistic playground with (adequate) test suites specifically developed using each one of the three mutation testing tools we study. For details regarding the benchmark, please refer to the study of Kintis *et al.* [7].

C. Experimental Environment and Used Tools

We use PIT v1.1.5 with its mutants and our implementation of the extended ones (detailed in the next section). To answer RQ2, we use major v1.1.8 [24] and muJava v3 [23]. Our experiments were performed on a quad-core Intel Xeon processor (3.1GHz) with 8GB RAM running Ubuntu 14.04.3 LTS (Trusty Tahr).

D. Employed Mutants

Table II details the studied mutants. PIT mutants (called 'common') are described in the upper part of the table while the extended ones are described in the lower part. The extended mutants were formed based on our experience, the discussions made during the Mutation 2014 and 2015 workshops, e.g., [27], and the literature. In particular, we adapt to Java the set of mutants that was suggested and used in the following studies [2], [3], [11]. Note that the extended mutants include all the common ones.

Special care was taken to reduce the duplicated mutant instances [10] by removing the overlap between the operators. It is also possible to reduce some redundancy when using weak mutation analysis. However, using such weak mutation analysis may degrade the effectiveness of the method in cases of mutants that cannot be propagated [14]. To avoid such a risk, we rely on disjoint mutants to remove redundancies among the mutants. Disjoint mutants were identified using the procedure proposed by Papadakis *et al.* [17].

E. Analysis Procedure for Answering the Research Questions

We use both mutation adequate and non-adequate test suites. In the case of adequate test suites we use a recent benchmark containing test suites designed (manually) to kill all the mutants supported by PIT, muJava and major [7]. Thus, we have three test suites that we merge and make a large one that we call the 'universe' suite. The universe suite was used to compute the disjoint mutants of the extended set. This practice results in an under-approximation of the killable mutants of the extended set with the unfortunate effect of under estimating their effectiveness. This is not a problem if we demonstrate that the extended mutants are more effective than the mutants of the other tools since their actual effectiveness will be higher. Note that the killable mutants of PIT, muJava

432

TABLE II
COMMON (PIT MUTANTS) AND EXTENDED MUTANTS.

	Name	Transformation	Example	Name	Transformation	Example
Common	Cond. Bound.	Replaces one relational operator instance with another one (single replacement).	$< \rightsquigarrow \leq$	**Return Values**	Transforms the return value of a function (single replacement).	`return 0 ⇝ return 1`
	Negate Cond.	Negates one relational operator (single negation).	$== \rightsquigarrow !=$	**Void Meth. Call**	Deletes a call to a void method.	`void m() ⇝`
	Remove Cond.	Replaces a cond. branch with true or false.	`if (...) ⇝ if (true)`	**Meth. Call**	Deletes a call to a non-void method.	`int m() ⇝`
	Math	Replaces a numerical op. by another one (single replacement).	$+ \rightsquigarrow -$	**Constructor Call**	Replaces a call to a constructor by null.	`new C() ⇝ null`
	Increments	Replace incr. with decr. and vice versa (single replacement).	$++ \rightsquigarrow --$	**Member Variable**	Replaces an assignment to a variable with the Java default values.	`a = 5 ⇝ a`
	Invert Neg.	Removes the negative from a variable.	$-a \rightsquigarrow a$	**Switch**	Replaces switch statement labels by the Java default ones.	
	Inline Const.	Replaces a constant by another one or increments it.	$1 \rightsquigarrow 0, a \rightsquigarrow a + 1$			
Extended	ABS	Replaces a variable by its negation.	$a \rightsquigarrow -a$	**OBBN**	Replaces the operators & by | and vice versa.	$a\&b \rightsquigarrow a\|b$
	AOD	Replaces an arithmetic expression by one of the operand.	$a + b \rightsquigarrow a$	**ROR**	Replaces the relational operators with another one. It applies every replacement.	$< \rightsquigarrow \geq, < \rightsquigarrow \leq$
	AOR	Replaces an artihmetic expression by another one.	$a + b \rightsquigarrow a * b$	**UOI**	Replaces a variable with a unary operator or removes an instance of an unary operator.	$a \rightsquigarrow a++$
	CRCR	Replaces a constant a with its negation, or with 1, 0, $a + 1$, $a - 1$.	$a \rightsquigarrow -a, a \rightsquigarrow a - 1$.	**Commons**	*All the common operators as described above.*	

TABLE III
NUMBER OF MUTANTS, KILLABLE MUTANTS AND MUTATION SCORE (MS) FOR THE COMMON (PIT MUTANTS) AND EXTENDED MUTANTS, PER CLASS.

Measurement		joda-time		jfreechart		jaxen		commons-lang	
		Common	Extended	Common	Extended	Common	Extended	Common	Extended
#Mutants	Min.	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Med.	100.00	262.00	98.00	259.00	25.00	46.00	32.00	60.00
	Mean	183.77	516.64	219.04	685.33	83.76	203.18	174.96	510.13
	Max.	1362.00	4,754	3,436.00	9,742.00	3,912.00	14,524.00	4,826.00	15,578.00
#Test Cases	Min.	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Med.	206.00	206.00	34.00	34.00	215.00	215.00	37.00	37.00
	Mean	565.68	565.68	256.06	256.06	1,238.61	1,238.61	267.00	267.00
	Max.	4,789.00	4,789.00	8,014.00	8,014.00	9,308.00	9,308.00	5,047.00	5,047.00
MS	Min.	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
	Med.	0.79	0.72	0.36	0.27	0.8	0.75	0.86	0.78
	Mean	0.73	0.67	0.41	0.33	0.67	0.63	0.75	0.69
	Max.	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

and major have been manually verified [7]. In the case of non-adequate test suites, we used the developer test suites augmented with a state-of-the-art test generation tool called Randoop [28] and applied our test selection procedure:

***Test selection** was performed, starting from empty sets, by incrementally adding random tests (from the whole test suites) that increase the mutation score. So, if a randomly selected test failed to kill any additional mutants, i.e., the test is redundant with respect to the employed mutants, the test was not included. This process mimics what testers do when they use mutation to improve their test suites [1], [3] and ensures that the selected tests are relevant to the mutants we study.* This a typical process followed by many studies [3], [11].

To compare the mutant sets, we use the *objective comparison score*. This is defined as the ratio of (measurement a) divided by (measurement b), where *(measurement a) is the number of the disjoint mutants of the extended set that are killed by the tests selected based on PIT mutants* and *(measurement b) is the number of the killable disjoint mutants of the extended set (approximated by using the whole test suites)*. The objective comparison score represents the disjoint mutation score of the PIT mutants' tests (selected based on PIT). The distance from value 1 on the objective comparison scores quantifies the ratio of the (disjoint) extended mutants that were not killed by the tests selected based on the PIT mutants. Since the tests were selected at random, this process was repeated 30 times. As a result, we obtain 30 instances of (measurement a) for every class of each project.

To answer RQ1 (in the case of large real-world projects), we record the number of classes for which there is a statistical and practical significance difference between (measurement a) and (measurement b) using a Wilcoxon test and the Vargha-Delaney effect size ($\hat{A}_{12}$). From the statistical test we obtain a p-value which in essence represents the probability that (measurement $b > a$). We consider the differences as statistically significant if they provide p-values lower than 0.05 and report the size of the differences ($\hat{A}_{12}$ values). We also record and visualise the objective comparison scores. For the case of the benchmark programs (with adequate test suites) we record the ratios of the disjoint mutants (of the extended set) that are

433

TABLE IV
NUMBER AND PROPORTION OF CLASSES WHERE EXTENDED MUTANTS
ARE STATISTICALLY SUPERIOR TO PIT (RQ1 - EFFECTIVENESS).

Subject	#Classes (proportion)	$\hat{A}_{12} > 0.50$	$\hat{A}_{12} > 0.75$
joda-time	148 (70%)	71%	63%
jfreechart	109 (32%)	35%	29%
jaxen	60 (38%)	39%	30%
commons-lang	92 (46%)	49%	43%

TABLE V
PIT, MAJOR AND MUJAVA MUTANTS VS EXTENDED MUTANTS WHEN
USING MUTATION ADEQUATE TEST SUITES (RQ1, RQ2).

Subject	PIT	major	muJava
gcd	78%	78%	67%
orthogonal	100%	100%	100%
toMap	50%	67%	83%
subarray	83%	100%	83%
lastIndexOf	76%	94%	94%
capitalize	67%	100%	100%
wrap	75%	84%	92%
addNode	94%	94%	94%
removeNode	71%	86%	100%
classify	62%	88%	100%
decodeName	89%	67%	100%
sqrt	80%	100%	100%
Average	**77%**	**88%**	**93%**

killed by the test suites designed to kill the PIT mutants.

To answer RQ2 we use a benchmark (with adequate test suites) and measure the disjoint mutation score of the extended mutants that is achieved by the test suites that were designed to kill all the mutants of muJava and major (for details refer to [7]). To further examine the strengths of the extended mutants, we compared them with a superset of mutants, constructed by merging the mutants of PIT (original version), muJava and major. Similarly to the process we used in RQ1 we computed the disjoint mutants of the superset and the extended set, using the universe test suite, and selected two test suites (one for the superset and one for the extended), using our test selection procedure. Based on these suites and disjoint mutants we compare the two mutant sets.

IV. RESULTS & ANSWERS TO THE RESEARCH QUESTIONS

A. RQ1 - Effectiveness

Table IV records the number and proportion of the classes for which there is a statistically significant difference between (measurement a) and (measurement b). As it can be seen, there is a significant difference that ranges from 35% to 70% of the project classes. The differences also have large ($\hat{A}_{12}$) effect sizes indicating that a significant number of faults introduced by the extended mutants are not encoded by the PIT mutants.

Figure 1 records the objective comparisons scores for the 4 programs. The objective comparison has been performed 30 times per class (represented on the x-axis), thus, yielding 30 different scores for each class of each program. The 4 colours of the plot represent the distribution of the 30 ordered scores according to the quartiles. Thus, from the lightest to the darkest colour, the first, second, third and fourth 25% of the resulting scores are represented. For instance, a light gray bar (the first quartile reaching 0.6) means that the lowest 25% of the 30 scores obtained are below or equal to 0.6. The black area represents the values above the third quartile, i.e., last 25% of the 30 scores. In that, if a bar is completely light gray, it means that most of the mutants killed by the tests are the same on both sets, while the presence of darker colours on the graph indicates that there are mutants missed by the tests.

The plots of Figure 1 reveal that there is a significant ratio of extended mutants that were missed by the tests of the common mutants. This is evident in all the examined cases. In the worst case, i.e., joda-time, there were too many mutants missed in more than half of the project classes. In particular, more than 30% of the extended mutants were missed in the majority of the classes. This indicates a major difference between the two sets of mutants.

That far in our results we only show results related to large real-world projects under-approximated by test suites (using both developer and automatically generated tests). While this practice is common in the literature, e.g., [26], [2], both developer and automatically generated test suites tend to give low mutation scores. This limits the differences between the examined mutant sets [2]. Therefore, to strengthen our study, we repeat our analysis on a benchmark set with hand-analysed data of the PIT mutants, i.e., mutation-adequate test cases and identified equivalent mutants. Thus, we use the benchmark test suites that were specifically designed to kill the PIT mutants and measure the ratio of the extended (disjoint) mutants that are killed by them. The results are recorded on the second column (named PIT) of Table V and show that the effectiveness of PIT ranges from 50% to 100% with an average value 77%. This indicates that, even when adequate test suites are used, extended mutants are much stronger (by 23%) than the original mutants.

Our results show that there is a significant difference between the PIT mutants and the extended ones suggesting that a large number of (killable) mutants, from the extended set, is not killed by the tests selected using the PIT mutants.

B. RQ2 - Comparison with major and muJava

This question concerns the comparison with other mutation testing tools when adequate test suites are used. Our results are recorded in Table V and show that the effectiveness of all three examined tools is much lower than the extended mutants. MuJava scores best with 93%, major follows with 88% and last is PIT with 77%. It is noted that the tests selected based on the extended mutants kill all the mutants of the three other tools, when considered either alone or altogether. When comparing the union of the mutants produced by all three tools with the extended ones we find that they are of equal power.

Our results show that the extended mutants are much stronger than those of PIT, major and muJava and at least as effective as using all these tools together.
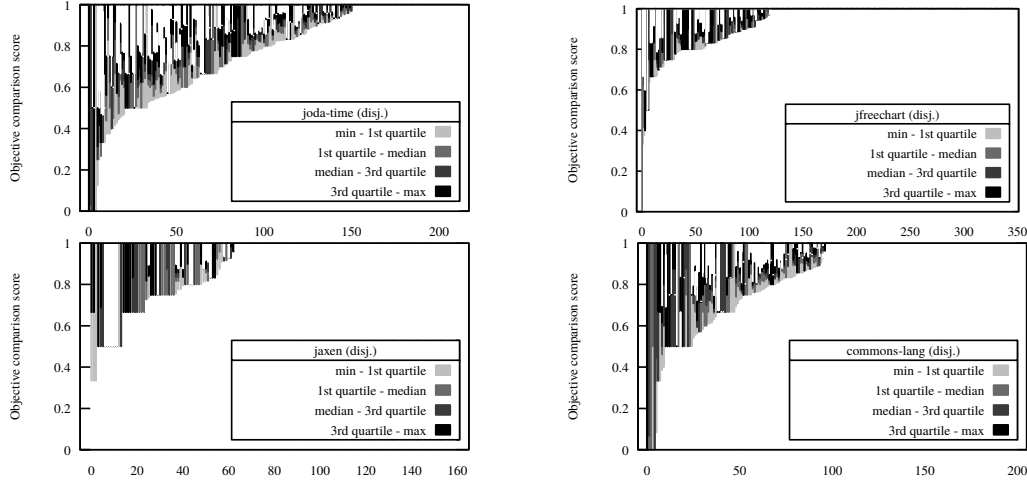
434

Fig. 1. Objective comparison results (RQ1 - effectiveness). The plots display the results of the disjoint mutants. The y-axis represents the distribution of the 30 scores per class, i.e., the minimum, first quartile, median, third quartile and maximum, while the x-axis represents the Java classes of the programs.

V. CONCLUSIONS

This paper investigates the validity of mutants used by popular mutation testing tools. Our study reveals a large divergence in the effectiveness of the PIT mutants from ours (called extended mutants). Extended mutants score considerably higher than those supported by the PIT, major and muJava mutation testing tools in most of the examined cases. To support future research, we augmented PIT, which is the most popular mutation testing tool, with the extended mutants. We make our version of PIT publicly available [9].

ACKNOWLEDGMENT

REFERENCES

[1] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2008.

[2] T. C. Thierry, M. Papadakis, Y. L. Traon, and M. Harman, "Empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *ICSE*, 2017.

[3] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria," *IEEE Trans. Softw. Eng.*, vol. 32, no. 8, pp. 608–624, 2006.

[4] J. Offutt, "A mutation carol: Past, present and future," *Information & Software Technology*, vol. 53, no. 10, pp. 1098–1107, 2011.

[5] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses vs Mutation Testing: an Experimental Comparison of Effectiveness," *Journal of Systems and Software*, vol. 38, no. 3, pp. 235–253, September 1997.

[6] M. Delahaye and L. du Bousquet, "Selecting a software engineering tool: lessons learnt from mutation analysis," *Softw., Pract. Exper.*, vol. 45, no. 7, pp. 875–891, 2015.

[7] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, and N. Malevris, "Analysing and comparing the effectiveness of mutation testing tools: A manual study," in *International Working Conference on Source Code Analysis and Manipulation*, 2016, pp. 147–156.

[8] R. Gopinath, I. Ahmed, M. A. Alipour, C. Jensen, and A. Groce, "Does choice of mutation tool matter?" *Software Quality Journal*, pp. 1–50, 2016.

[9] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "PIT: a practical mutation testing tool for java (demo)," in *ISSTA*, 2016, pp. 449–452.

[10] M. Papadakis, Y. Jia, M. Harman, and Y. L. Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *ICSE*, 2015, pp. 936–946.

[11] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An Experimental Determination of Sufficient Mutant Operators," *ACM T. Softw. Eng. Meth.*, vol. 5, no. 2, pp. 99–118, April 1996.

[12] A. M. R. Vincenzi, J. C. Maldonado, E. F. Barbosa, and M. E. Delamaro, "Unit and integration testing strategies for C programs using mutation," *Softw. Test., Verif. Reliab.*, vol. 11, no. 3, pp. 249–268, 2001.

[13] K.-C. Tai, "Theory of Fault-based Predicate Testing for Computer Programs," *IEEE Trans. Softw. Eng.*, vol. 22, no. 8, pp. 552–562, 1996.

[14] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution," in *ISSRE*, 2010, pp. 121–130.

[15] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?" in *ICSE*, 2005, pp. 402 – 411.

[16] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649 –678, 2011.

[17] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. L. Traon, "Threats to the validity of mutation-based test assessment," in *ISSTA*, 2016, pp. 354–365.

[18] Y. Jia and M. Harman, "Higher Order Mutation Testing," *Journal of Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, October 2009.

[19] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating mutation testing alternatives: A collateral experiment," in *APSEC*, 2010, pp. 300–309.

[20] P. Ammann, M. E. Delamaro, and J. Offutt, "Establishing theoretical minimal sets of mutants," in *ICST*, 2014, pp. 21–30.

[21] B. Kurtz, P. Ammann, J. Offutt, M. E. Delamaro, M. Kurtz, and N. Gökçe, "Analyzing the validity of selective mutation with dominator mutants," in *FSE*, 2016, pp. 571–582.

[22] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. L. Traon, "Comparing white-box and black-box test prioritization," in *ICSE*, 2016, pp. 523–534.

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

[24] R. Just, F. Schweiggert, and G. M. Kapfhammer, "MAJOR: an efficient and extensible tool for mutation analysis in a java compiler," in *ASE*, 2011, pp. 612–615.

[25] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *ICSE*, 2014, pp. 435–445.

[26] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Trans. Software Eng.*, vol. 38, no. 2, pp. 278–292, 2012.

[27] P. Amman, "Transforming mutation testing from the technology of the future into the technology of the present." [Online]. Available: https://sites.google.com/site/mutationworkshop2015/program/MutationKeynote.pdf?attredirects=0&d=1

[28] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *ICSE*, 2007, pp. 75–84.